ISO 29990:2010

NATIONAL INFOCOMM COMPETENCY FRAMEWORK

NUS National University of Singapore | ISS INSTITUTE OF SYSTEMS SCIENCE

Text Processing using Machine Learning

Recap

Liling Tan

2019

Course Schedule

Introduction
- Classic vs Deep NLP
- NN from Scratch

Deep Learning Foundations
- Matrix Calculus for Deep Learning
- Backpropagation

Word Embeddings
- Word2Vec, GloVe, Fasttext, and friends
- Word Embeddings from Scratch

Nuts and Bolts
- Bias - Variance
- Regularization, Loss Functions, Optimizers

Language Models
- N-gram + Neural Language Models
- Recurrent Neural Nets

Memory Networks and Conditional Generation
- Exploding and Vanishing Gradients
- LSTM + GRU and Seq2Seq Models

Attention Networks
- Lots of Different Attentions
- Attention is all you need (aka "Transformer")

Sentence Representation
- Multi-tasks vs Transfer Learning
- Pretrained Language Models (aka "Sesame Street")

Machine Translation (Bonus)
- Phrase-based to Neural MT
- 101 tricks to Neural MT training and quirks

Summary
- Recap
- Ask Me Anything

Lesson 1: Classic vs Deep NLP

Classic NLP

"***You shall know a word by the company it keeps…***"
– John R. Firth (1957)

"***Everyone is about Firth 1957 (you shall know a word…). Somehow we all skipped Firth 1935***"
– Yoav Goldberg (2016)

"***… the complete meaning of a word is always contextual, and no study of meaning apart from context can be taken seriously***"
– John R. Firth (1935)

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the TF-IDF features.
48  vectorizer = TfidfVectorizer(input=train_texts, analyzer='word',
49                               ngram_range=(2,2), # Note this parameter!
50                               min_df = 0, stop_words=None)
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Classic NLP: Feature Engineering

- **Raw frequency is useful**
  - *but frequent words non-content words are not very informative*
- **TF-IDF resolves high freq non-content words issue**
  - *but each word is still somewhat independent of each other*
- **PPMI provides information about whether a word is informative in the context of another word**
  - *but biased towards infrequent events*

Classic NLP: Feature Engineering

- **TF-IDF and PPMI vectors are**
  - long (|V| > 100,000)
  - sparse (lots of zero)

- **Deep learning can create vectors that are**
  - short (often fixed-sized <2000, decided empirically)
  - dense (most are non-zeros)

- **But it's not unlike *'modern'* deep learning based NLP**
  - one model improves upon another often incremental
  - they always come with certain caveats

Deep 'Magic' NLP

"***The frequencies of word in a document tend to indicate the relevance of document to a query***"
– Gerard Salton (1975)

"***From frequency to meaning…. Statistical patterns of human word usage can be used to figure out what people mean***"
– Turney and Pantel (2010)

"We propose a unified NN architecture by trying to avoid task-specific engineering therefore disregarding a lot of prior knowledge"
– Collobert and Weston (2011)

Definitions (NLP, ML, DL)

- **Natural Language Processing (NLP) is …**
  - making computers understand and produce human languages.

- **Machine Learning is …**
  - optimizing parameters/weights to best make a decision
  - *well-defined counting*

- **Deep Learning, some people say it's …**
  - neural nets
  - stacking multiple layers of "representation learning"
  - something that burns up as much GPUs as Bitcoin mining
  - a subset of methods in machine learning

Perceptron

Perceptron algorithm is a:

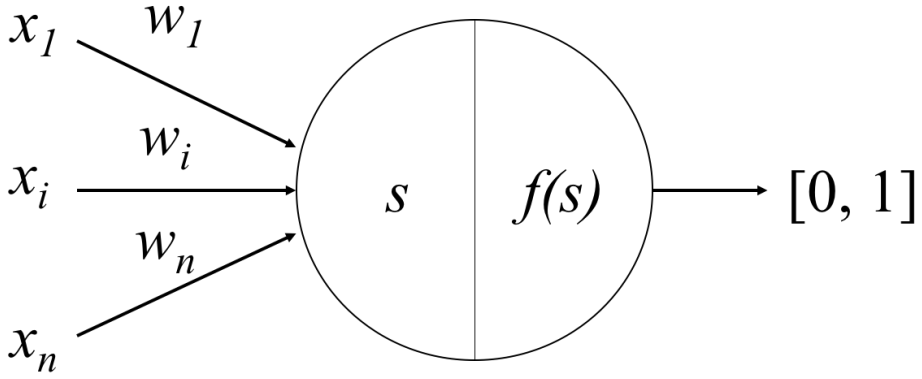
> "*system that depends on **probabilistic** rather than deterministic principles for its operation, gains its reliability from the **properties of statistical measurements obtain from a large population of elements***"
> - Frank Rosenblatt (1957)

```
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Perceptron

Given a **set of inputs x**, perceptron

- learns ***w* vector** to map the inputs to a real-value output between $[0,1]$
- through the **summation of the dot product of the *w*·*x***
- with a **transformation function** (aka. activation function)



Summation

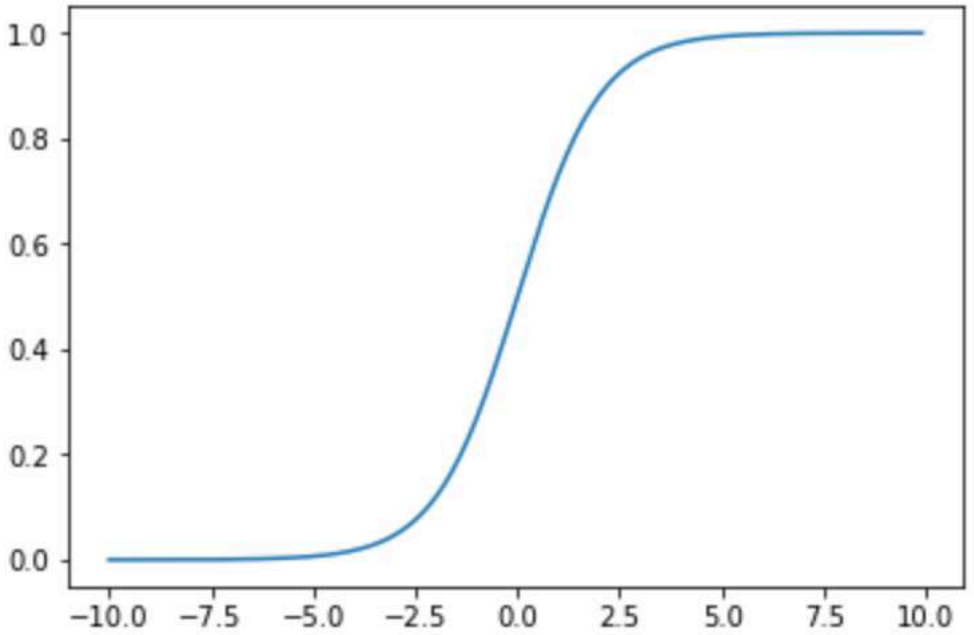
$$s = \sum w \cdot x$$

Transformation

$$f(s) = \frac{1}{1+e^{-s}}$$

Activation Function (Sigmoid)

```
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  def sigmoid(x):
7      return 1/(1+np.exp(-x))
8
9  # Generate points from -10 to +10,
10 # in steps of 0.1
11 x = np.arange(-10, 10, 0.1)
12 y = sigmoid(x)
13
14 # Plot the graph.
15 plt.plot(x, y)
16 plt.show()
```



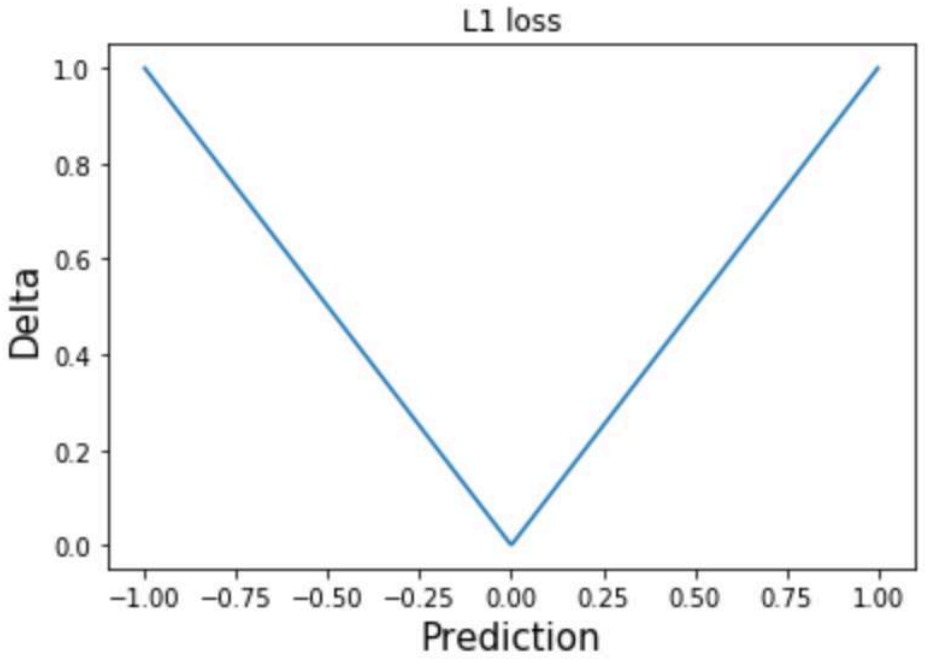
Loss Function (aka. Criterion)

For regression problems, the simplest loss function is to simply take the difference between the predictions and the truth value, i.e. the L1 Loss / Mean Absolute Error (MAE)

Loss Function (aka. Criterion)

```
38  import numpy as np
39
40  # Create 500 points betwen -1 and 1.
41  predictions = np.linspace(-1, 1., 500)
42  # Set truth to be the constant 0.
43  truth = np.zeros(500)
44  # Calcuate the absolute differences
45  delta = np.abs(truth - predictions)
46
47  # Plotting magic
48  plt.plot(predictions, delta, 'b-', label='L1 loss' )
49  plt.title('L1 loss')
50  plt.xlabel('Prediction', fontsize=15)
51  plt.ylabel('Delta', fontsize=15)
```

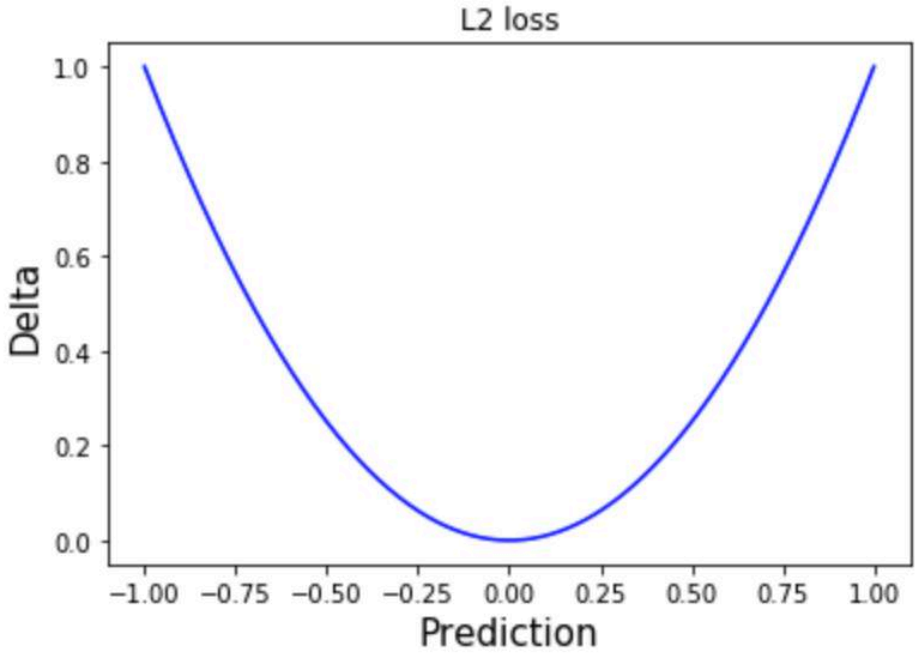
$$L_{mae} = \frac{1}{n} \sum_{i=1}^{n} |y - \hat{y}|$$



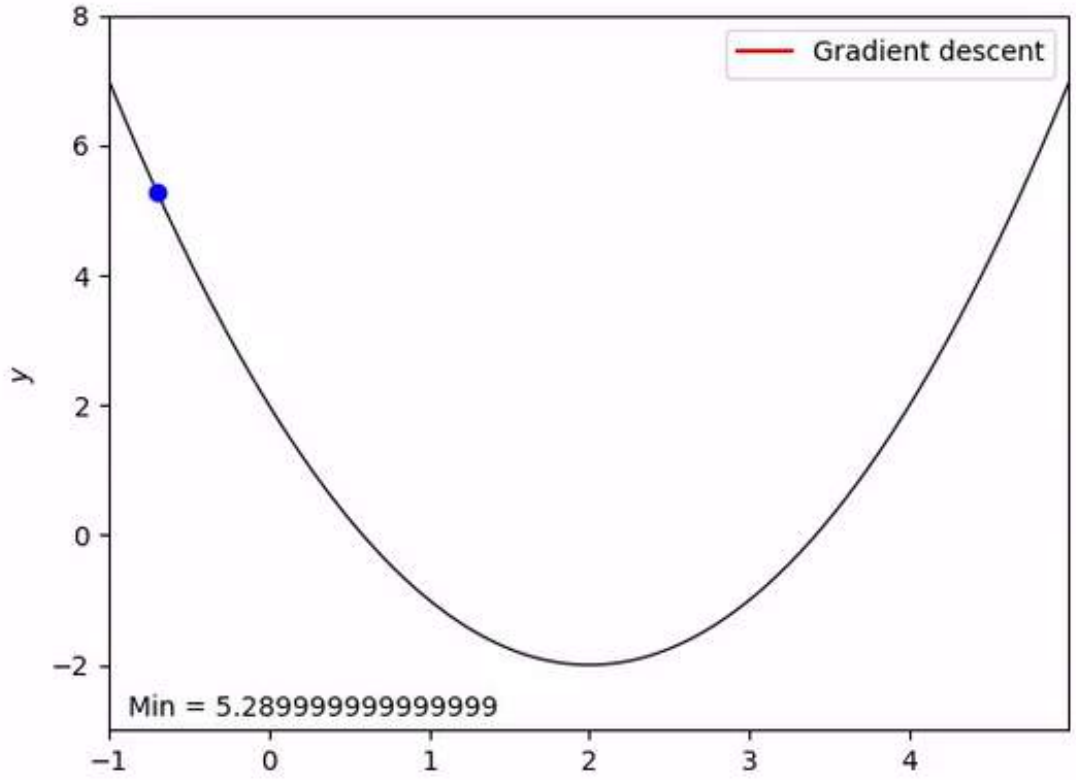
Loss Function (aka. Criterion)

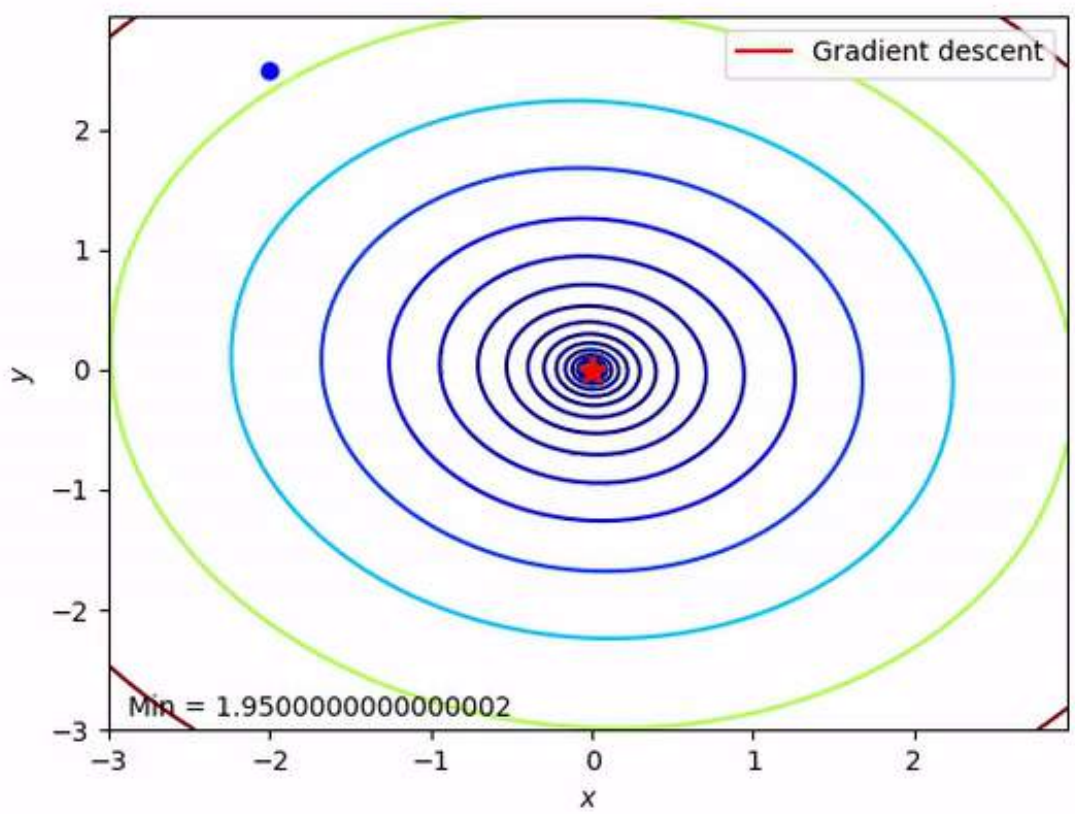
```
38  import numpy as np
39
40  # Create 500 points betwen -1 and 1.
41  predictions = np.linspace(-1, 1., 500)
42  # Set truth to be the constant 0.
43  truth = np.zeros(500)
44  # Calcuate the absolute differences
45  delta = np.abs((truth - predictions)**2)
46
47  # Plotting magic
48  plt.plot(predictions, delta, 'b-', label='L2 loss' )
49  plt.title('L2 loss')
50  plt.xlabel('Prediction', fontsize=15)
51  plt.ylabel('Delta', fontsize=15)
```

$$L_{mse} = \frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2$$



Optimization (Gradient Descent)





(Images from https://jed-ai.github.io/py1_gd_animation/)
It has some cool code to generate the GD pictures

Training Routine

Typically, process performs the following 4 steps iteratively.

Initialization

1. Initialize weights vector

Forward Propagation

2a. Multiply the weights vector with the inputs, sum the products.
2b. Put the sum through the activation function, e.g. sigmoid

Training Routine

Back Propagation

3a. Compute the errors, i.e. difference between expected output and predictions
3b. Multiply the error with the derivatives to get the delta
3c. Multiply the delta vector with the inputs, sum the product

Optimizer takes a step

4. Multiply the learning rate with the output of step 3c

Repeat 1-4 until *desired*

Training a PyTorch model

To train a model using PyTorch, we simply iterate through the no. of epochs and imperatively state the computations we want to perform.

Remember the steps?

1. Initialize
2. Forward Propagation
3. Backward Propagation
4. Update Optimizer

```
num_epochs = 7000
```

```
# Step 1: Initialization.
# Note: When using PyTorch a lot of the manual weights
#       initialization is done automatically when we define
#       the model (aka architecture)
model = nn.Sequential(
            nn.Linear(input_dim, output_dim),
            nn.Sigmoid())
criterion = nn.L1Loss()
learning_rate = 1.0
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 10000

losses = []
```

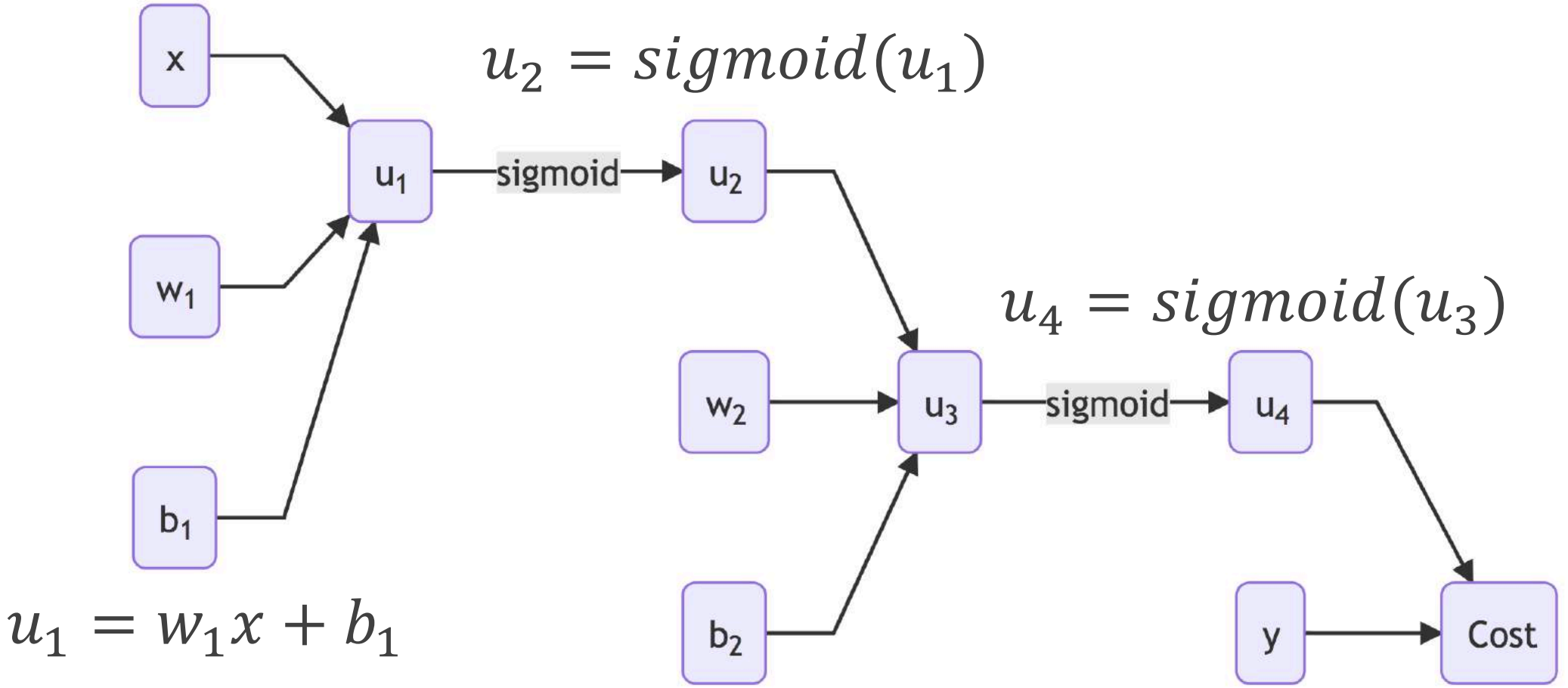
```
# Step 1: Initialization.
# Note: When using PyTorch a lot of the manual weights
#       initialization is done automatically when we define
#       the model (aka architecture)
model = nn.Sequential(
            nn.Linear(input_dim, output_dim),
            nn.Sigmoid())
criterion = nn.MSELoss()
learning_rate = 1.0
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 10000

losses = []

for i in tqdm(range(num_epochs)):
    # Reset the gradient after every epoch.
    optimizer.zero_grad()
    # Step 2: Foward Propagation
    predictions = model(X_pt)

    # Step 3: Back Propagation
    # Calculate the cost between the predictions and the truth.
    loss_this_epoch = criterion(predictions, Y_pt)
    # Note: The neat thing about PyTorch is it does the
    #       auto-gradient computation, no more manually defining
    #       derivative of functions and manually propagating
    #       the errors layer by layer.
    loss_this_epoch.backward()

    # Step 4: Optimizer take a step.
    # Note: Previously, we have to manually update the
    #       weights of each layer individually according to the
    #       learning rate and the layer delta.
    #       PyTorch does that automatically =)
    optimizer.step()

    # Log the loss value as we proceed through the epochs.
    losses.append(loss_this_epoch.data.item())

# Visualize the losses
plt.plot(losses)
plt.show()
```

Now, try again with 2 layers using PyTorch

```
%%time

hidden_dim = 5
num_data, input_dim = X_pt.shape
num_data, output_dim = Y_pt.shape

model = nn.Sequential(nn.Linear(input_dim, hidden_dim),
                      nn.Sigmoid(),
                      nn.Linear(hidden_dim, output_dim),
                      nn.Sigmoid())

criterion = nn.MSELoss()
learning_rate = 0.3
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 5000

losses = []

for _ in tqdm(range(num_epochs)):
    optimizer.zero_grad()
    predictions = model(X_pt)
    loss_this_epoch = criterion(predictions, Y_pt)
    loss_this_epoch.backward()
    optimizer.step()
    losses.append(loss_this_epoch.data.item())
    ##print([float(_pred) for _pred in predictions], list(map(int, Y_pt)), loss_this_epoch.data[0])

# Visualize the losses
plt.plot(losses)
plt.show()
```

Lesson 2: Deep Learning Foundations

Derivatives of a Multi-Layered Perceptron



$u_2 = sigmoid(u_1)$

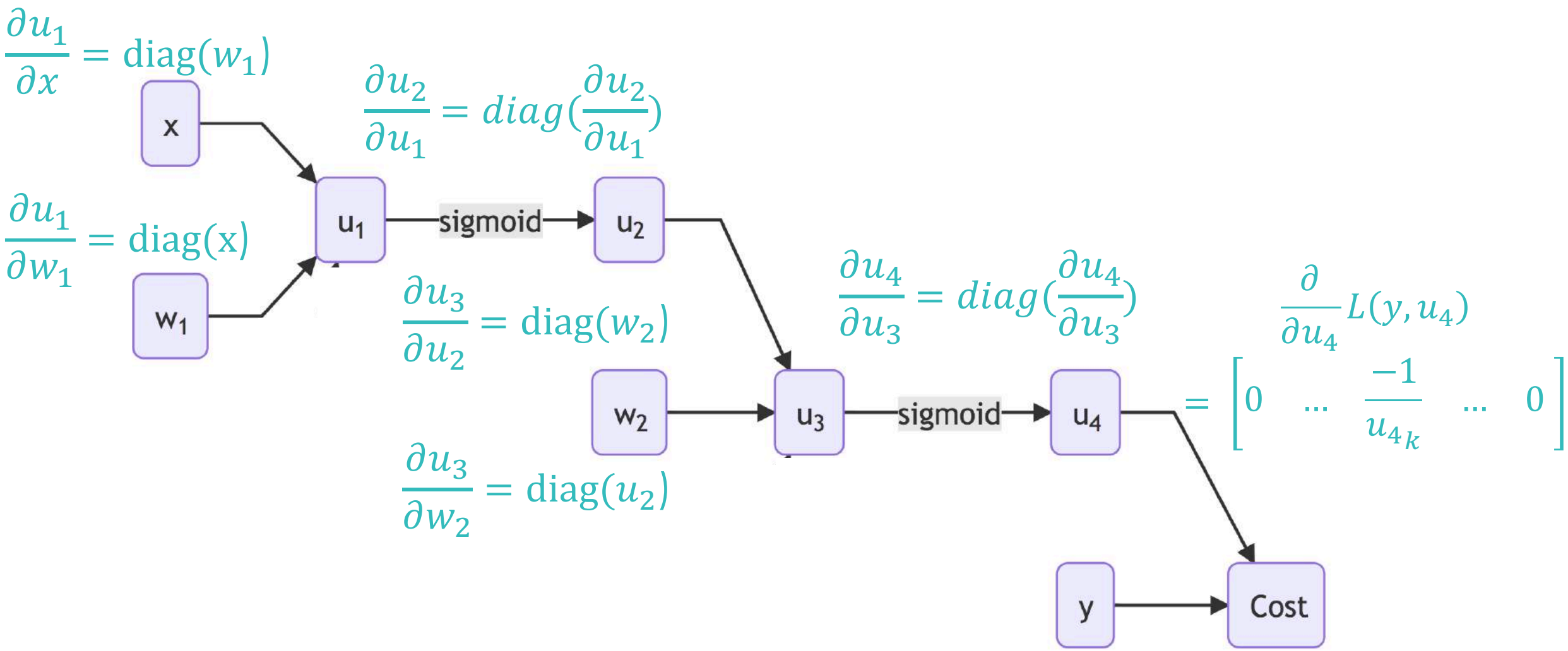
$u_4 = sigmoid(u_3)$

$u_1 = w_1 x + b_1$

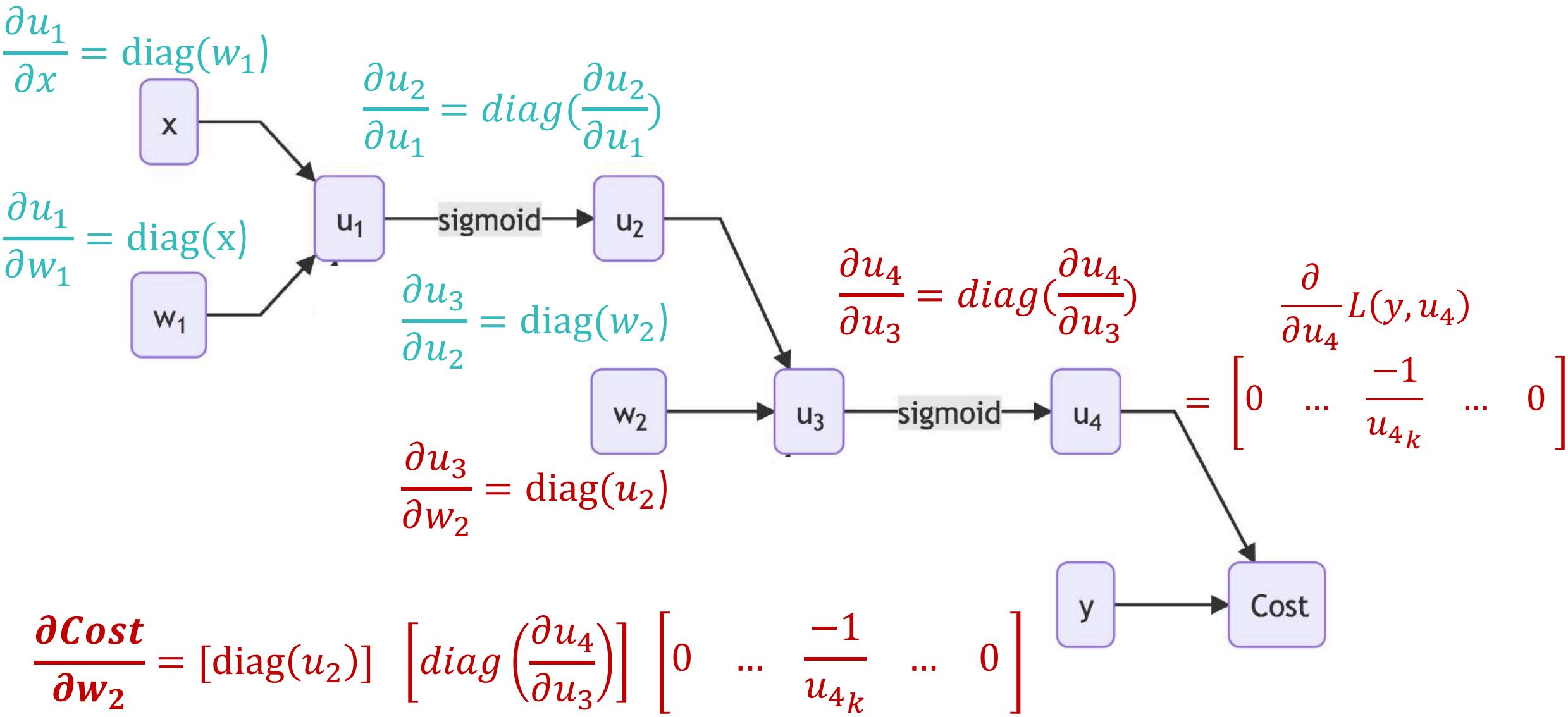
$u_3 = w_2 u_2 + b_2$

$cost = L(y, u_4)$

Derivatives of a Multi-Layered Perceptron

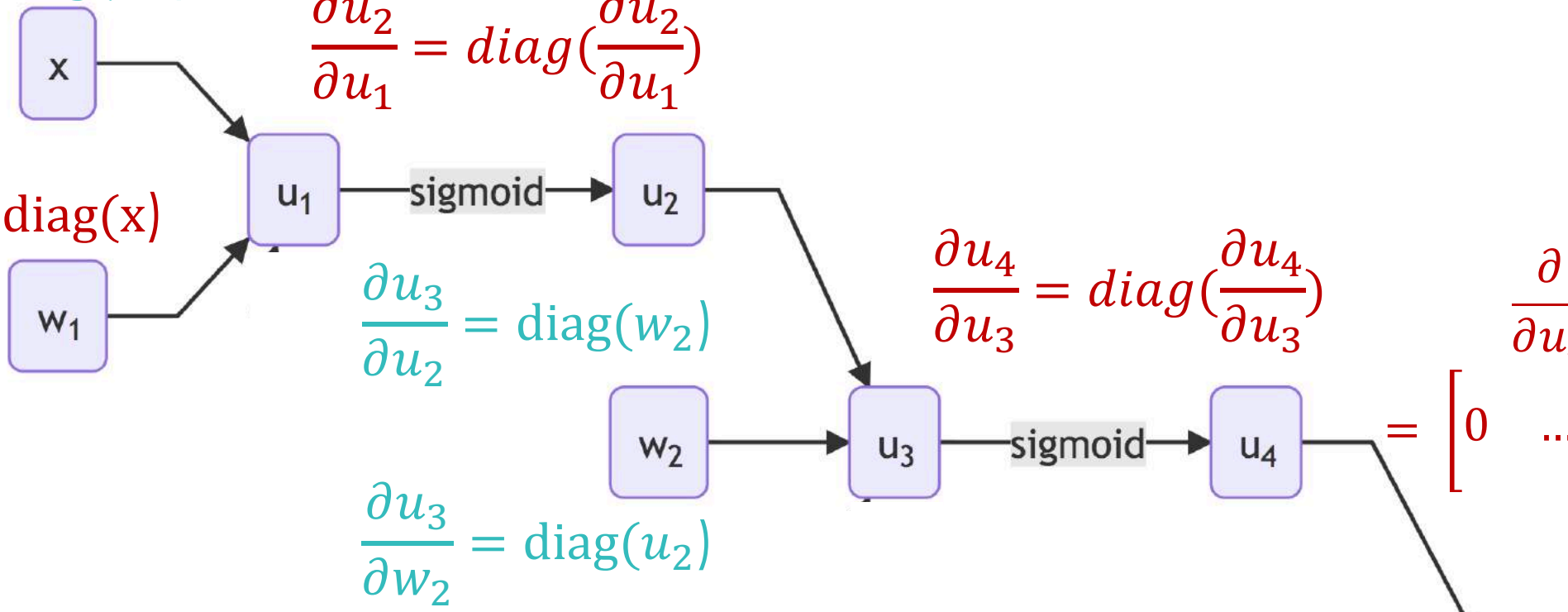


Derivatives of a Multi-Layered Perceptron



$$\frac{\partial Cost}{\partial w_2} = [\text{diag}(u_2)] \left[diag\left(\frac{\partial u_4}{\partial u_3}\right)\right] \left[0 \quad \dots \quad \frac{-1}{u_{4_k}} \quad \dots \quad 0\right]$$

Derivatives of a Multi-Layered Perceptron



$$\frac{\partial u_1}{\partial x} = \text{diag}(w_1)$$

$$\frac{\partial u_1}{\partial w_1} = \text{diag}(\text{x})$$

$$\frac{\partial u_2}{\partial u_1} = diag(\frac{\partial u_2}{\partial u_1})$$

$$\frac{\partial u_3}{\partial u_2} = \text{diag}(w_2)$$

$$\frac{\partial u_3}{\partial w_2} = \text{diag}(u_2)$$

$$\frac{\partial u_4}{\partial u_3} = diag(\frac{\partial u_4}{\partial u_3})$$

$$\frac{\partial}{\partial u_4} L(y, u_4) = \begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

$$\boldsymbol{\frac{\partial Cost}{\partial w_1}} = \text{diag}(\text{x})\ diag\left(\frac{\partial u_2}{\partial u_1}\right)\ diag\left(\frac{\partial u_4}{\partial u_3}\right) \begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

Derivatives of a Multi-Layered Perceptron

$$\frac{\partial Cost}{\partial w_1} = \text{diag(x)}\ diag\left(\frac{\partial u_2}{\partial u_1}\right)\ diag\left(\frac{\partial u_4}{\partial u_3}\right)\begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

$$\frac{\partial Cost}{\partial w_2} = \text{diag}(u_2)\ diag\left(\frac{\partial u_4}{\partial u_3}\right)\begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

$$w_2 \mathrel{+}= -lr * \frac{\partial Cost}{\partial w_2}$$

$$w_1 \mathrel{+}= -lr * \frac{\partial Cost}{\partial w_1}$$

Lesson 3: Word Embeddings

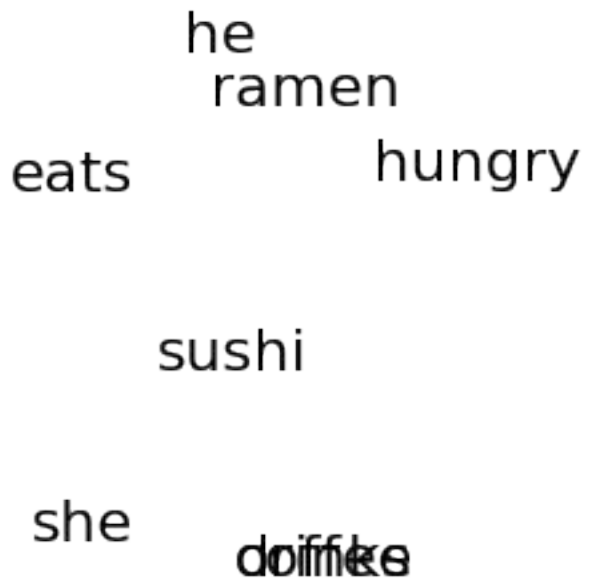
Classic NLP: Feature Engineering

- **Count-based vectors are**
  - e.g. TF-IDF, PPMI
  - long (|V| > 100,000)
  - sparse (lots of zero)

- **Vector compression** (aka ***dimensionality reduction***)
  - shorter vectors easier to use as features in machine learning
  - ***compression*** use to make vectors short and dense, e.g. Singular Value Decomposition (SVD), Non-negative Matrix Factorization (NMF)

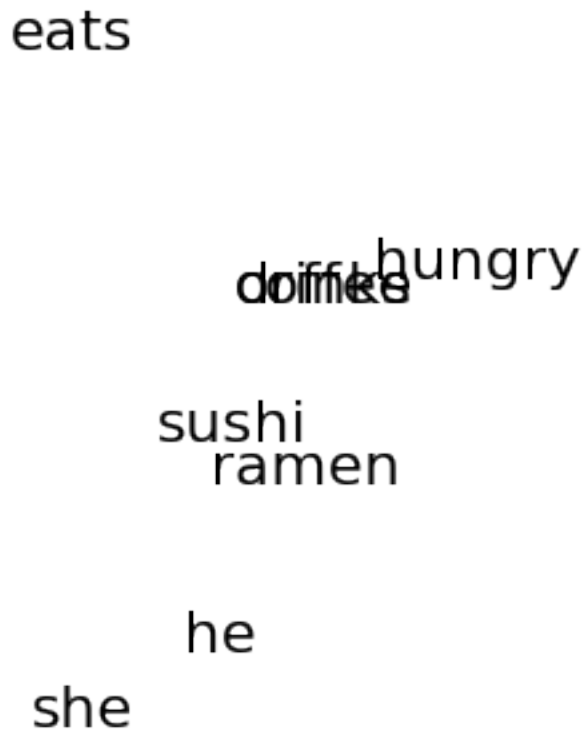
Dimensionality Reduction (SVD)

```
corpus = ['he eats ramen', 'she eats sushi',
          'he hungry', 'she drinks coffee']

# Get the standard term-doc matrix
count_model = CountVectorizer(ngram_range=(1,1))
X = count_model.fit_transform(corpus)
# Multiplying the term-doc matrix with itself
# produces the co-occurence matrix.
X_cooc = (X.T * X)
X_cooc.setdiag(0)   # set co-occurence with self to 0.

U, s, Vh = np.linalg.svd(X_cooc.todense(),
                         full_matrices=False)

words = sorted(count_model.vocabulary_,
               key=count_model.vocabulary_.get)

# Plot pretty words.
for i in range(len(words)):
    plt.text(U[i,0], U[i,1], words[i], fontsize=22)
plt.axis('off')
plt.show()
```



Dimensionality Reduction (SVD)

```
corpus = ['he eats ramen', 'she eats sushi',
          'he hungry', 'she drinks coffee']

# Get the standard term-doc matrix
count_model = CountVectorizer(ngram_range=(1,1))
X = count_model.fit_transform(corpus)
# Multiplying the term-doc matrix with itself
# produces the co-occurence matrix.
X_cooc = (X.T * X)
X_cooc.setdiag(0)   # set co-occurence with self to 0.

U, s, Vh = np.linalg.svd(X_cooc.todense(),
                         full_matrices=False)

words = sorted(count_model.vocabulary_,
               key=count_model.vocabulary_.get)

# Plot pretty words.
for i in range(len(words)):
    plt.text(U[i,0], U[i,2], words[i], fontsize=22)
plt.axis('off')
plt.show()
```



Count-based Vectors

tokenization
annotation
tagging
parsing
feature selection
⋮ cluster texts by date/author/discourse context/...
↓ ↙

Matrix type		Weighting		Dimensionality reduction		Vector comparison
word × document		probabilities		LSA		Euclidean
word × word		length normalization		PLSA		Cosine
word × search proximity	×	TF-IDF	×	LDA	×	Dice
adj. × modified noun		PMI		PCA		Jaccard
word × dependency rel.		Positive PMI		IS		KL
verb × arguments		PPMI with discounting		DCA		KL with skew
⋮		⋮		⋮		⋮

(Nearly the full cross-product to explore; only a handful of the combinations are ruled out mathematically, and the literature contains relatively little guidance.)

Potts (2013)

Don't Count, Predict!

	rg	ws	wss	wsr	men	toefl	ap	esslli	battig	up	mcrae	an	ansyn	ansem
best setup on each task														
cnt	74	62	70	59	72	76	66	84	98	41	27	49	43	60
pre	84	75	**80**	**70**	**80**	91	75	86	**99**	41	28	**68**	**71**	**66**
best setup across tasks														
cnt	70	62	70	57	72	76	64	84	98	37	27	43	41	44
pre	83	73	78	68	**80**	86	71	77	98	41	26	67	69	64
worst setup across tasks														
cnt	11	16	23	4	21	49	24	43	38	-6	-10	1	0	1
pre	74	60	73	48	68	71	65	82	88	33	20	27	40	10
best setup on rg														
cnt	(74)	59	66	52	71	64	64	84	98	37	20	35	42	26
pre	(84)	71	76	64	79	85	72	84	98	39	25	66	70	61
other models														
soa	**86**	**81**	77	62	76	**100**	**79**	**91**	96	**60**	**32**	61	64	61
dm	82	35	60	13	42	77	76	84	94	51	29	NA	NA	NA
cw	48	48	61	38	57	56	58	61	70	28	15	11	12	9

Table 2: Performance of count (cnt), predict (pre), dm and cw models on all tasks. See Section 3 and Table 1 for figures of merit and state-of-the-art results (soa). Since dm has very low coverage of the an* data sets, we do not report its performance there.

Baroni et al. (2014)

Word Embeddings: Not New, but Different

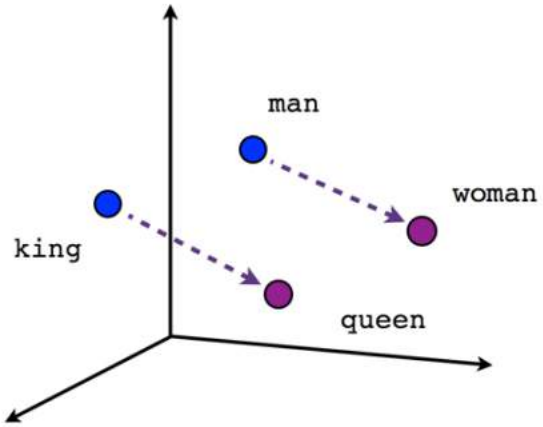
- Learning representations by back-propagating errors (Rumelhart et al. 1986)
- Neural Probabilistic Language Model (Bengio et al. 2003)
- NLP (almost) from Scratch (Collobert and Weston, 2008)
- Word2Vec (Mikolov et al., 2013)

Deep NLP: Featurize and Predict

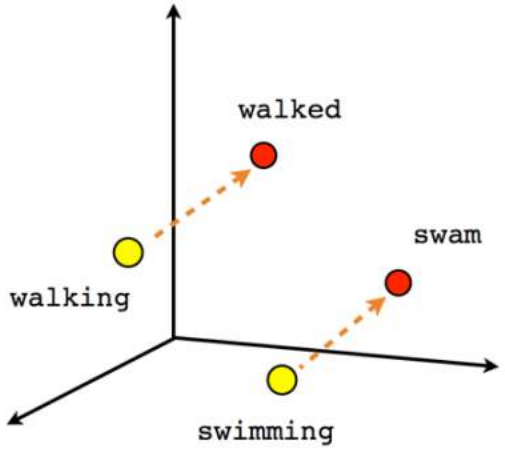
- **Deep learning can create vectors that are**
  - short (often fixed-sized <2000, decided empirically)
  - dense (most are non-zeros)

- **How might we "featurize" the vectors through some tasks and update the vectors based on gradient descent?**

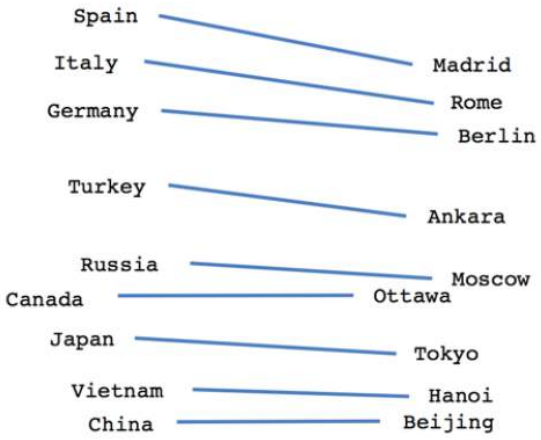
Word2Vec



Male-Female



Verb tense



Country-Capital

Ingredients

Corpus of text	As large as possible
Annotations	0
Initialize weights (aka Embeddings)	1x per word
Deep Learning Model	1x
Cost Function	Appropriately
GPU	Lotsa of it

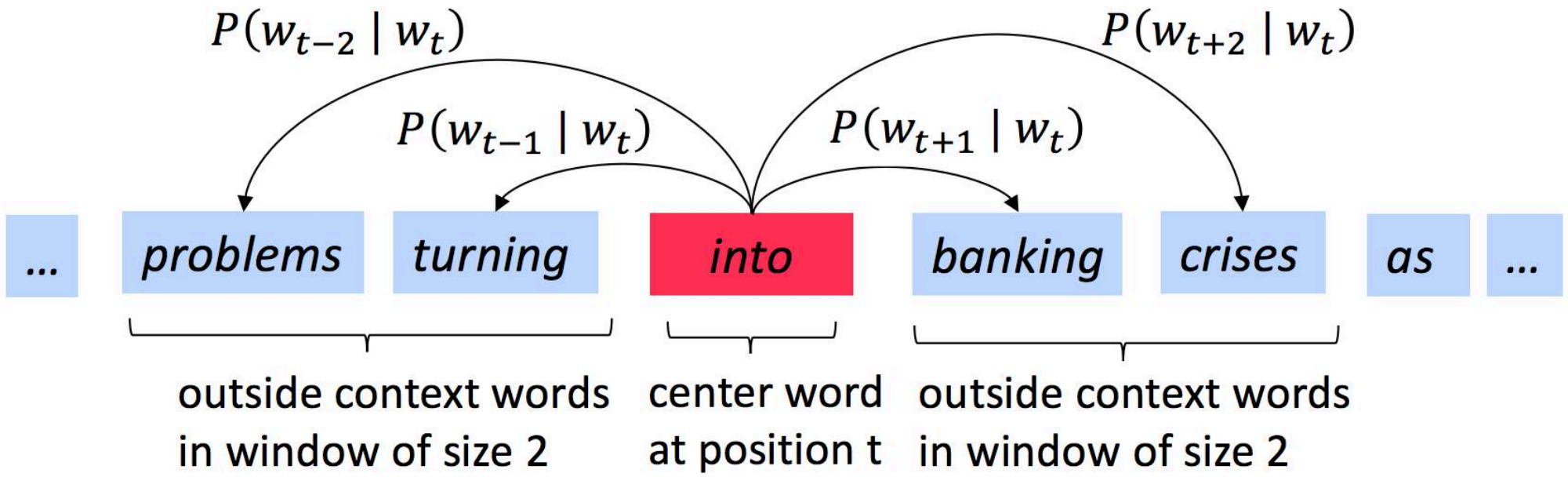
Word2Vec

Steps

1. Define task that we want to predict
2. Go through each sentence and create the task's in-/outputs
3. Iterate through task's I/O, put the inputs through the embeddings and models to create predictions
4. Measure cost of the predicted and expected output
5. Update embedding weights accordingly (*backprop)
6. Repeat Step 3-5 until desired.

Word2Vec (CBOW)

Task: Iterate through each word with a given window; for each word predict the context words within the window



(E.g. from Manning (2018) Stanford cs224n course)

Word2Vec (CBOW)

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size m, given center word w_j.

Likelihood = $$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P\left(w_{t+j} \mid w_t; \theta\right)$$

θ is all variables to be optimized

sometimes called *cost* or *loss* function

The objective function $J(\theta)$ is the (average) negative log likelihood:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P\left(w_{t+j} \mid w_t; \theta\right)$$

Minimizing objective function $\Leftrightarrow$ Maximizing predictive accuracy

Word2Vec (CBOW)

We want to minimize the objective function:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\leq j\leq m \\ j\neq 0}} \log P\left(w_{t+j} \mid w_t; \theta\right)$$

Question: How to calculate $P\left(w_{t+j} \mid w_t; \theta\right)$?

Answer: We will *use two* vectors per word *w*:

- v_w when *w* is a center word
- u_w when *w* is a context word

Then for a center word *c* and a context word *o*:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w\in V}\exp(u_w^T v_c)}$$

Word2Vec (CBOW)

Exponentiation makes anything positive

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Dot product compares similarity of *o* and *c*.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

Normalize over entire vocabulary
to give probability distribution

- This is an example of the **softmax function** $\mathbb{R}^n \rightarrow \mathbb{R}^n$

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

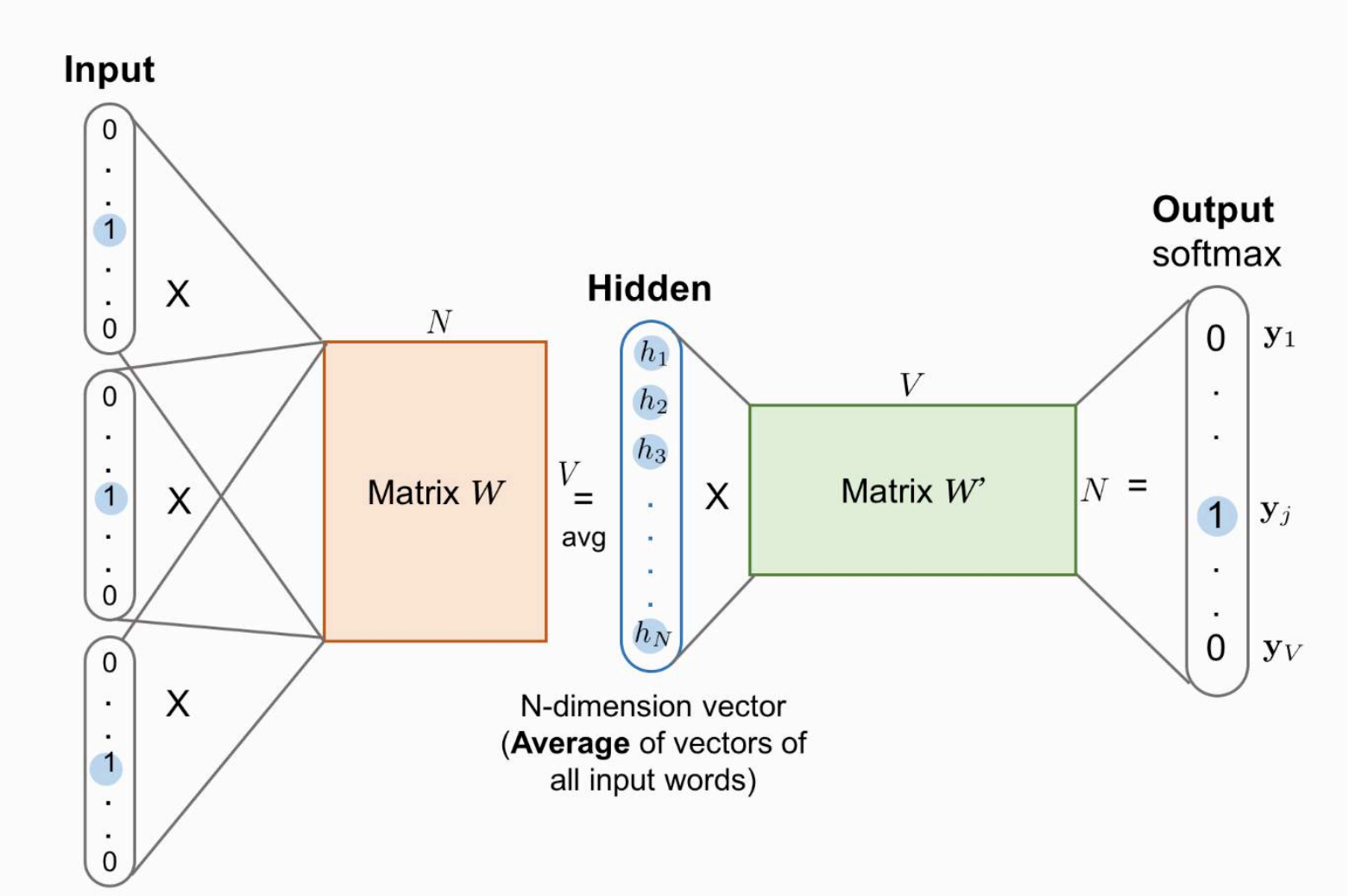
- The softmax function maps arbitrary values x_i to a probability distribution p_i
  - "max" because amplifies probability of largest x_i
  - "soft" because still assigns some probability to smaller x_i
  - Frequently used in Deep Learning

Word2Vec (CBOW)

Sentence: the bulk of linguistic questions concern the dis- tinction between a and m. a linguistic account of phenomenon …

of	the bulk _____ linguistic questions
linguistic	bulk of _____ questions concern
questions	of linguistic _____ concern the
concern	linguistic questions _____ the dis-
the	questions concern _____ dis- tinction
dis-	concern the _____ tinction between
tinction	the dis- _____ between a
between	dis- tinction _____ a and
a	tinction between _____ and m.
and	between a _____ m. a
m.	a and _____ a linguistic
a	and m. _____ linguistic account
linguistic	m. a _____ account of
account	a linguistic _____ of a
of	linguistic account _____ a phenomenon
a	account of _____ phenomenon gen-
phenomenon	of a _____ gen- erally

Word2Vec (CBOW)



(Image from https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html)

Word2Vec (CBOW)

Sentence: language users never choose words randomly , and language is essentially non-random .

In-/Outputs:

```
[(['language', 'users', 'choose', 'words'], 'never'),
(['users', 'never', 'words', 'randomly'], 'choose'),
(['never', 'choose', 'randomly', ','], 'words'),
(['choose', 'words', ',', 'and'], 'randomly'),
(['words', 'randomly', 'and', 'language'], ','),
(['randomly', ',', 'language', 'is'], 'and'),
([',', 'and', 'is', 'essentially'], 'language'),
(['and', 'language', 'essentially', 'non-random'], 'is'),
(['language', 'is', 'non-random', '.'], 'essentially')]
```

Word2Vec (Skipgram)

Sentence: `language users never choose words randomly , and language is essentially non-random .`

Windows:

```
['language', 'users', 'never', 'choose', 'words']

('never', 'language', 1),
('never', 'users', 1),
('never', 'choose', 1),
('never', 'words', 1),
('never', ',', 0),
('never', 'non-random', 0),
('never', 'is', 0),
('never', 'is', 0)
```

aka. negative sampling

Intrinsic Evaluation of Embeddings

- **Relatedness:** Measures correlations between embedding cosine similarity and human evaluation of similarity
- **Analogy:** "a is to b, as x is to ____"
- **Categorization:** Measure purity of clusters based on embeddings
- **Selectional Preference**: "tall" vs "high" man/building

Extrinsic Evaluation of Embeddings

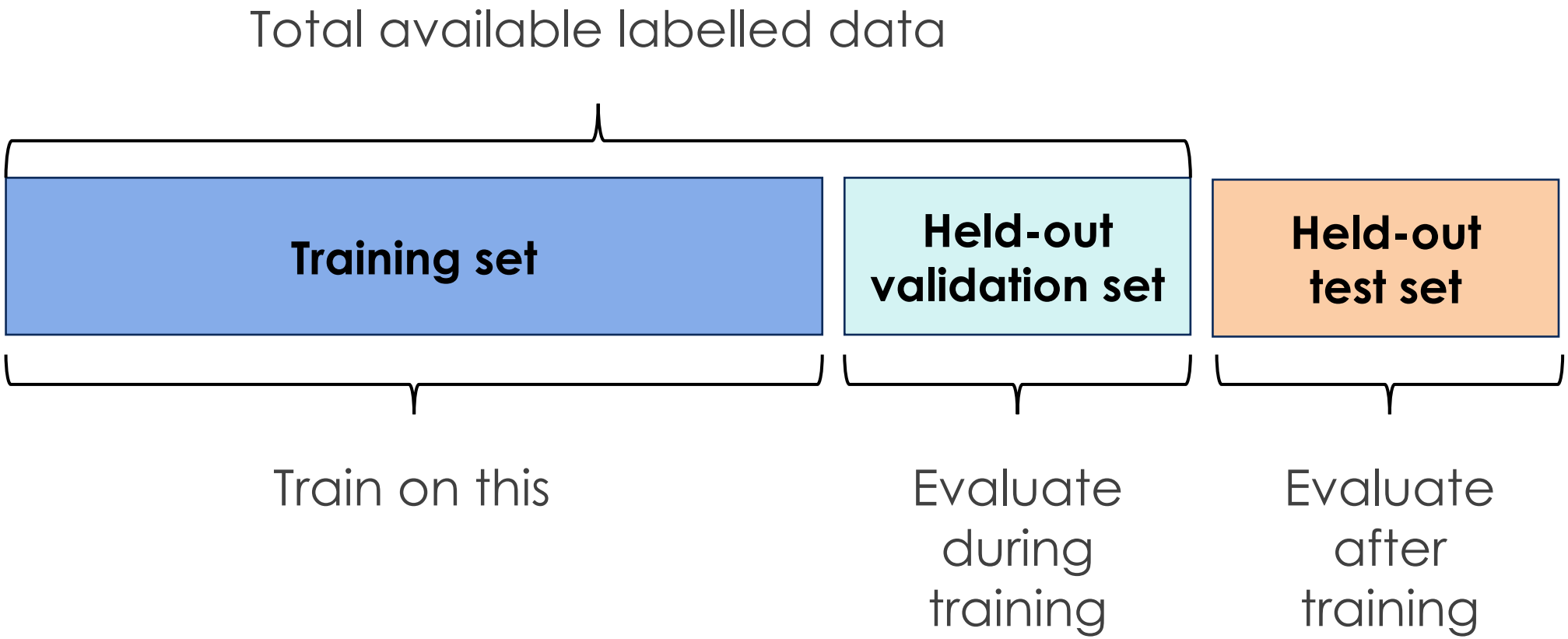
- Load the pretrained embeddings
- Embed the input words as use the embeddings as input to models
- Evaluate which pre-trained embeddings are better for task X

Limitations

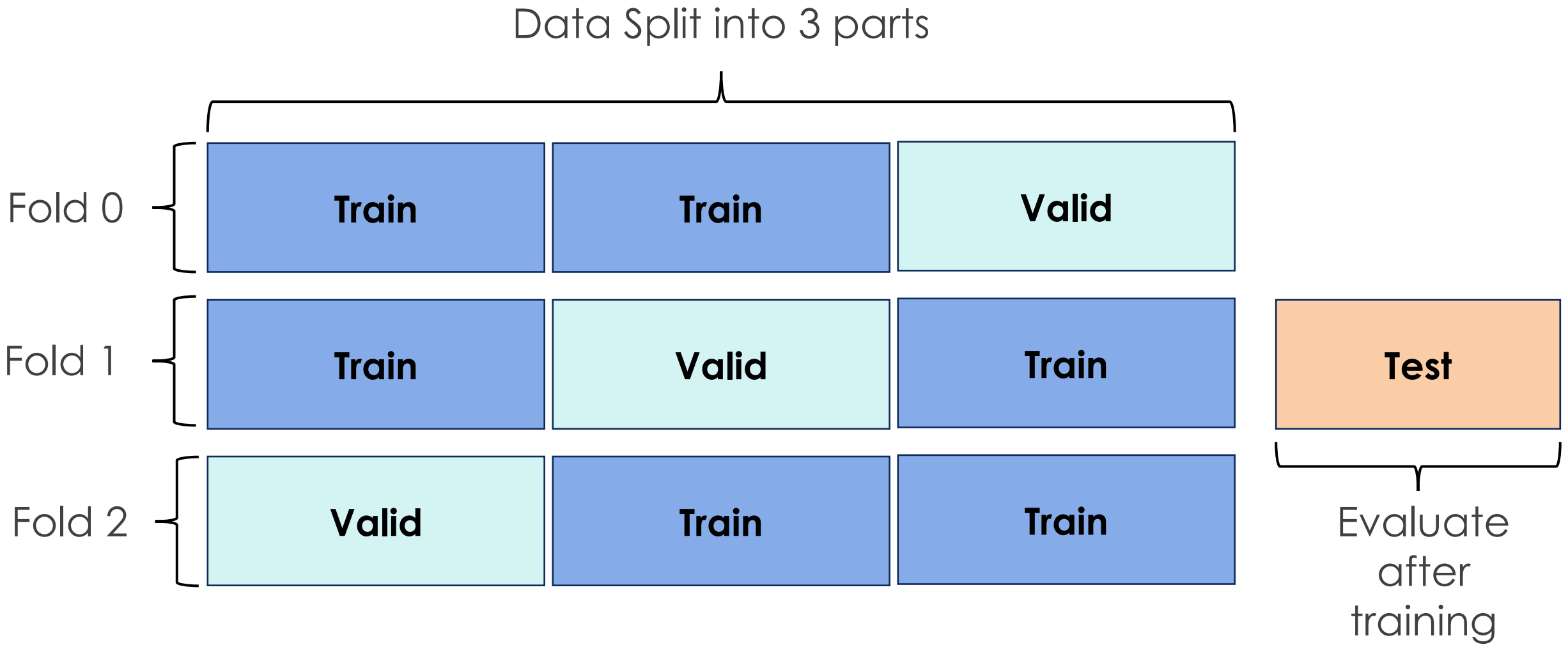
- **Sensitive to "tokens"** (cat vs cats)
- **Insensitive to polysemy** (Industrial *plant* vs "*I'm Groot*")
- **Inconsistent across space**, embeddings for the same words trained with different data are different
- **Can encode bias** (stereotypical gender roles, racial bias)
- **Not interpretable**

Lesson 4: Nuts and Bolts

Hold-Out Evaluation



K-Fold Cross Validation Evaluation



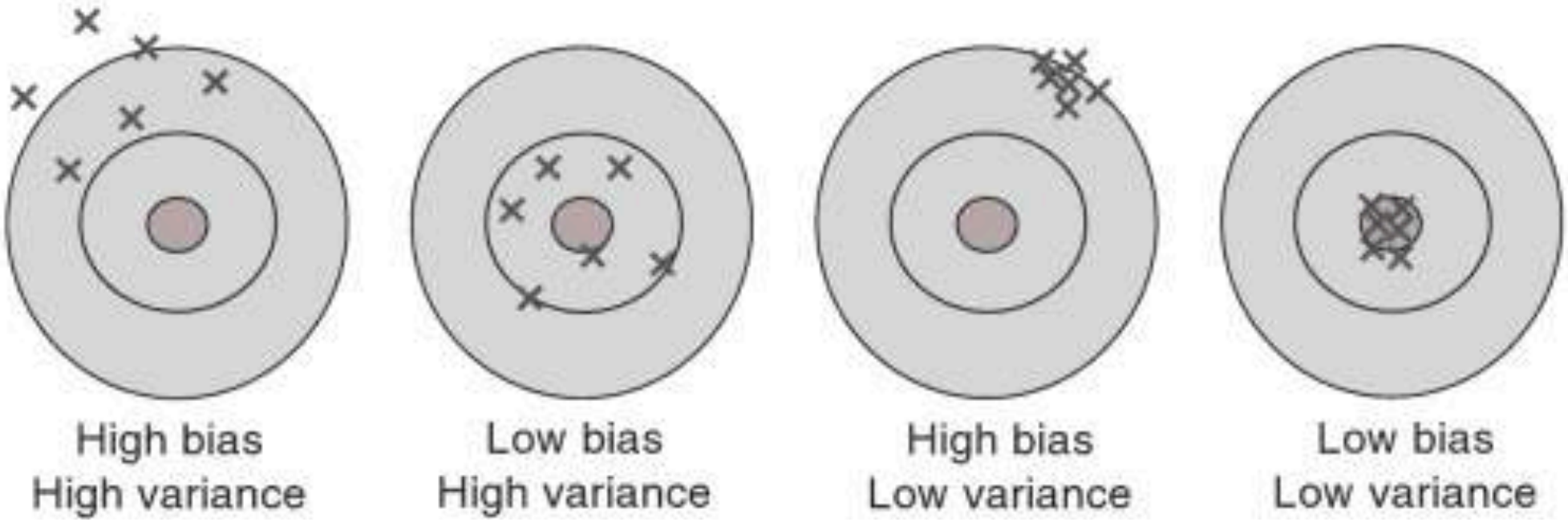
Bias-Variance Tadeoff

- "*A **small network**, with say one hidden unit **is likely to be biased**, since the repertoire of available functions spanned by $f(x, w)$ over allowable weights will in this case be quite limited.*"

- "*if we **overparameterize**, via a large number of hidden units and associated weights, then bias will be reduced (… **with enough weights and hidden units, the network will interpolate the data**) but there is then the **danger of significant variance** contribution to the mean-square error*"

(German et al. 1992)

Bias-Variance Tadeoff



(Moore and McCabe, 2009)

Optimization and Generalization

- **Optimization:** *process of adjusting a model to get the best performance possible on the training data*

- **Generalization:** *how well trained model performs on data it has never seen before*

Over-/Underfitting

- **Underfit:** when model is *not optimized*

- **Overfit:** when model *fails to generalize*

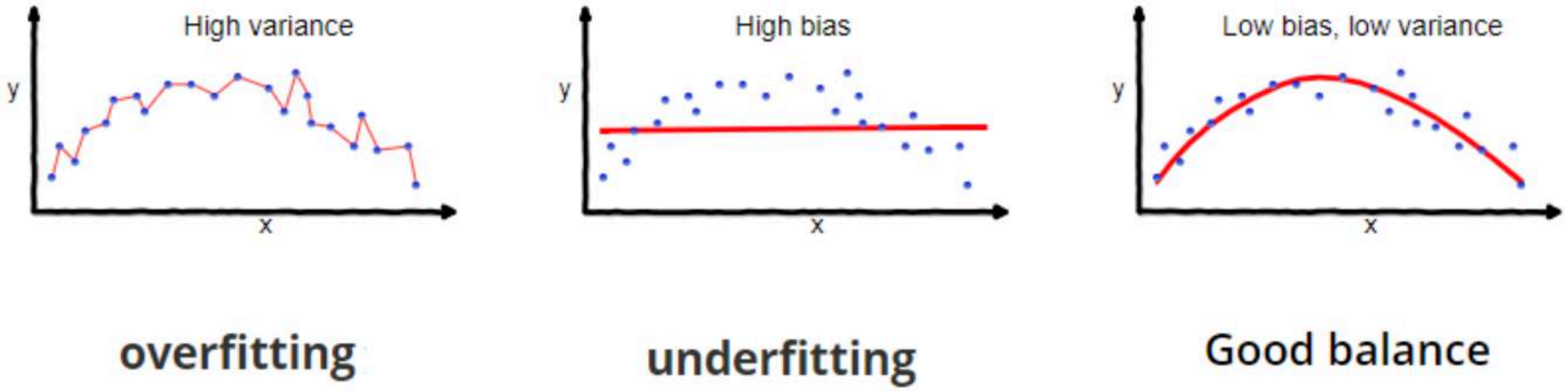


Image from https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

Last Layer Activation and Loss Function

For labels $y_i \in \{0,1\}$the likelihood of some binary data under the Bernoulli model with parameters θ is

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p(y_i = 1|\theta)^{y_i} p(y_i = 0|\theta)^{1-y_i}$$

NLLoss = $$\ln \mathcal{L}(\theta) = -\sum_{i=1}^{n} y_i \ln(\hat{y})$$

CE = $$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln(\hat{y})$$

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification	Sigmoid / Softmax	Categorical Cross Entropy	torch.nn.CrossEntropyLoss
	LogSoftmax	Negative Log Loss	torch.nn.NLLLoss
Multi-class, multi-label classification	Sigmoid / Softmax	Binary Cross Entropy	torch.nn.BCELoss
Regression to arbitrary value	None	L2 Loss	torch.nn.MSELoss
Regression (0, 1)	Sigmoid	L2 Loss	torch.nn.MSELoss

Gradient Descents

Stochastic Gradient Descent (SGD)
torch.optim.SGD

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

Mini-batch SGD
torch.optim.SGD

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

SGD with momentum
torch.optim.SGD

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Gradient Descents

```
def step(self, closure=None):
    """Performs a single optimization step.

    Arguments:
        closure (callable, optional): A closure that reevaluates the model
            and returns the loss.
    """
    loss = None
    if closure is not None:
        loss = closure()

    for group in self.param_groups:
        weight_decay = group['weight_decay']
        momentum = group['momentum']
        dampening = group['dampening']
        nesterov = group['nesterov']
```

Gradient Descents

```
        for p in group['params']:
            if p.grad is None:
                continue
            d_p = p.grad.data
            if weight_decay != 0:
                d_p.add_(weight_decay, p.data)
            if momentum != 0:
                param_state = self.state[p]
                if 'momentum_buffer' not in param_state:
                    buf = param_state['momentum_buffer'] = torch.zeros_like(p.data)
                    buf.mul_(momentum).add_(d_p)
                else:
                    buf = param_state['momentum_buffer']
                    buf.mul_(momentum).add_(1 - dampening, d_p)
                if nesterov:
                    d_p = d_p.add(momentum, buf)
                else:
                    d_p = buf

            p.data.add_(-group['lr'], d_p)

    return loss
```

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

Adaptive Moment Estimation (Kingma, 2015)

- "*momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface*" – (Dozat, 2016)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

SGD vs Adam

```
# Vanilla SGD
x += - learning_rate * dx
```

x is a vector of parameters and
dx is the gradient

```
# Adam
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m /
        (np.sqrt(v) + eps)
```

x is a vector of parameters and
dx is the gradient
m is the smoothen gradient
v is the 'cache' used to normalize **x**
eps is smoothing term (1e-4 to 1e-8)
beta1,beta2 are hypers (0.9, 0.999)

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from CLR_preview import CyclicLR
from adamW import AdamW

# Step 1: Initialization.
model = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.Sigmoid(),
            nn.Linear(hidden_dim, output_dim),
            nn.Sigmoid()
        )

optimizer = AdamW(model.parameters(), lr=0.001, betas=(0.9, 0.99), weight_decay = 0.1)
# For classification.
##clr_stepsize = (num_classes*50//int(batchsz))*4
clr_stepsize = 3e-4
clr_wrapper = CyclicLR(optimizer, step_size=clr_stepsize)

# Initialize the loss function.
criterion = nn.MSELoss()

losses = [] # Keeps track of the loses.
# Step 2-4 of training routine.
for _e in tqdm(range(num_epochs)):
    # Reset the gradient after every epoch.
    optimizer.zero_grad()
    # Step 2: Foward Propagation
    predictions = model(X)
    # Step 3: Back Propagation
    # Calculate the cost between the predictions and the truth.
    loss = criterion(predictions, Y)
    # Remember to back propagate the loss you've computed above.
    loss.backward()
    # Step 4: Optimizer take a step and update the weights.
    optimizer.step()
    losses.append(loss.data.item())
```

Super convergence, see https://www.fast.ai/2018/07/02/adam-weight-decay/

CLR_preview from https://github.com/ahirner/pytorch-retraining/blob/master/CLR_preview.py

AdamW from https://github.com/egg-west/AdamW-pytorch

Lesson 5: Language Models and RNN

Language Model

- Language Modelling is task of ***predicting which word comes next***

- ***Predict next word*** x_i, ***given all context words up till the current word,*** $x_1, \ldots, x_{i-1}$

- Language Model aka. ***assigning probability of text as an accumulated probability of all individual words given their contexts***

Evaluating Language Model

- **Log-likelihood:**

$$LL(\mathcal{E}_{test}) = \sum_{E \in \mathcal{E}_{test}} \log P(E)$$

- **Per-word Log Likelihood:**

$$WLL(\mathcal{E}_{test}) = \frac{1}{\sum_{E \in \mathcal{E}_{test}} |E|} \sum_{E \in \mathcal{E}_{test}} \log P(E)$$

- **Per-word (Cross) Entropy:**

$$H(\mathcal{E}_{test}) = \frac{1}{\sum_{E \in \mathcal{E}_{test}} |E|} \sum_{E \in \mathcal{E}_{test}} -\log_2 P(E)$$

- **Perplexity:**

$$ppl(\mathcal{E}_{test}) = 2^{H(\mathcal{E}_{test})} = e^{-WLL(\mathcal{E}_{test})}$$

Perplexity

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Normalized by number of words

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

Inverse probability of test set

Perplexity

- Maximizing probability = Minimizing perplexity

- **What is perplexity in Deep Learning models?**

NLLoss = $\ln \mathcal{L}(\theta) = -\sum_{i=1}^{n} y_i \ln(\hat{y})$

CE = $\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln(\hat{y})$

Lower is better!!!

Generating text with Ngram LM

While not </s>:

calculate probability of possible next words

choose a word from the top N most probable

<context> = "<s> he likes to"

P(drink | <context>) > P(coffee | <context>)

Limitations of Ngram Language Models

- Storage and retrieving ngrams probabilities
- Sparsity and smoothing hacks
- Context windows limits long-distance dependencies
- Frequencies says little about semantics

Language Model Evaluation

- A language model that can ***predict the right words***, i.e. ***assign a higher probability to words that occurs*** is a better model

- Traditionally, LM is evaluated on **perplexity**

- Perplexity is the ***inverse probability of the test set***, normalized by the number of words

RNN vs N-gram Language Model

N-gram model

RNN with increased complexity

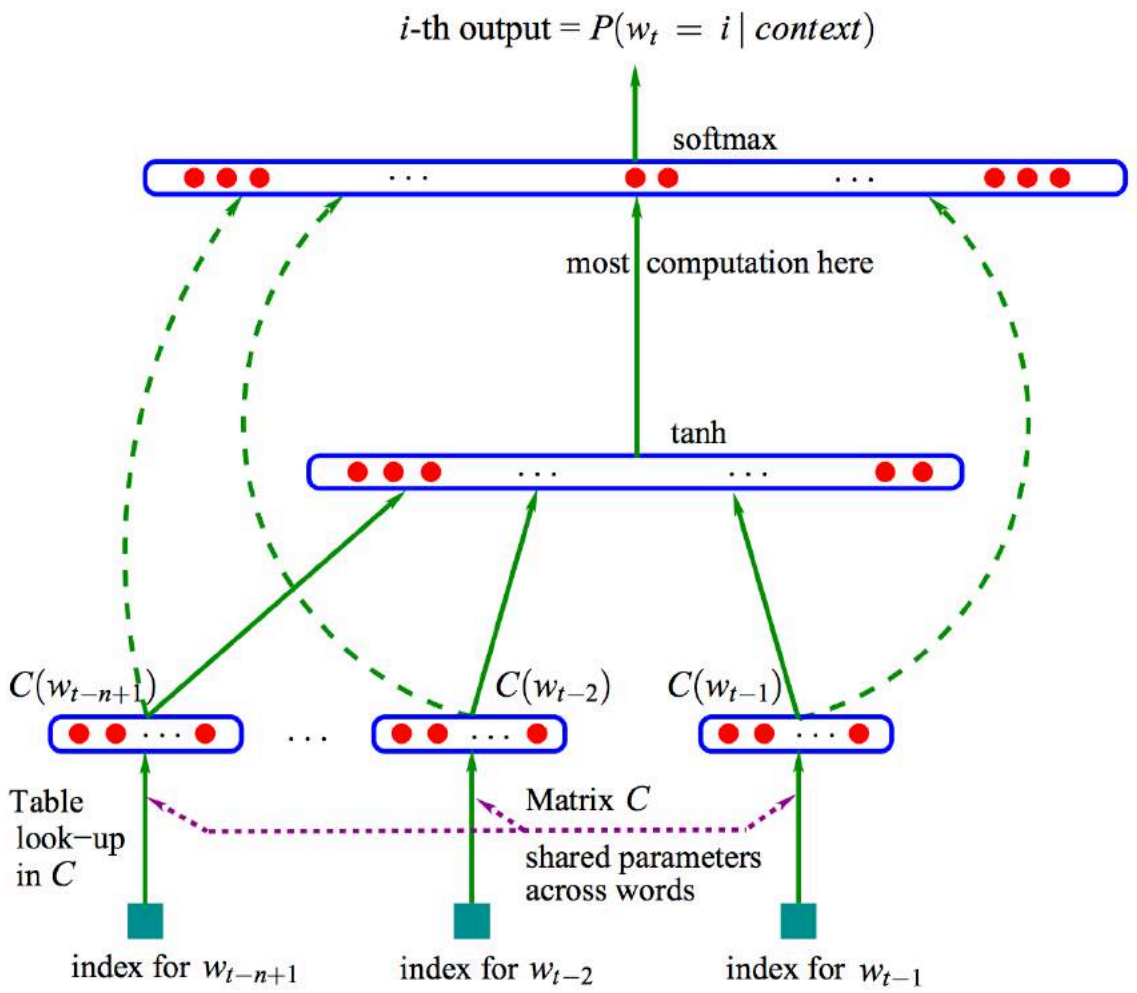
Model	Perplexity
Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)	51.3
RNN-2048 + BlackOut sampling (Ji et al., 2015)	68.3
Sparse Non-negative Matrix factorization (Shazeer et al., 2015)	52.9
LSTM-2048 (Jozefowicz et al., 2016)	43.7
2-layer LSTM-8192 (Jozefowicz et al., 2016)	30
Ours small (LSTM-2048)	43.9
Ours large (2-layer LSTM-2048)	39.8

From https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/

Deep Language Models

- Calculate weights/features of context
- Based on the weights, compute probabilities
- Optimize weights using gradient descent to minimize errors on probabilities computation

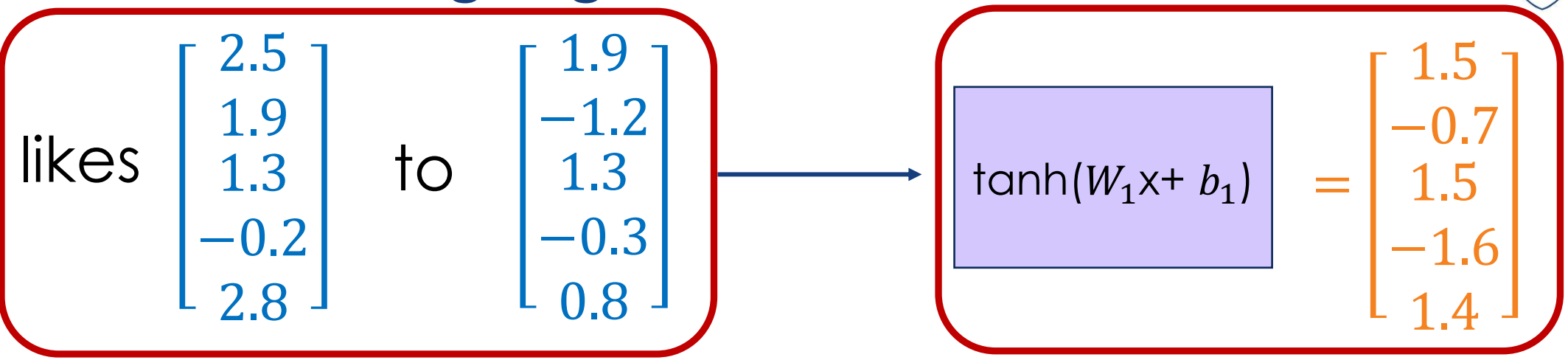
A Neural Probabilistic Language Model



(Bengio et al. 2004)

Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

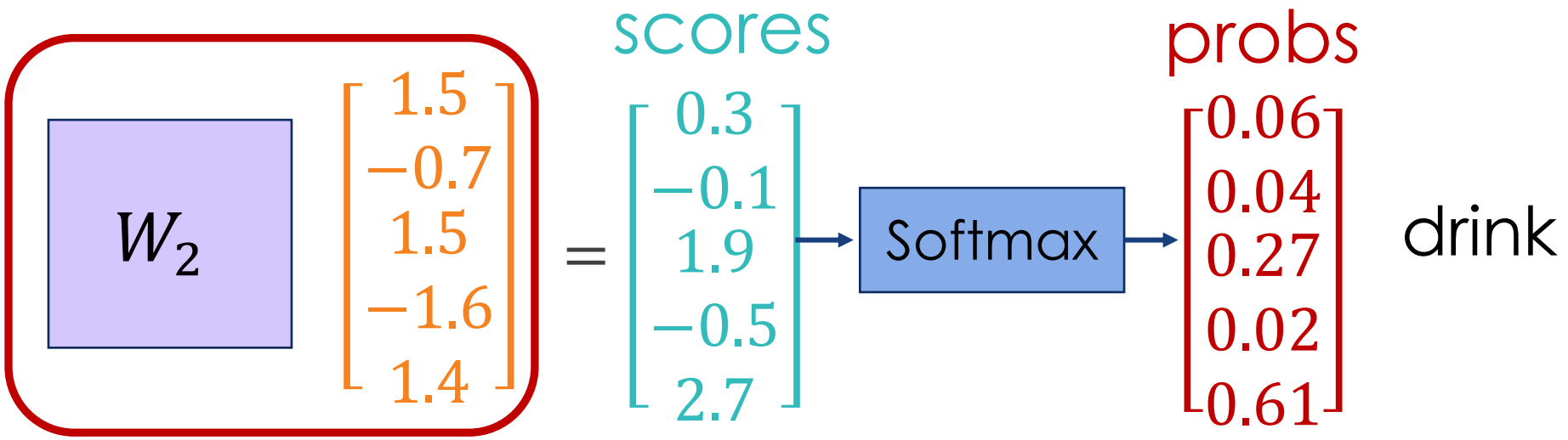
Neural Language Model



Lookup function

Transform

Predict



Tricks to Prevent Overfitting

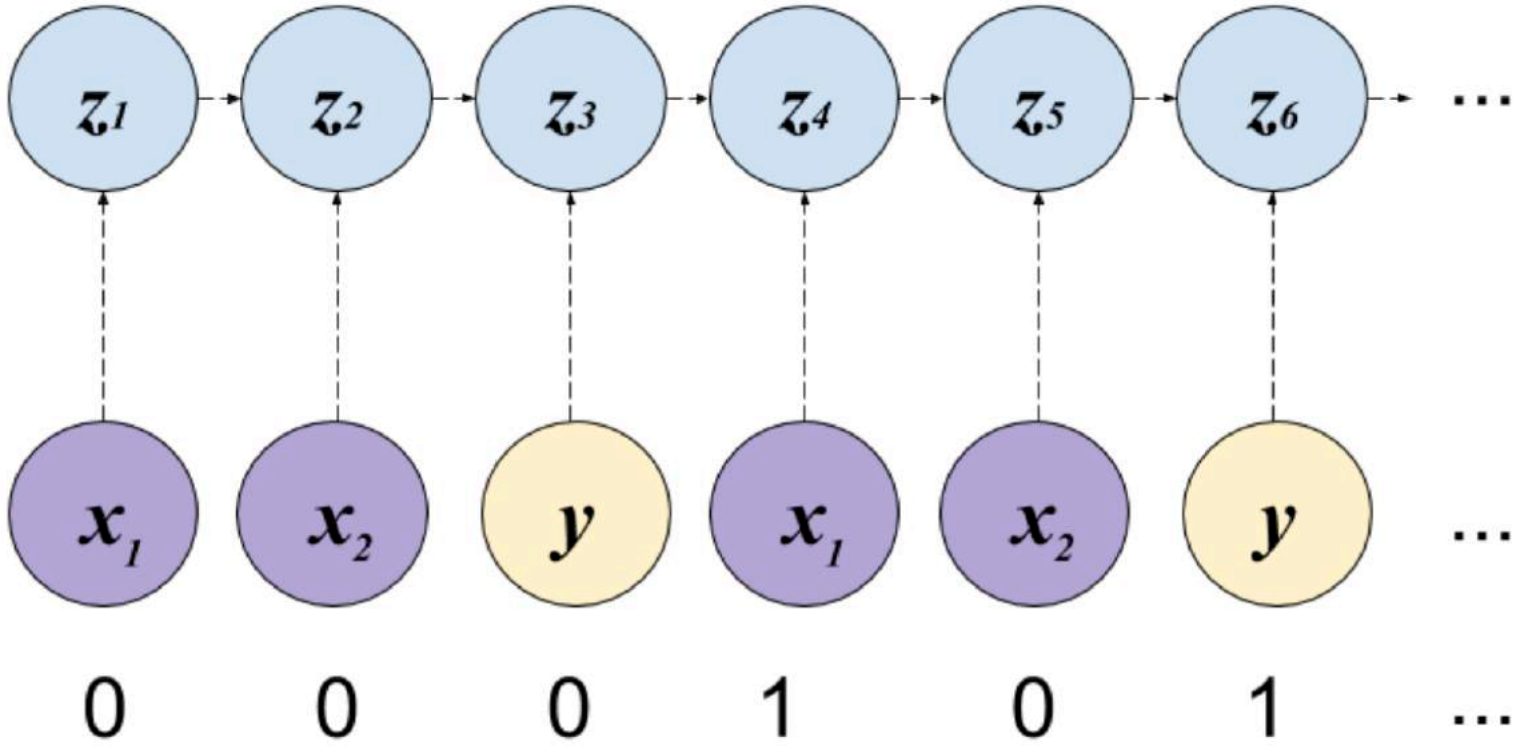
- **Shuffle the input sentences**
  - Imagine seeing "*Justin Bieber is the best singer, baby, baby, baby, oh*" 100x at the start of the corpus…
- **Early stopping** based on some validation set
- **Dropout** during training
- **Mini-batching** makes training much faster

(*Almost*) Everything is a Sequence in Language

- Natural language is full of sequential data
- Word == sequence of characters
- Sentence == sequence of words
- Dialog/Discourse == sequence of sentences

Recurrent Neural Net (RNN)

(Input + **Prev_Hidden**) -> Hidden -> Output



(Elman, 1990)

Recurrent Neural Net (RNN)

(Input + Empty_Hidden) -> Hidden -> Output

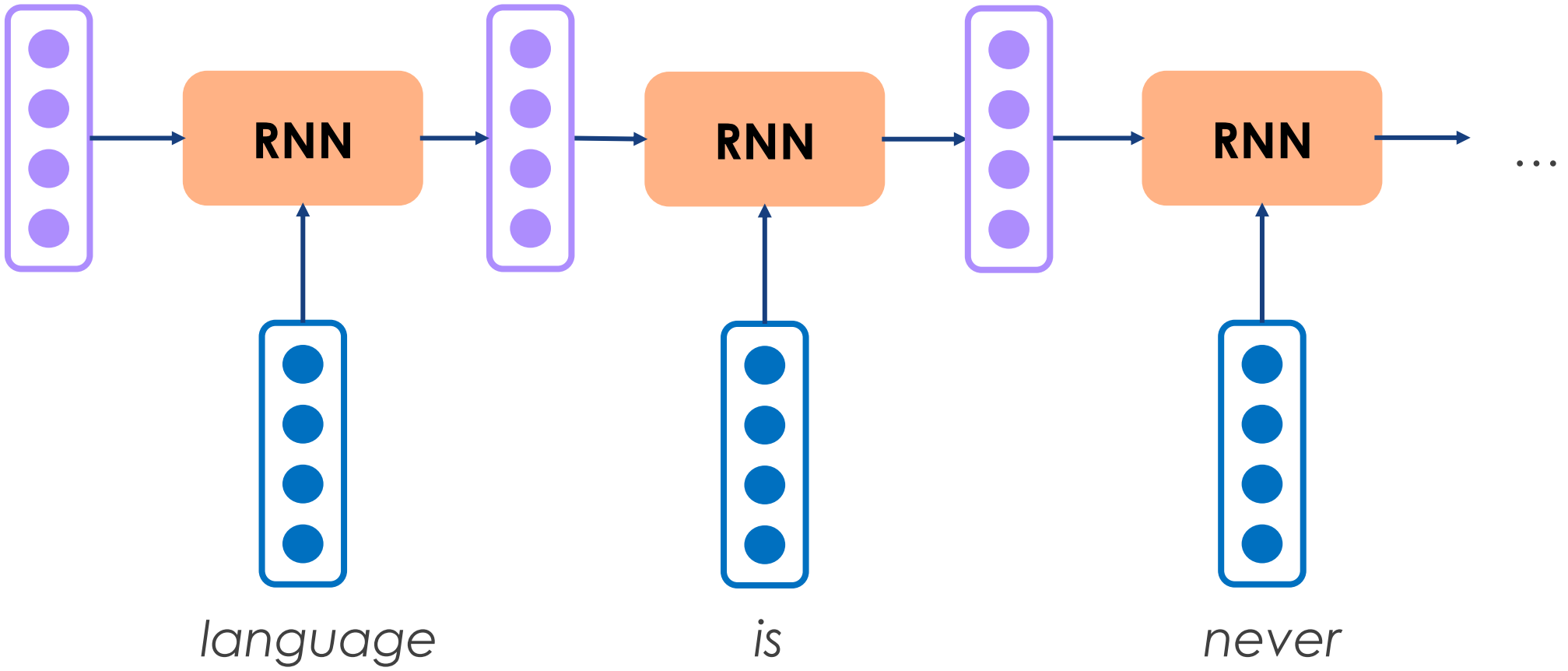
(Input + Prev_Hidden) -> Hidden -> Output

(Input + Prev_Hidden) -> Hidden -> Output

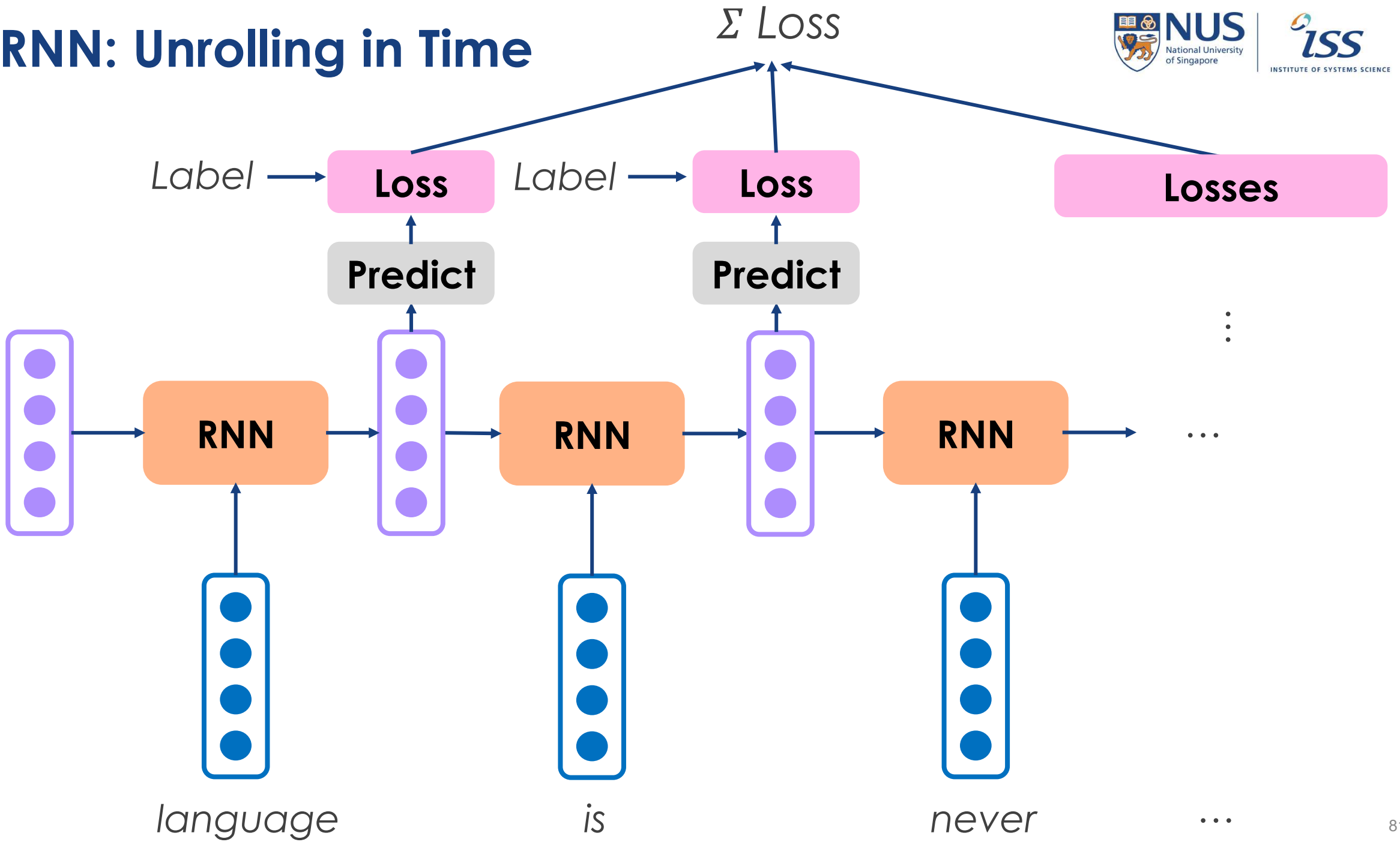
(Input + Prev_Hidden) -> Hidden -> Output

(Source: A really good blogpost on RNN by Trask: https://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/)

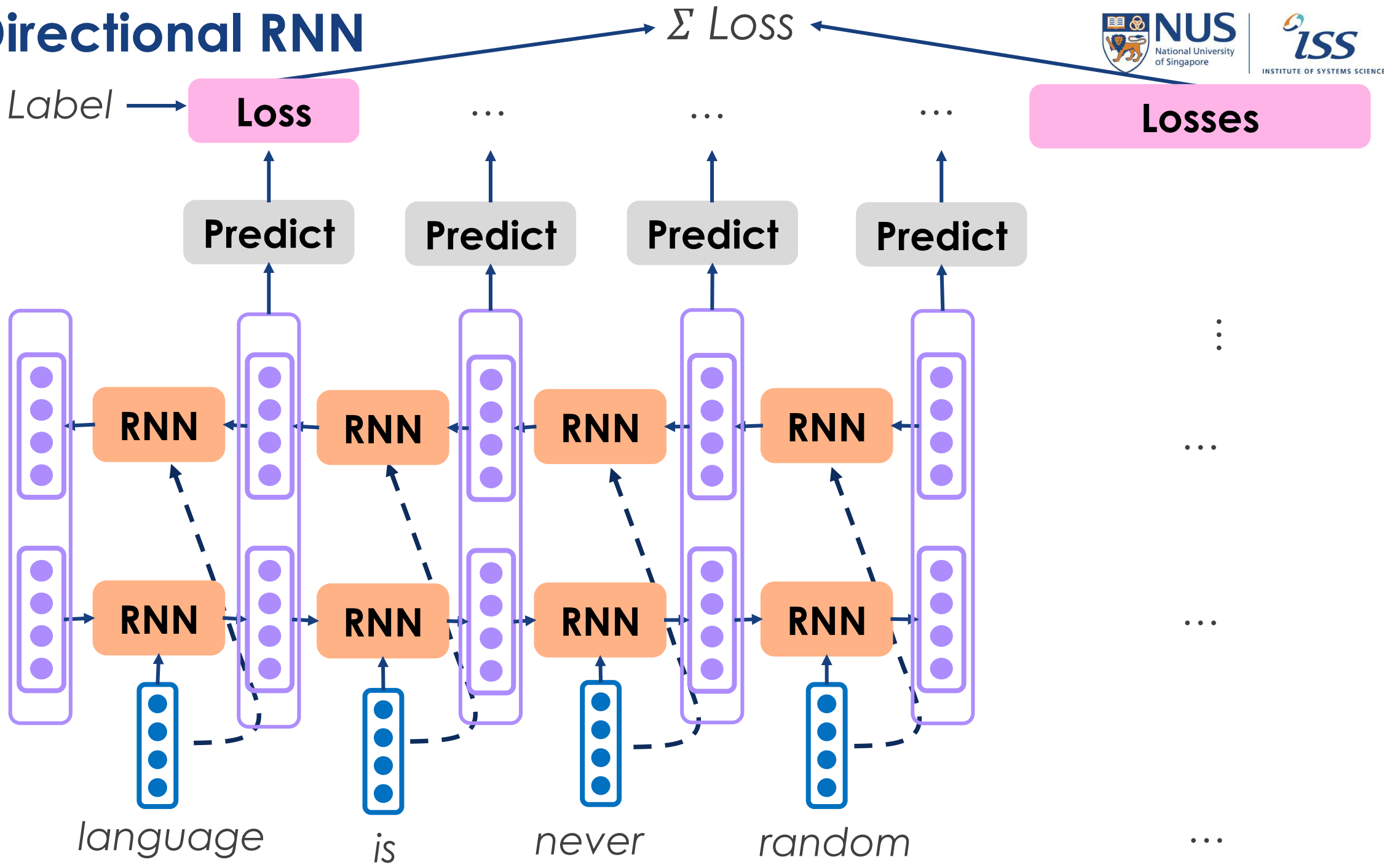
RNN: Unrolling in Time



RNN: Unrolling in Time



Bi-Directional RNN



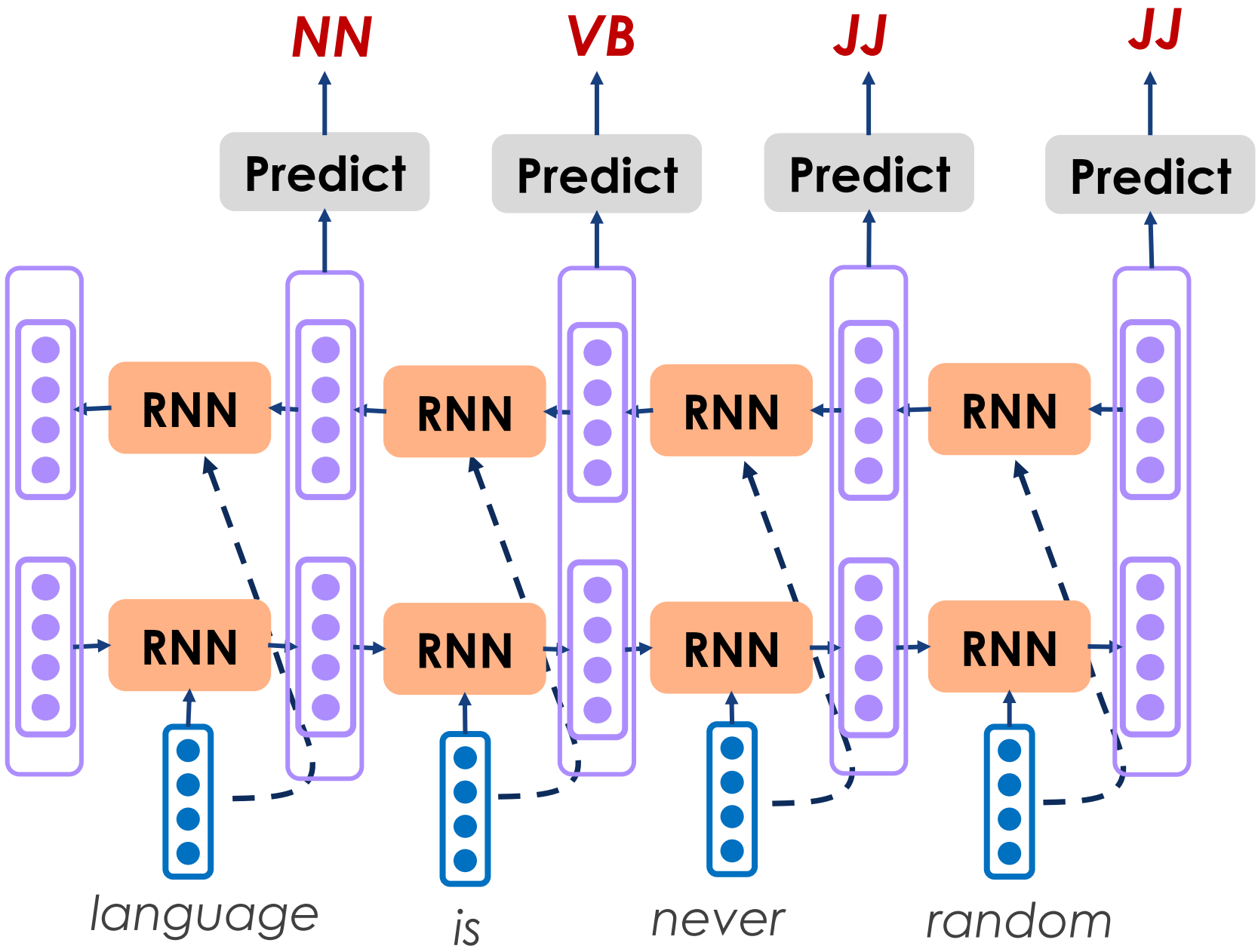
Sequence Predictions

- **Token Tagging**
  - RNN generates a prediction per token, so we can do most NLP annotations where each token comes with their respective tag, e.g. POS, NER, etc.

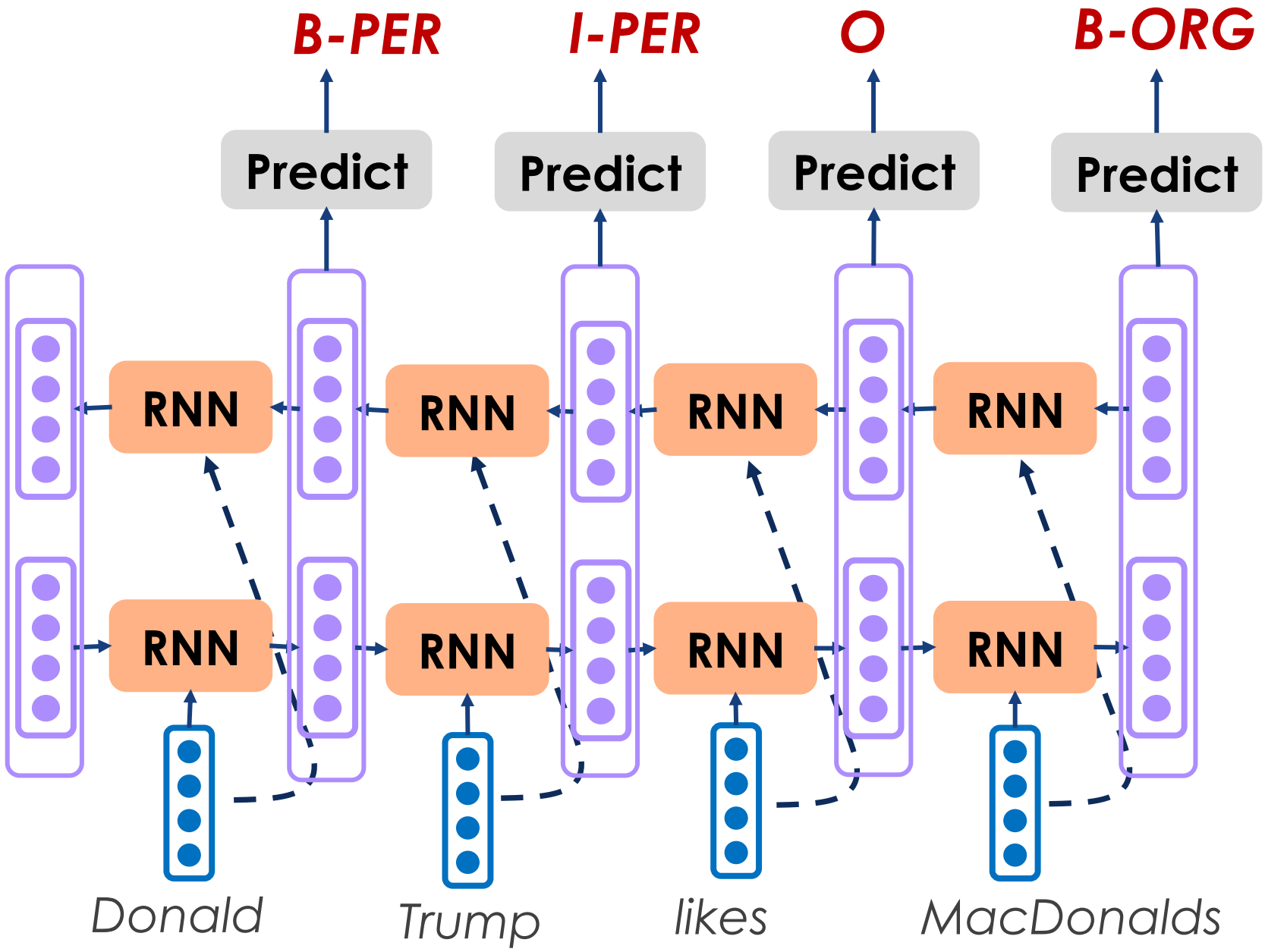
- **Sentence/Text Classification**
  - Put the end or aggregate of the hidden layer(s) under Softmax to produce probabilistic classifier

- **Sequence Generation**
  - Using the previous hidden layer as input to predict the next most probable word

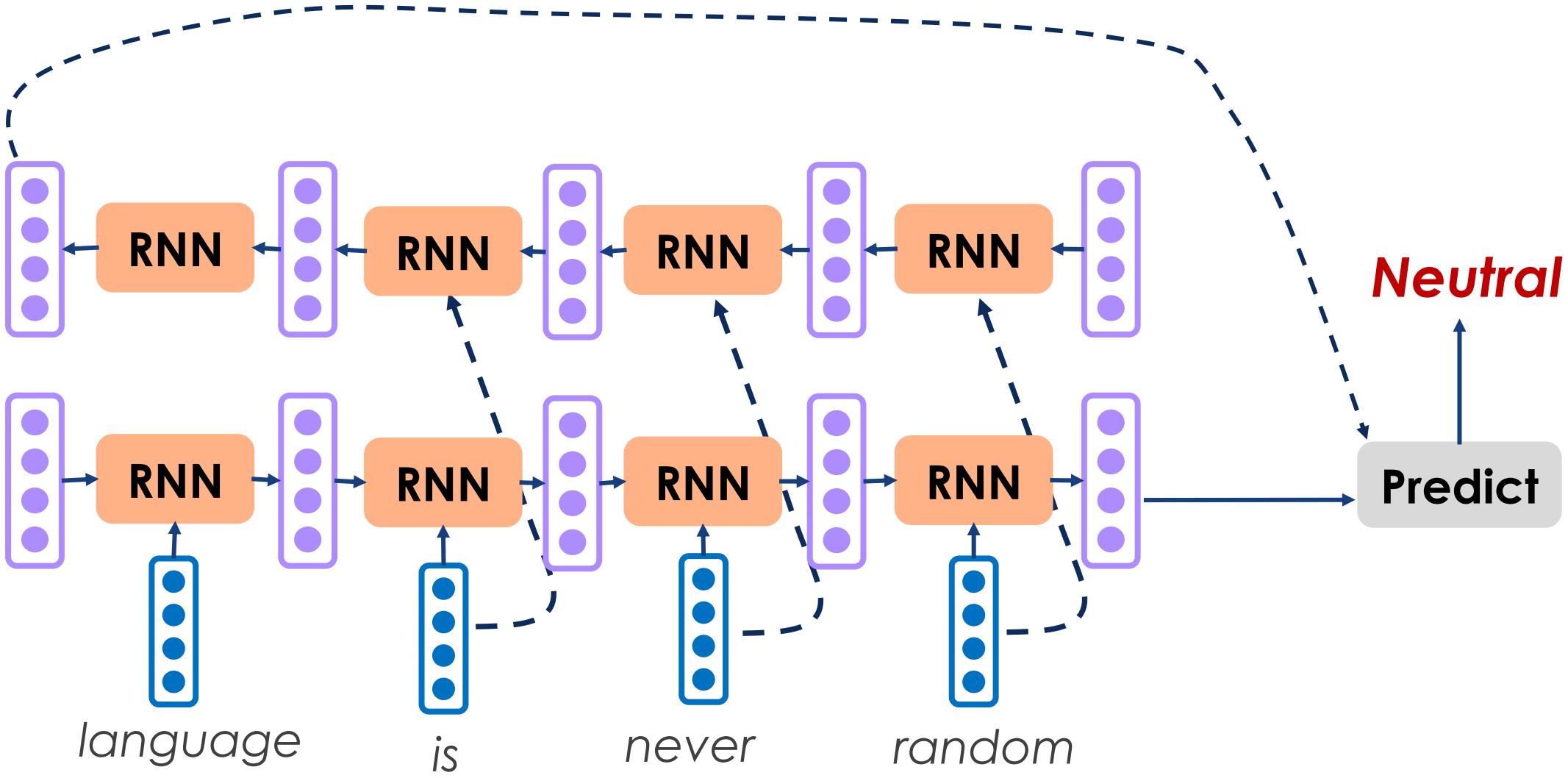
Sequence Tagging



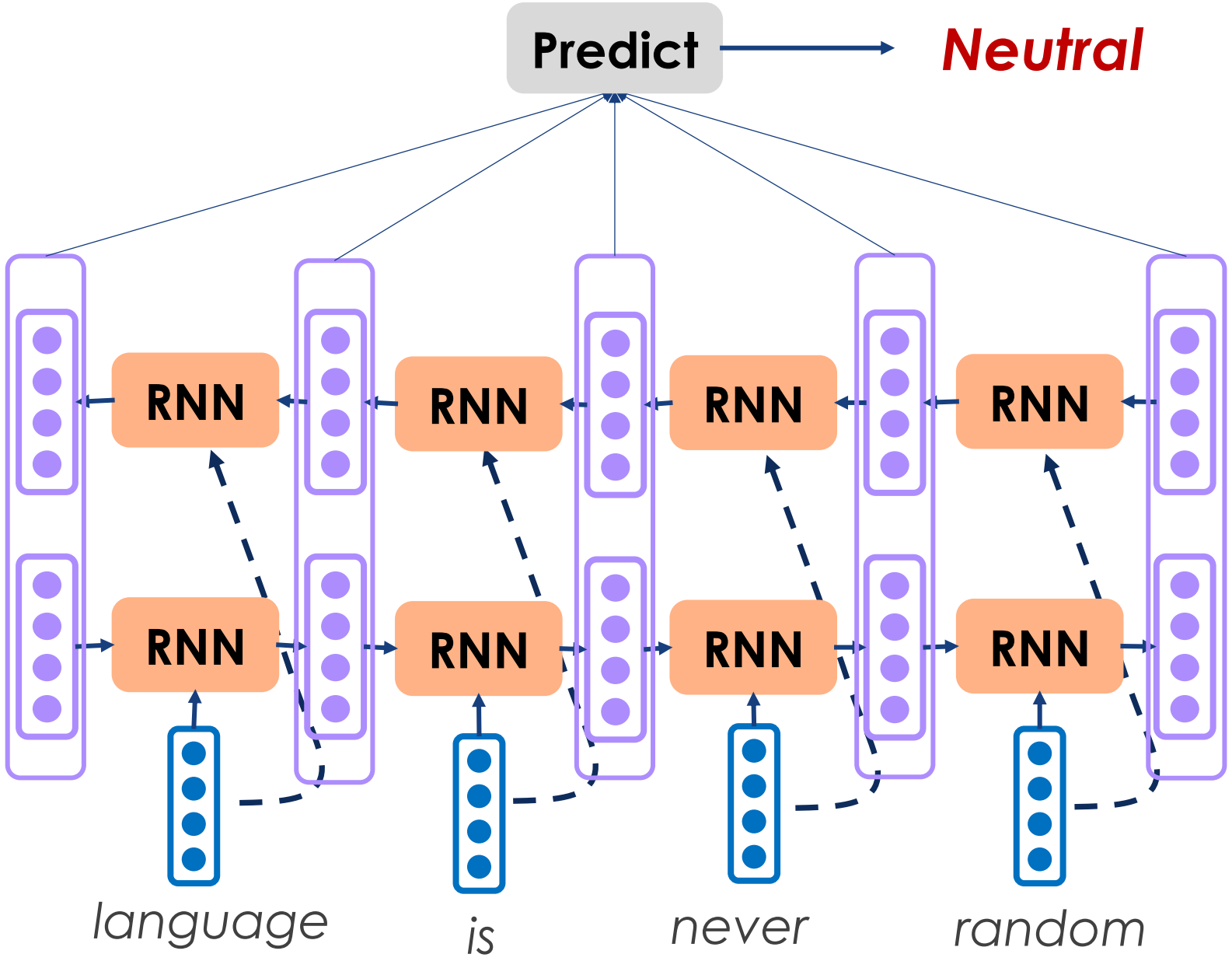
Sequence Tagging



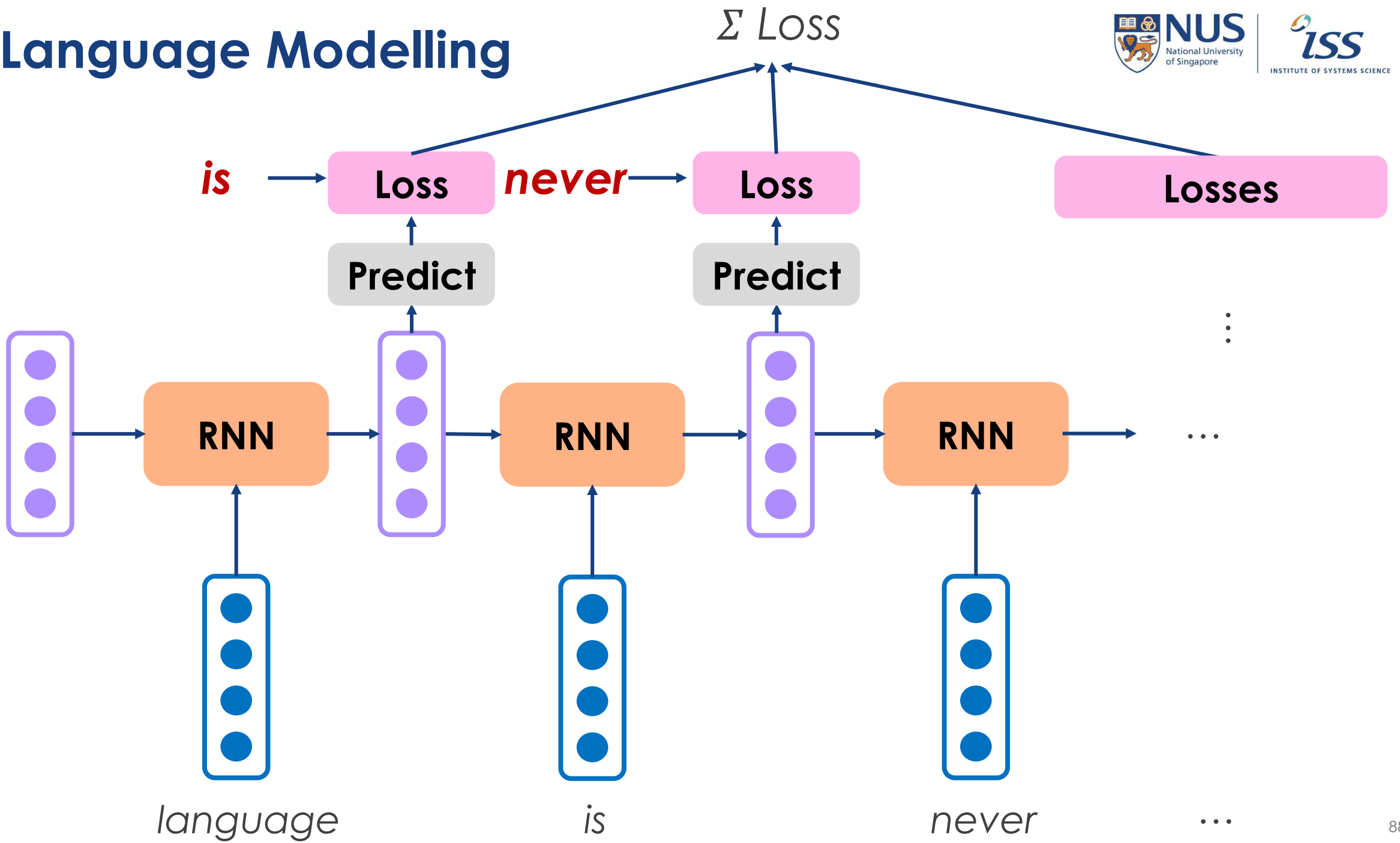
Text Classification



Text Classification



Language Modelling



Lesson 6: Memory Networks and Conditional Generation (Seq2Seq)

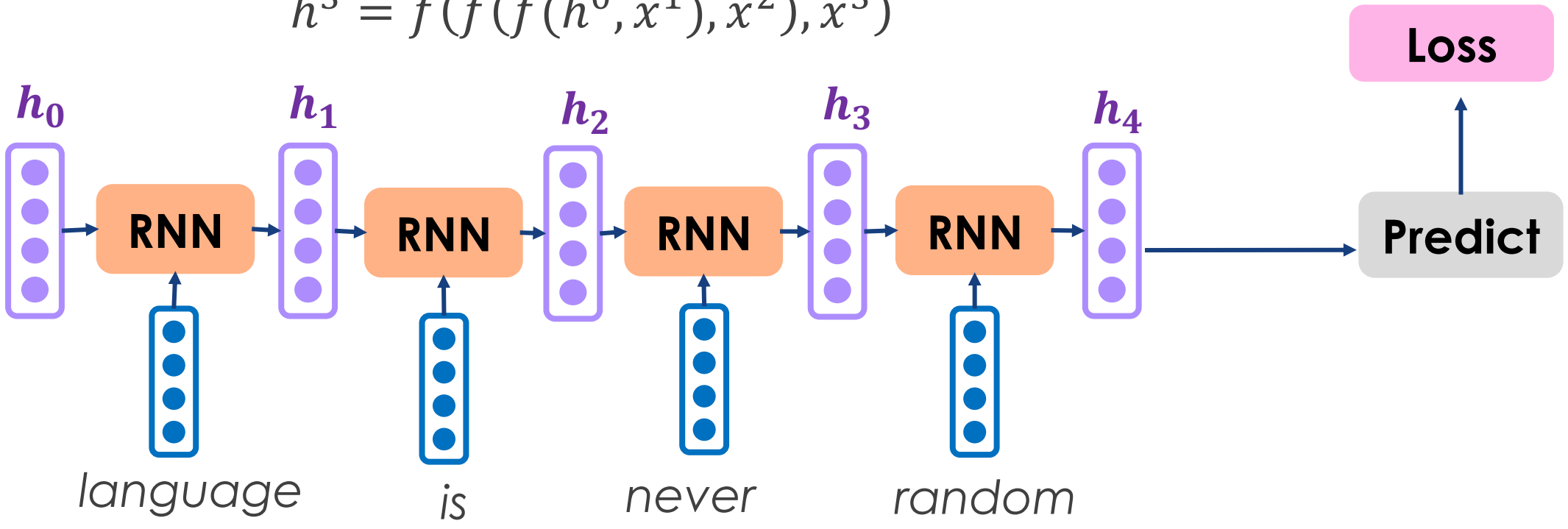
RNN Exploding gradient

Lets look at the **"curled up" RNN:**

$$h^t = f(h^{t-1}, x^t)$$

If we **unroll the RNN** that has 3 time-step, we get:

$$h^3 = f(f(f(h^0, x^1), x^2), x^3)$$



Exploding gradient

Lets look at the **"curled up" RNN:**

$$h^t = f(h^{t-1}, x^t)$$

If we **unroll the RNN** that has 3 time-step, we get:

$$h^3 = f(f(f(h^0, x^1), x^2), x^3)$$

And if consider $f(x)$ as a "harmless-looking"

$$f(x) = 3.5x(1-x)$$

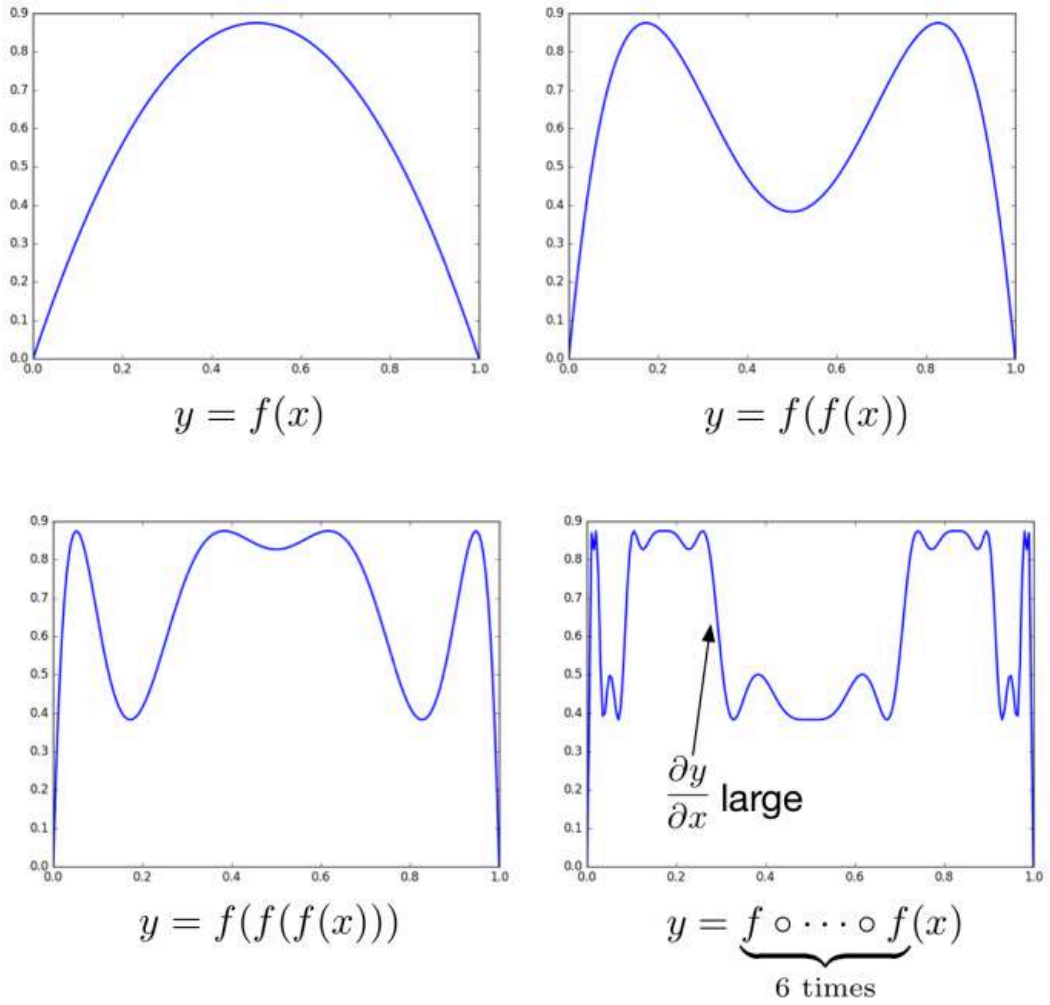


Figure and example from (Grosse 2017)

Solution: Gradient Clipping

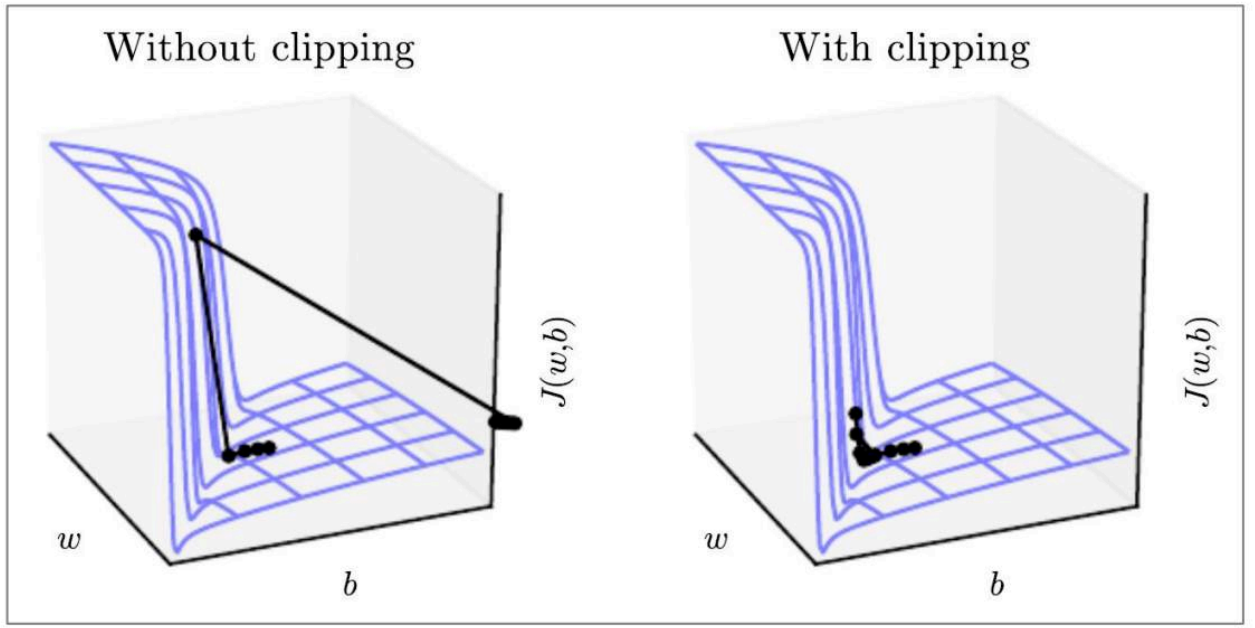
- **General idea:** if we don't like the big numbers, just put a max to the gradient values

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

Source: (Pascanu et al, 2013)

Solution: Gradient Clipping



- Figure above shows a loss surface of an RNN
- **(Left) without clipping**, weights values increases and so does gradient causing loss to "jump off the clip"
- **(Right) with clipping**, weights are kept to a max and loss doesn't inflate

Source: (Goodfellow et al. 2016)

Partial Gradients

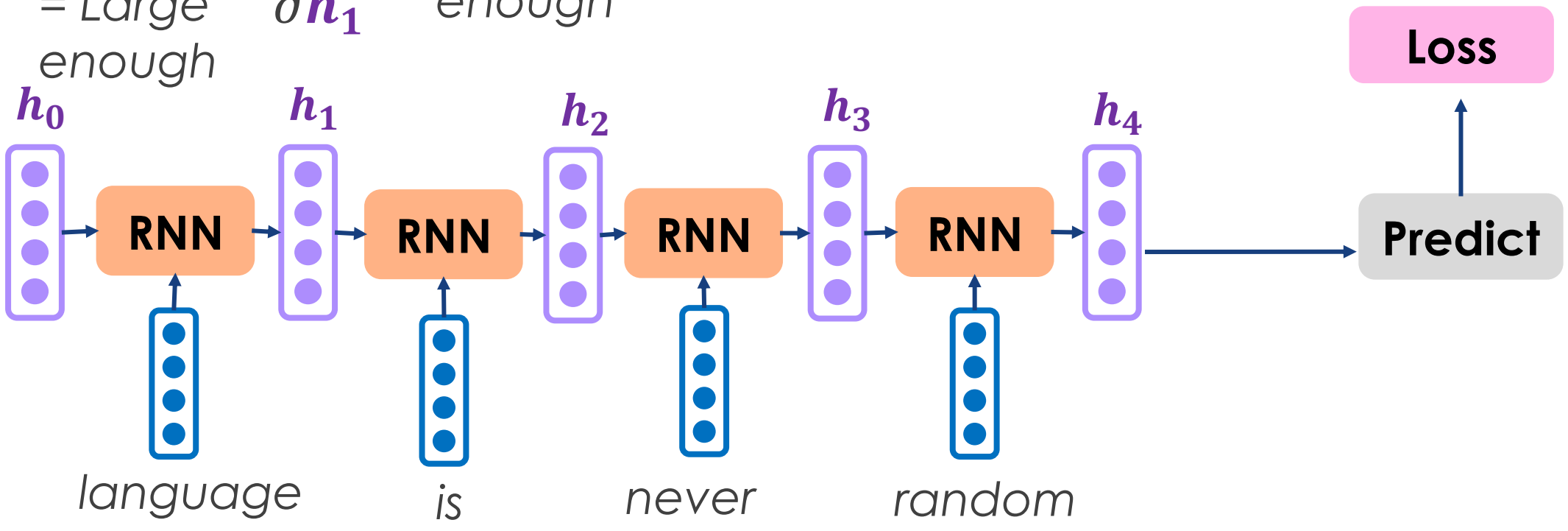
$\frac{\partial(\boldsymbol{loss})}{\partial \boldsymbol{h_4}}$ = Large enough

$\frac{\partial(\boldsymbol{h_4})}{\partial \boldsymbol{h_3}}$ = Large enough

$\frac{\partial(\boldsymbol{h_3})}{\partial \boldsymbol{h_2}}$ = Large enough

$\frac{\partial(\boldsymbol{h_2})}{\partial \boldsymbol{h_1}}$ = Large enough

$\frac{\partial(\boldsymbol{h_1})}{\partial \boldsymbol{h_0}}$ = Large enough



RNN Vanishing Gradients

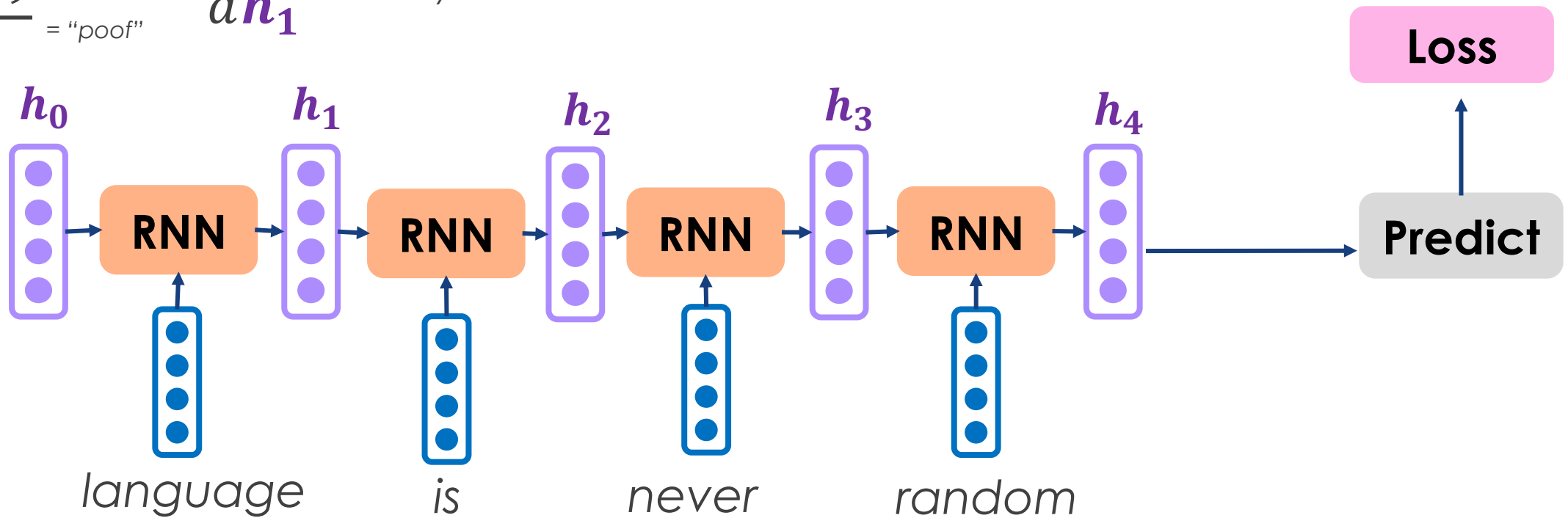
$\frac{d(loss)}{dh_4}$ = Large enough

$\frac{d(loss)}{dh_3}$ = Okay

$\frac{d(loss)}{dh_2}$ = small

$\frac{d(loss)}{dh_1}$ = tiny

$\frac{d(loss)}{dh_0}$ = "poof"



Why is vanishing gradient a problem?

- When moving along the timestep of RNN, gradient can be thought as "***the effects of the past on the future" (See, 2019)***

- Vanishing gradient means info from the ***previous words are less influential to predict words in the future***

- When gradients are small over longer distance, it's hard to know whether
  - **Trump** has no relation to the **cheezburger** or
  - the model learns to **wrong parameter** that doesn't capture relation between **Trump** and **cheezburger**

Solution: Information Controlling RNN

- **General idea:** make additive connections between time steps
- ***Keeping a memory of past time steps and add them to the current one***
- ***Addition don't change gradient***, no vanishing
- ***Gates control different information strengths*** from the past

Long Short Term Memory (LSTM)

- At each time step, perform the following operations

Input: controls how much new cell info is written to cell

$$i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$$

Forget: controls how much previous cell info should be forgotten

$$f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$$

Output: controls how much info is written to hidden state

$$o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$$

New cell info: new content to write to cell

$$\tilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$$

Cell state (Memory): forget some of the previous cell info and write some new cell info

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

Hidden state: output some bits of info from the cell state for next cell to "remember"

$$h_t = o_t \circ \tanh(c_t)$$

Long Short Term Memory (LSTM)

- At each time step, perform the following operations

Sigmoid to make sure all outputs are (0,1)

$$i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$$

$$f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$$

$$o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$$

$$\tilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$$

Elementwise product

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

Bruce Lee on LSTM

"*Absorb what is useful, discard what is not, add what is uniquely your own.*"

– Bruce Lee

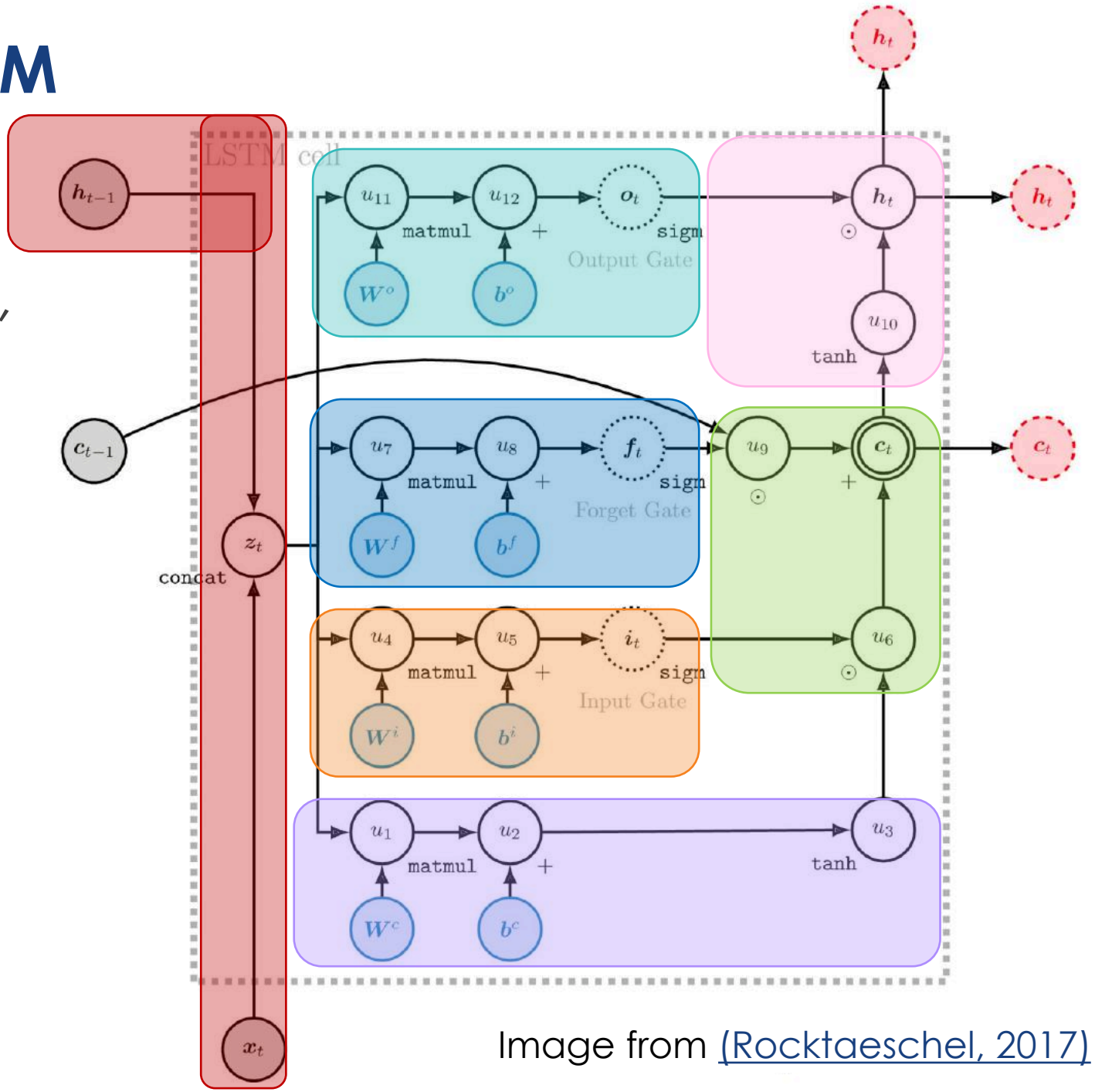


Image from (Rocktaeschel, 2017)

Gated Recurrent Neural Nets

- At each time step, perform the following operations

Update: controls how much info in the new hidden states are kept or updated

Reset: controls how much info in the previous hidden states are kept

New Hidden state: Reset selects info from previous hidden state and combines it with the current input

Hidden state: Update selects info from previous hidden state and combines it with the current input

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{W}_u \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_u \boldsymbol{x}^{(t)} + \boldsymbol{b}_u\right)$$

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{W}_r \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r \boldsymbol{x}^{(t)} + \boldsymbol{b}_r\right)$$

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{W}_h (\boldsymbol{r}^{(t)} \circ \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}_h \boldsymbol{x}^{(t)} + \boldsymbol{b}_h\right)$$

$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{u}^{(t)}) \circ \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \circ \tilde{\boldsymbol{h}}^{(t)}$$

LSTM/GRU solves the vanishing gradients?

- The additive memory of the "**forget gate**" prevents small partial gradients from disappearing

- It remembers information over multiple timesteps so the **hidden states** don't disappear across time

- But **LSTM/GRU doesn't guarantee no gradient vanishing/exploding**, it's just better than the vanilla RNN

Is vanishing/exploding gradient just an RNN problem?

- Nope, as long as ***functions keeps getting nested*** in the network and the ***partial derivations needs to be multiplied*** causing unstable gradients

- As long as there are many layers, the functions and gradients gets multiplied in a nested manner

- Without using gates, we can just "***short-circuit" the network and make previous layers interact directly with the current layers***

Throw away your RNN

- "*We fell for RNN, LSTM, and all their variants.* ***Now it is time to drop them!***" – Eugenio Culurciello

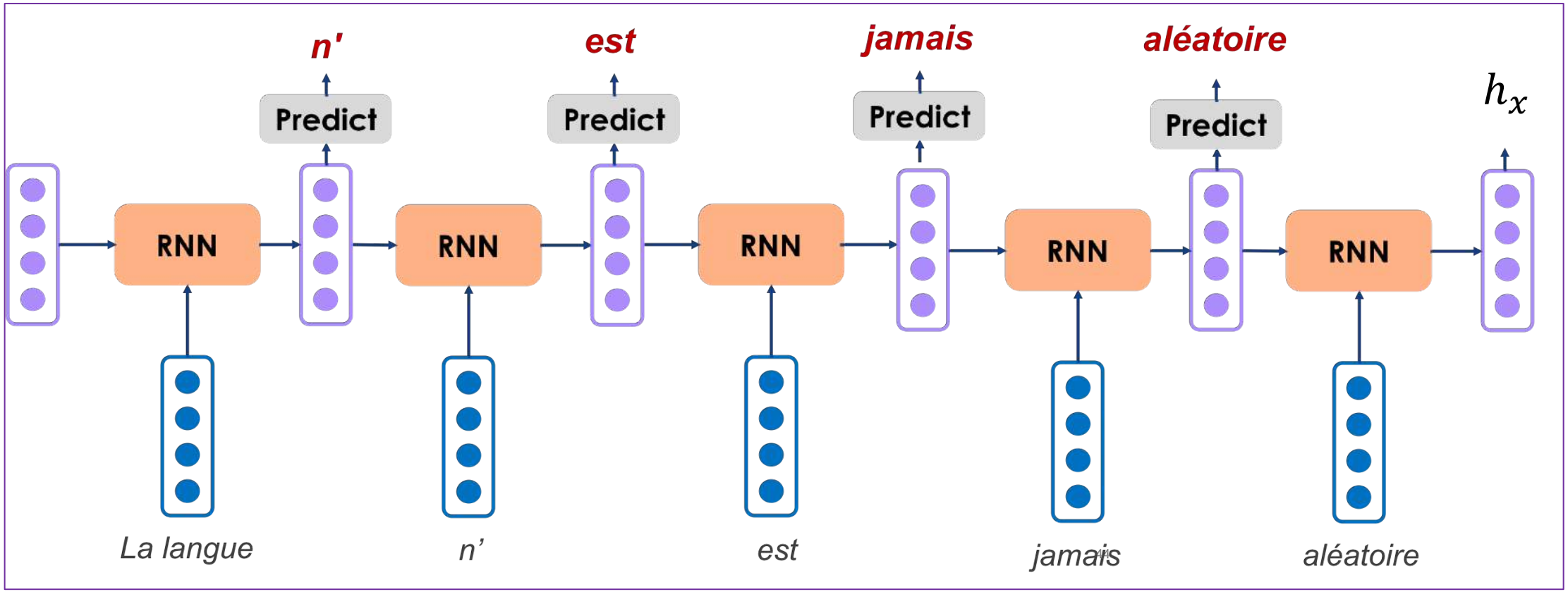
- RNNs are **not hardware friendly**

- **Attention-base models** (e.g. Transformer) outperforms RNN (Vaswani et al. 2017)

- **Hierarchical Attention** (Yang et al. 2016) or **Casual Convolution Networks** (Elbayad et al. 2018) outperforms LSTMs and Transformers

RNN Generation

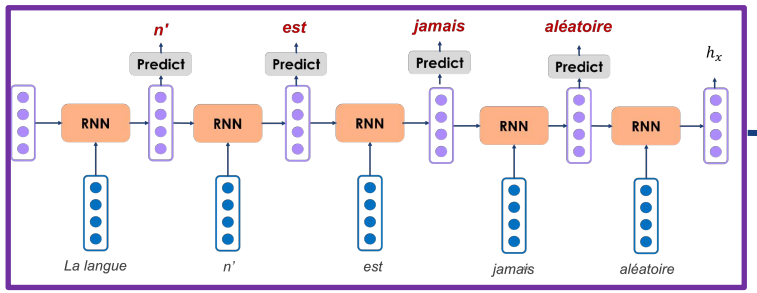
- What if h_0 is something non-random?



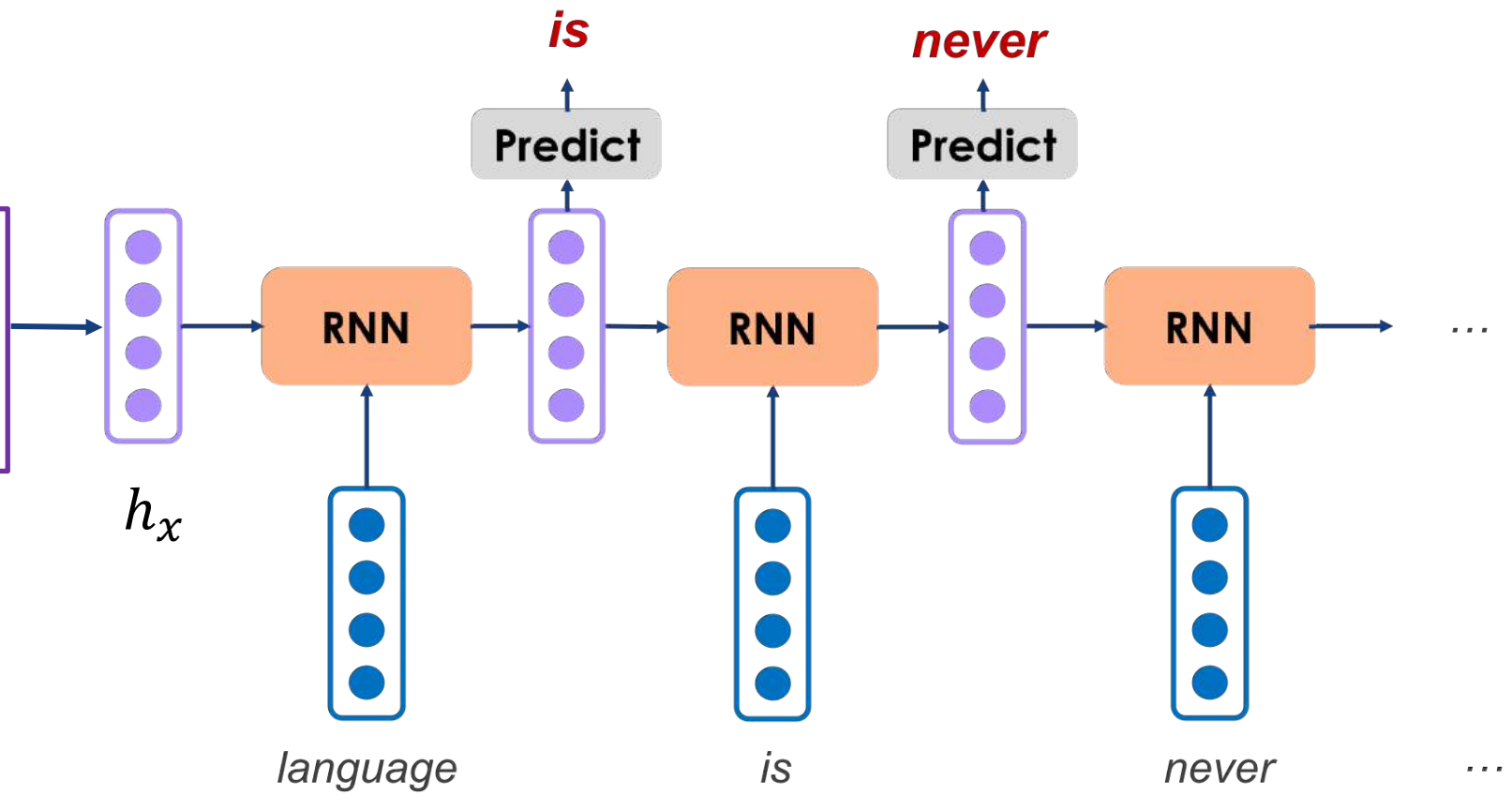
Encoder Model

RNN Generation

- What if h_0 is something non-random?

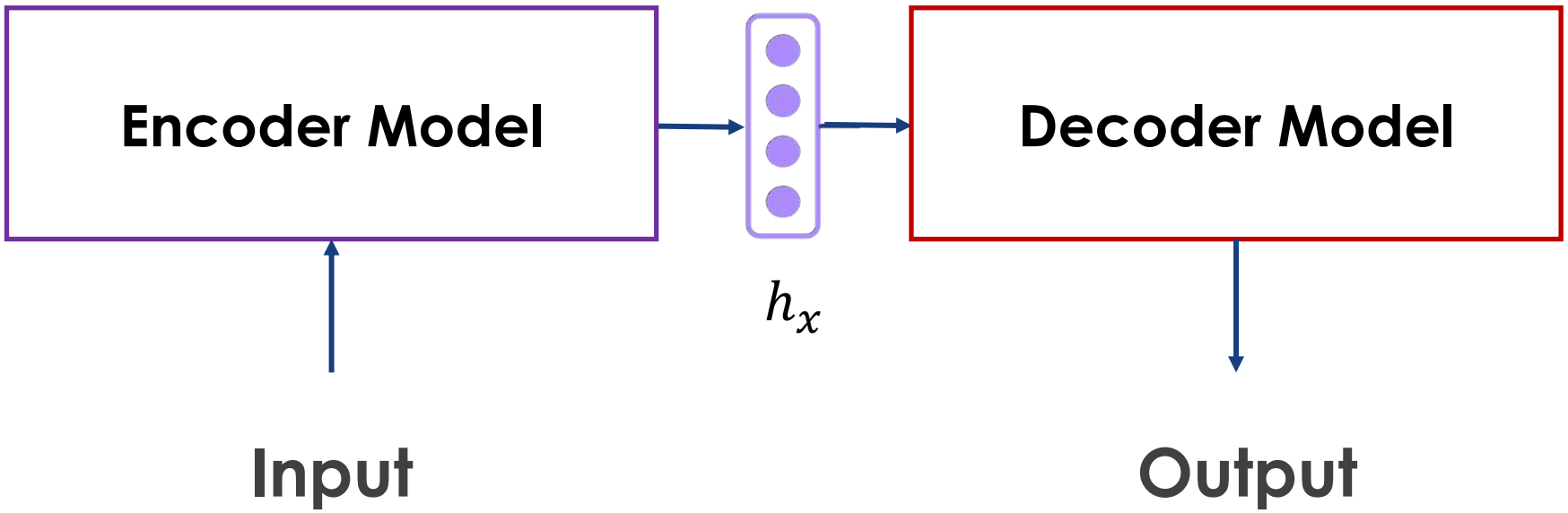


Encoder Model



RNN Generation

- Generalized Encode-Decoder Framework



Conditional RNN Generation

- RNN generation starts with a random hidden state h_0

- What if h_0 is something non-random?

- **Translation:** French -> English
- **Image Captioning:** Image -> Text
- **Speech Recognition:** Audio -> Text
- **Summarization:** Document -> Summary
- **Chatbots:** Utterance -> Response

Conditioned RNN Generation

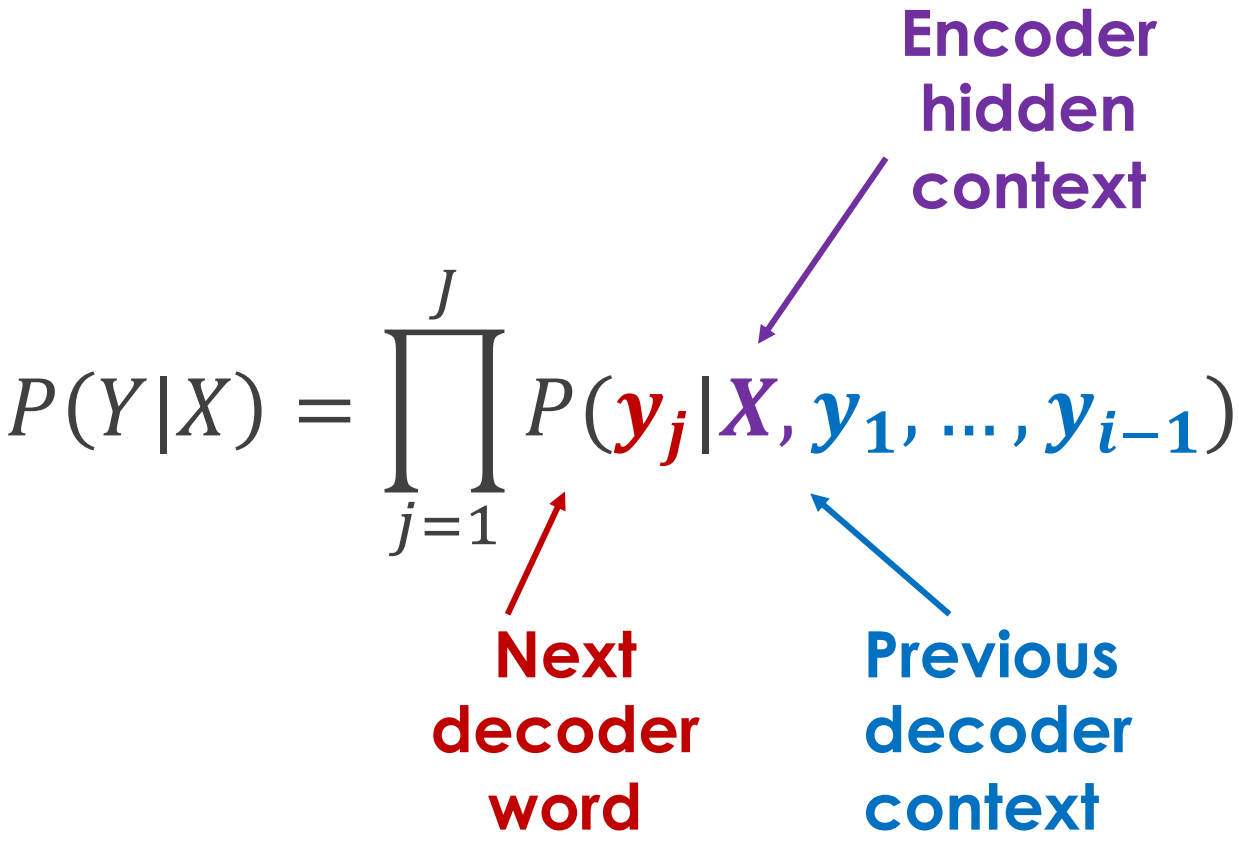
- **Endless possibilities** of what to condition on and what to generate
- But training requires the **paired condition and target generation**
- And relatively **large amount of data is needed** for model to train well

Language Model Probability

$$P(X) = \prod_{i=1}^{I} P(\boldsymbol{x_i} | \boldsymbol{x_1}, \ldots, \boldsymbol{x_{i-1}})$$

Next word **Context**

Conditioned Language Model Probability

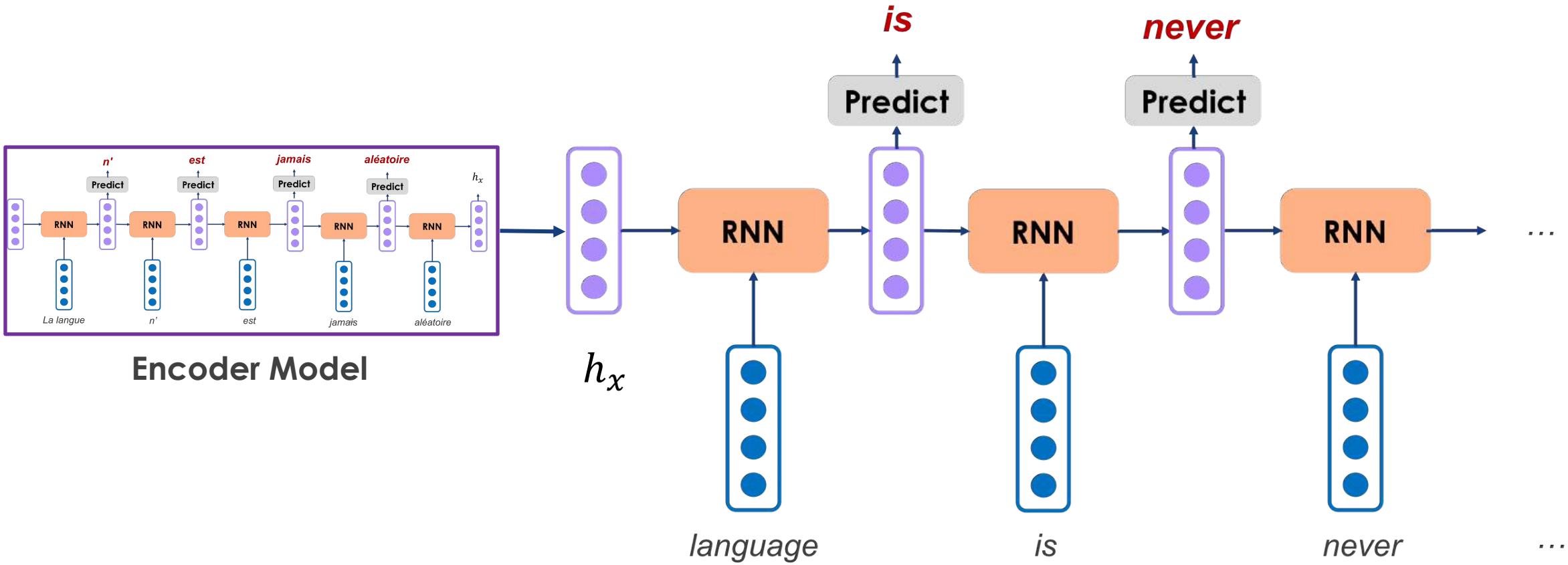


Sequence to Sequence Learning with NN

- Take final hidden state of an encoder model, feed it as the start state of a decoder model (Sutskever et al. 2014)
- RNNs deal naturally with undefined encoding input lengths
- But we're depending on a single hidden state to squeeze information of the inputs to feed to the decoder

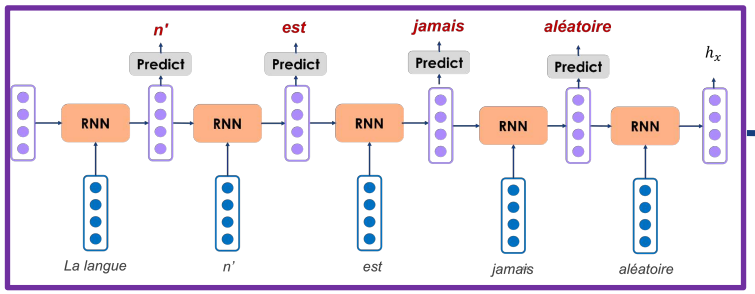
Sequence to Sequence Learning with NN

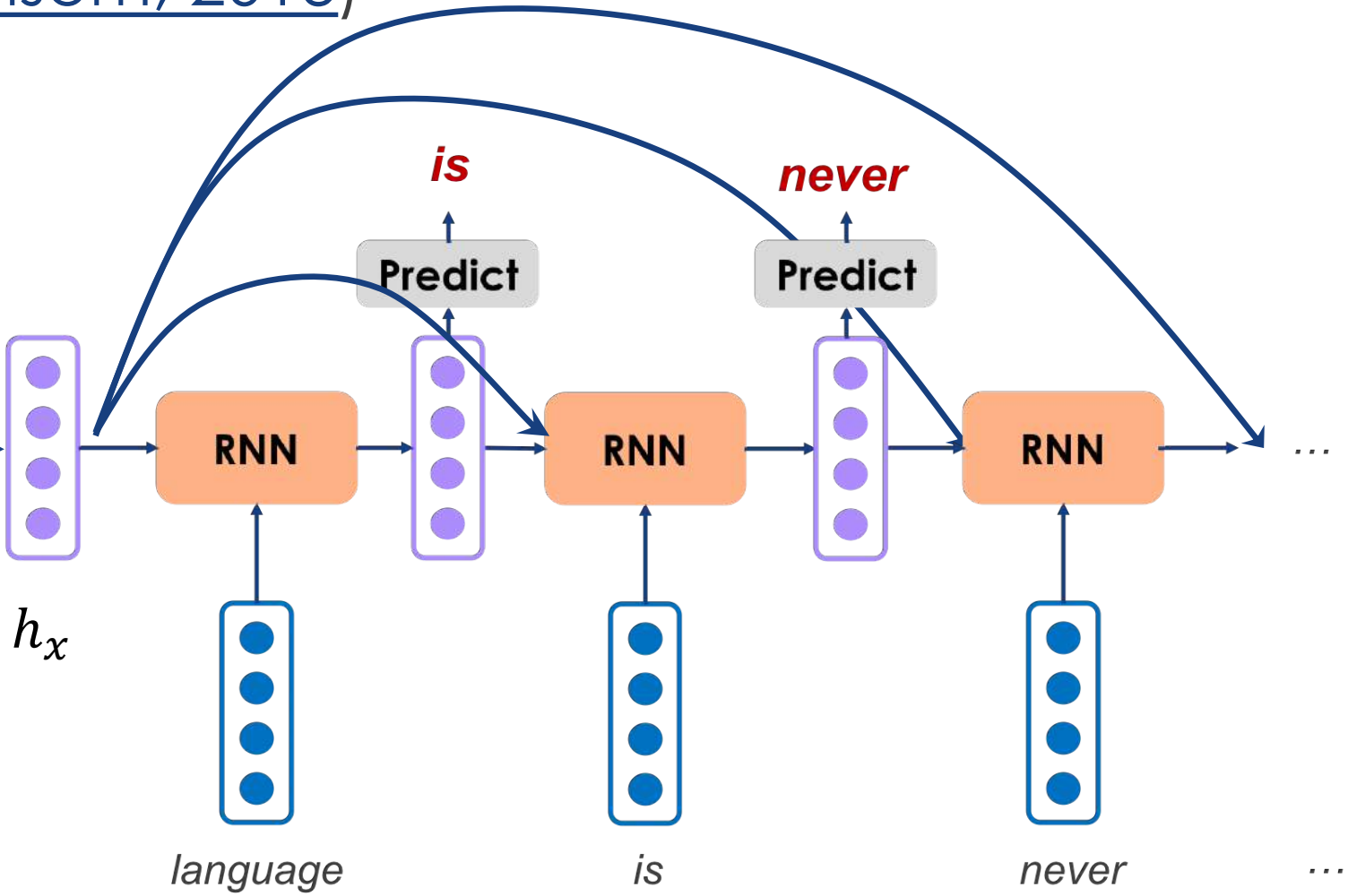
- Take final hidden state of an encoder model, feed it as the start state of a decoder model (Sutskever et al. 2014)



Recurrent Continuous Translation Models

- Add the encoded hidden state at every decoder time step (Kalchbrenner & Blunsom, 2013)





Greedy Decoding

- Generally, we want to find the most probable output given the input
- Simple approximation is to pick the highest probability word at each time step, `torch.max(predictions, 1)`
- Simple causes problem…
  - Often generate “easy” words first
  - Prefers common phrases to one rare word

Beam Search

- Better approximation is to predict k-best outputs at each time step

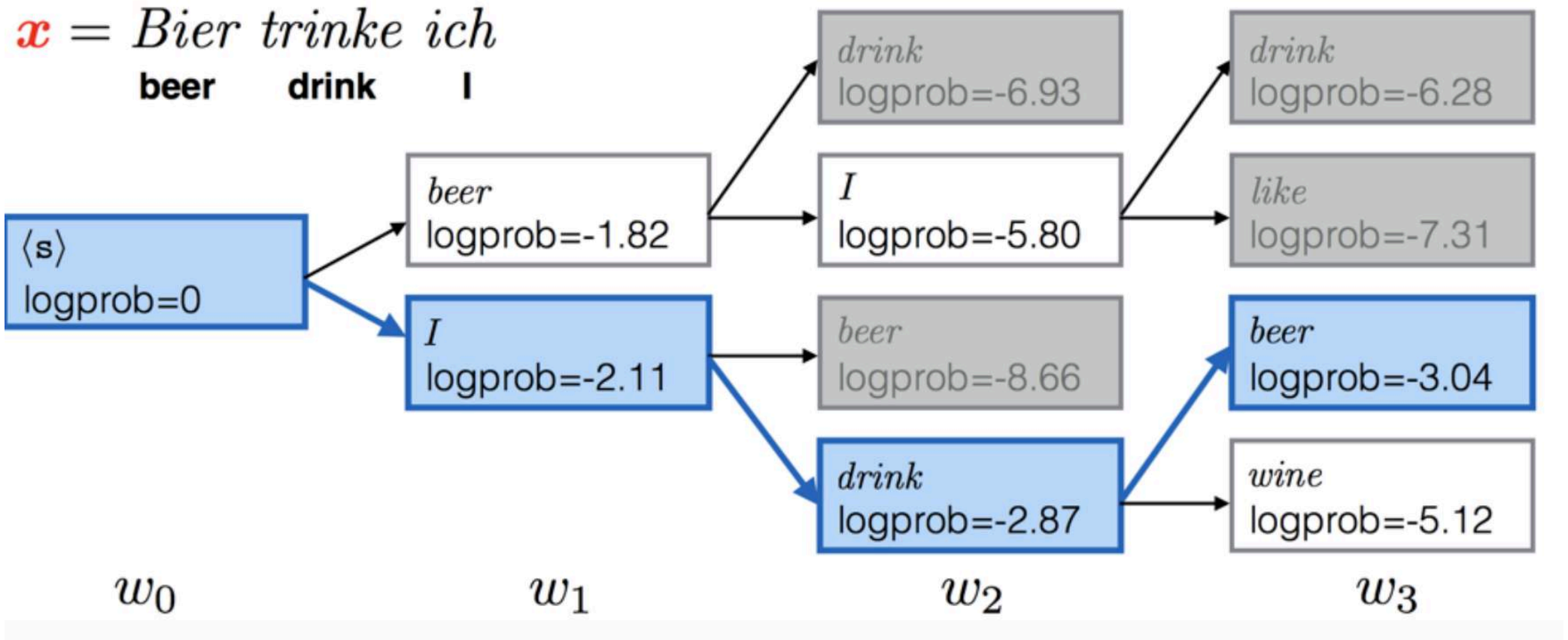


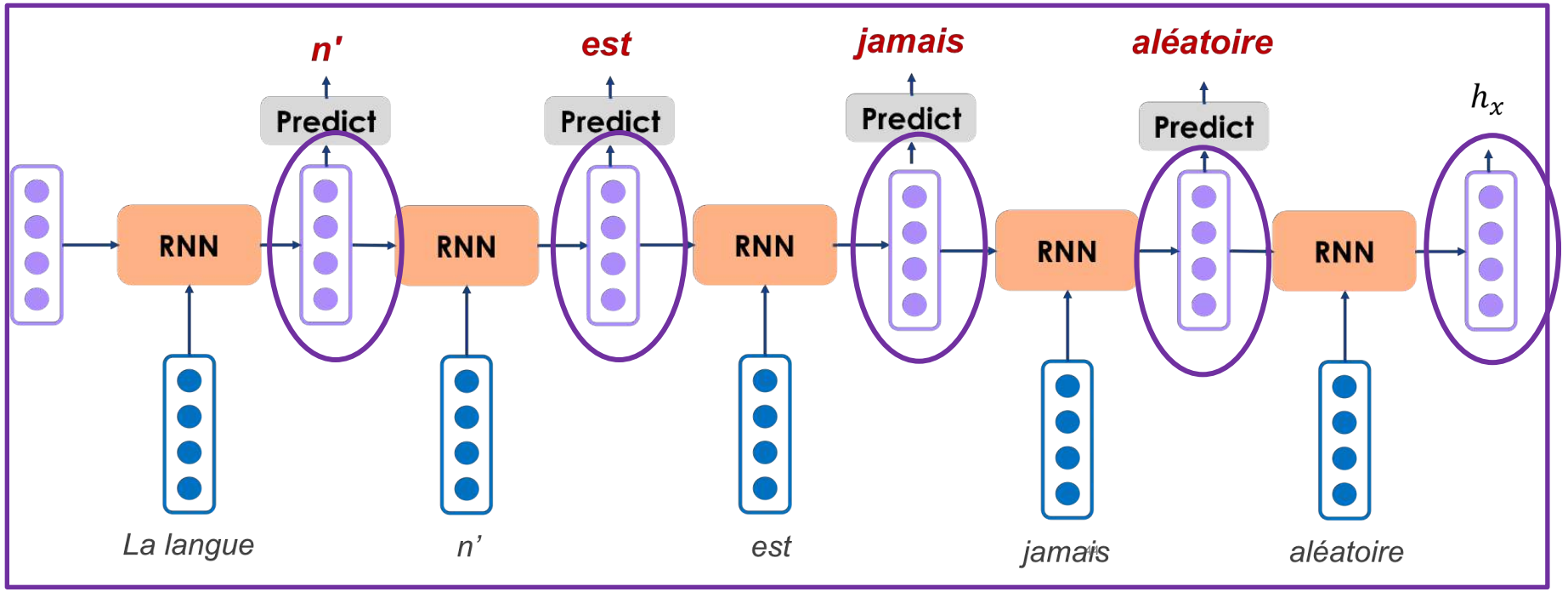
Figure and example from (Dyer, 2018)

Lesson 7: Attention Networks and Transformer

Sentence Representation

"You can't cram the meaning of a whole !@#$-ing sentence into a single %^&-ing vector!"*

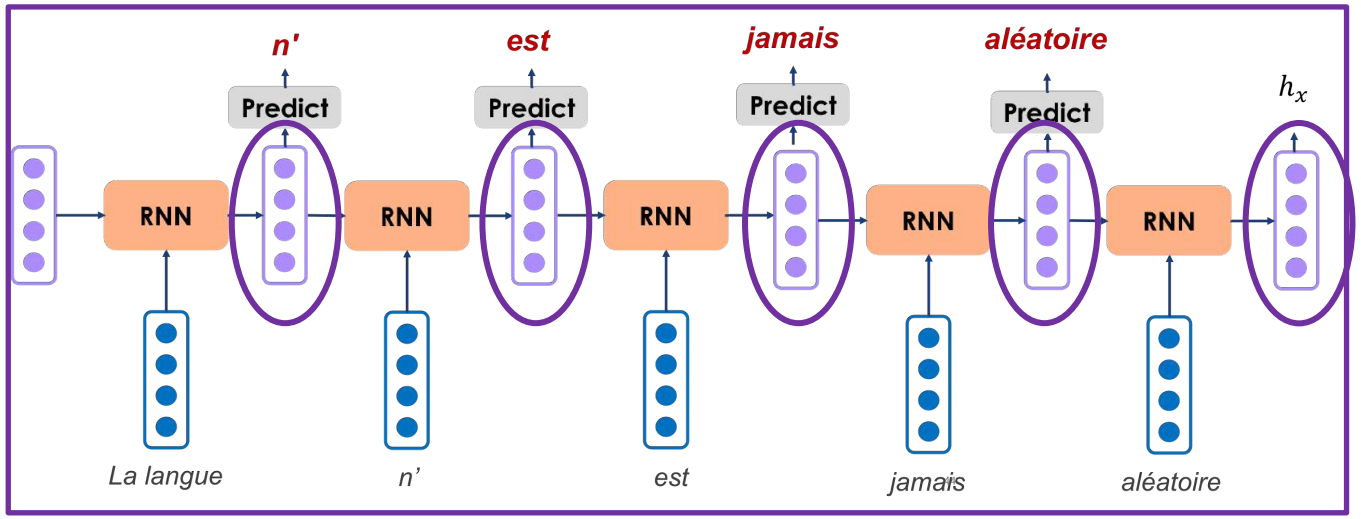
– Raymond Mooney



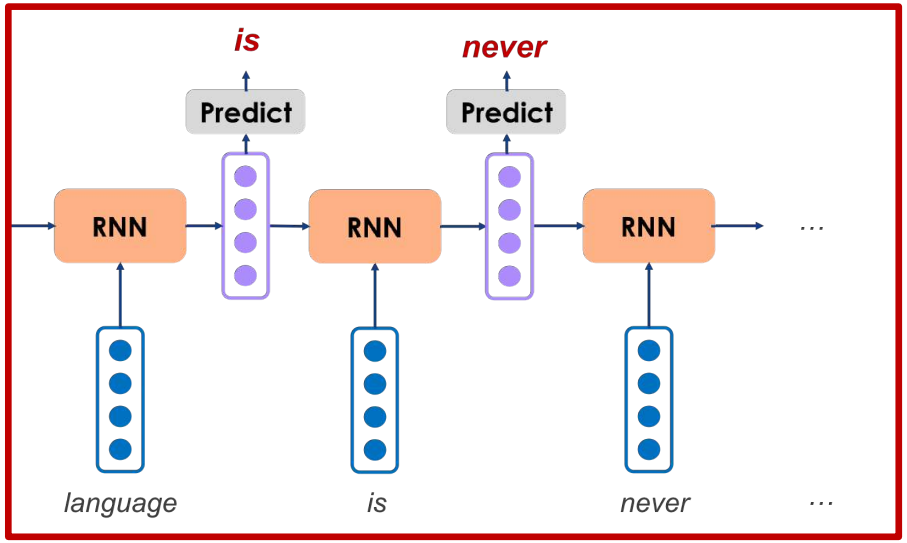
What if instead of taking just h_x, we take **all the hidden states**?

Jointly Learning to Align and Translate

- Encode each word in input into a vector
- When decoding, linearly combine encoded vector vectors weighted by "attention"
- Bahdanau et al. (2015) proposes to learn a MLP mapping between the query-key vector pairs



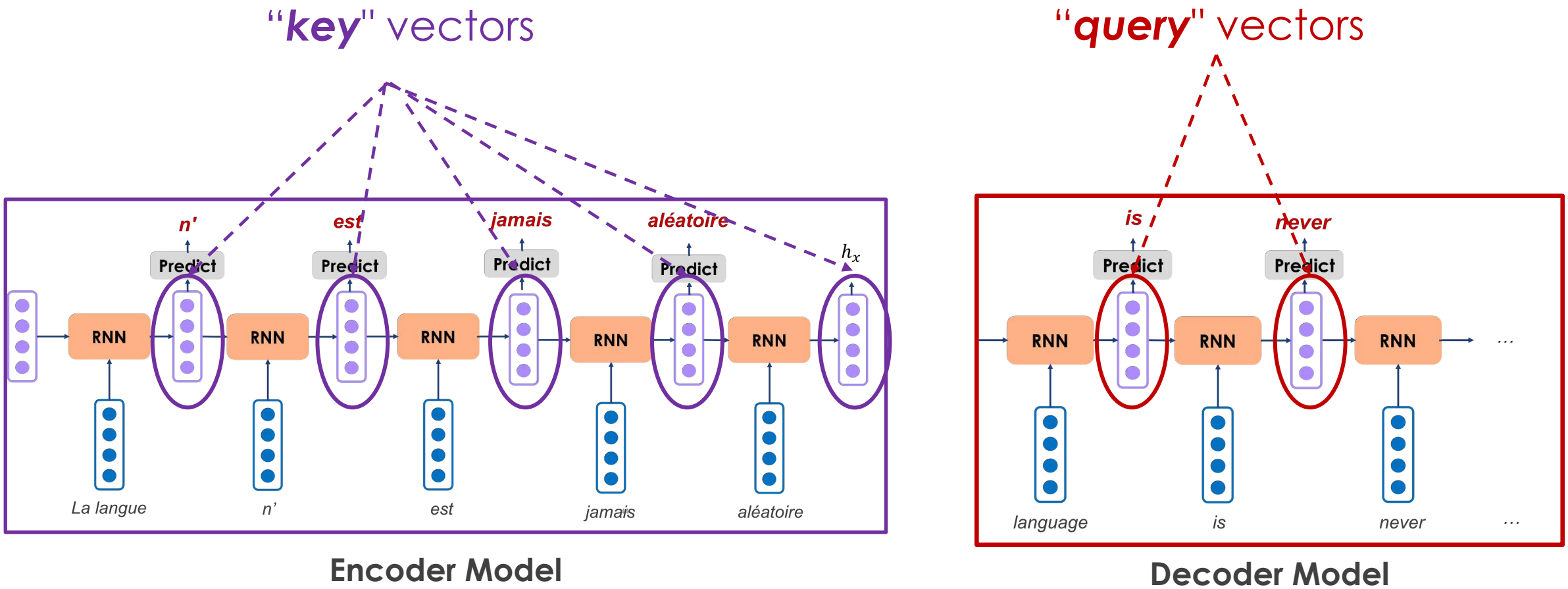
Encoder Model



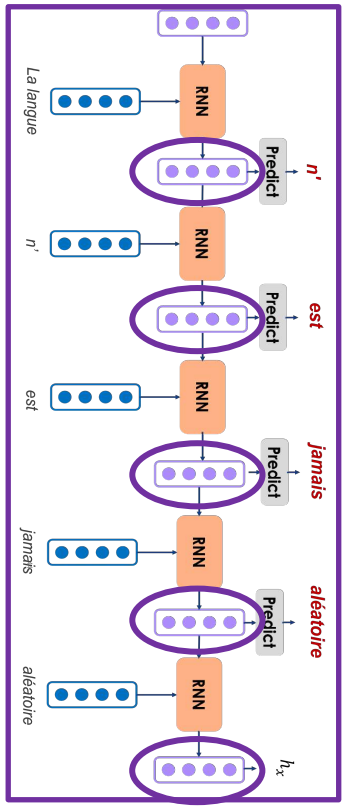
Decoder Model

Jointly Learning to Align and Translate

- Bahdanau et al. (2015) proposes to learn a MLP mapping between the query-key vector pairs



Jointly Learning to Align and Translate



Encoder Model

$a(q_1, k_1) = 7.5$

$a(q_1, k_2) = 1.9$

$a(q_1, k_3) = 2.1$

$a(q_1, k_4) = -3.2$

$a(q_1, k_5) = -0.5$

Softmax

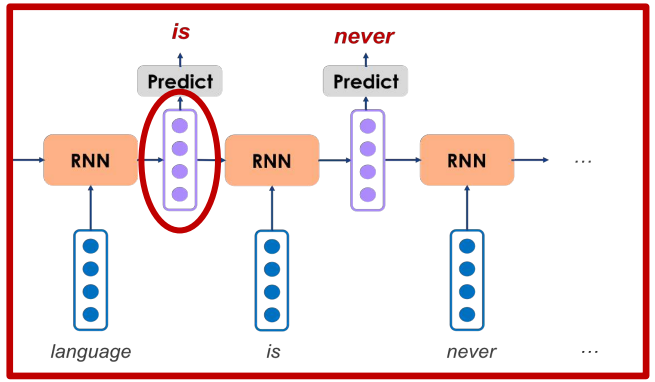
0.9

0.003

0.004

0.00005

0.0003



Decoder Model

Alignment score between *"**key**"* and *"**query**"*

$$a(q, k) = w_2^T (W_1 [q; k])$$

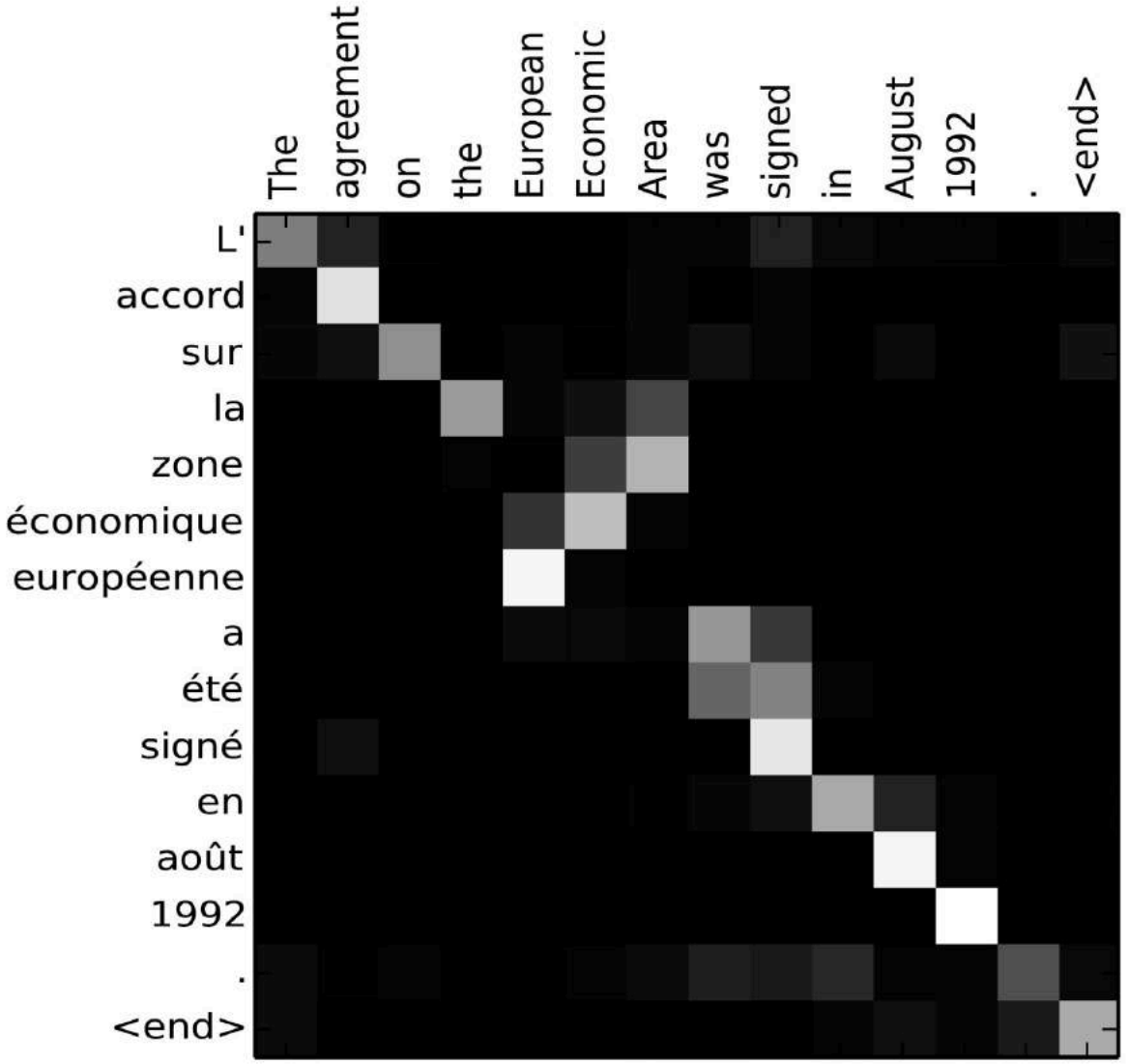
Can be learned by a w_2^T *vector and* W_1 *matrix*

Jointly Learning to Align and Translate

- Bahdanau et al. (2015) proposes to learn a MLP mapping between the query-key vector pairs

- Each decoder state generates a probabilistic "attention" vector that aligns the predicted word at the time step to every input word in the encoder

- As a by-product, it creates a word-alignment matrix (e.g. figure on right)



Jointly Learning to Align and Translate

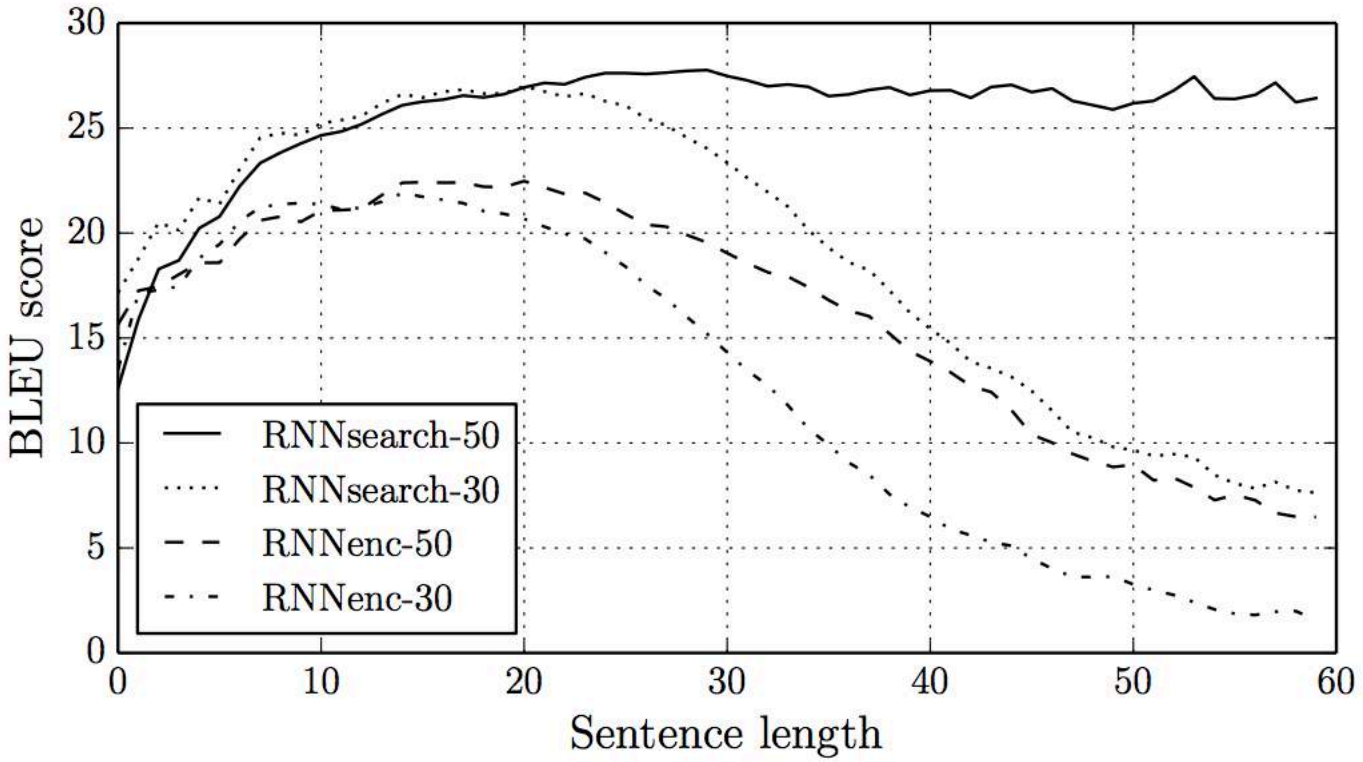


Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.

Bahdanau et al. (2015)

Many Other Attentions

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

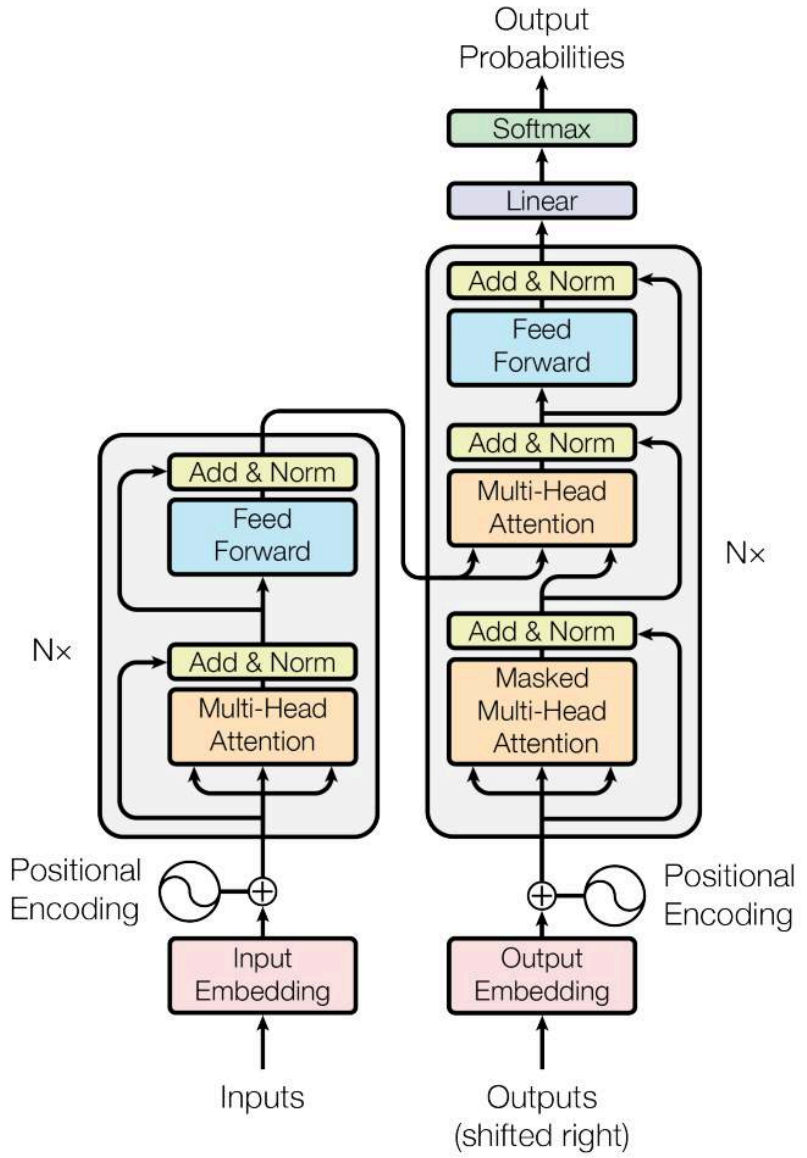
(*) Referred to as "concat" in Luong, et al., 2015 and as "additive attention" in Vaswani, et al., 2017.

(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

(Cite from Lilian Weng's blog)

Attention is All You Need

- Sequence-to-Sequence model entirely based on attention, no recurrence, no convolution
- Fast because only matrix multiplication operation
- It's everywhere…



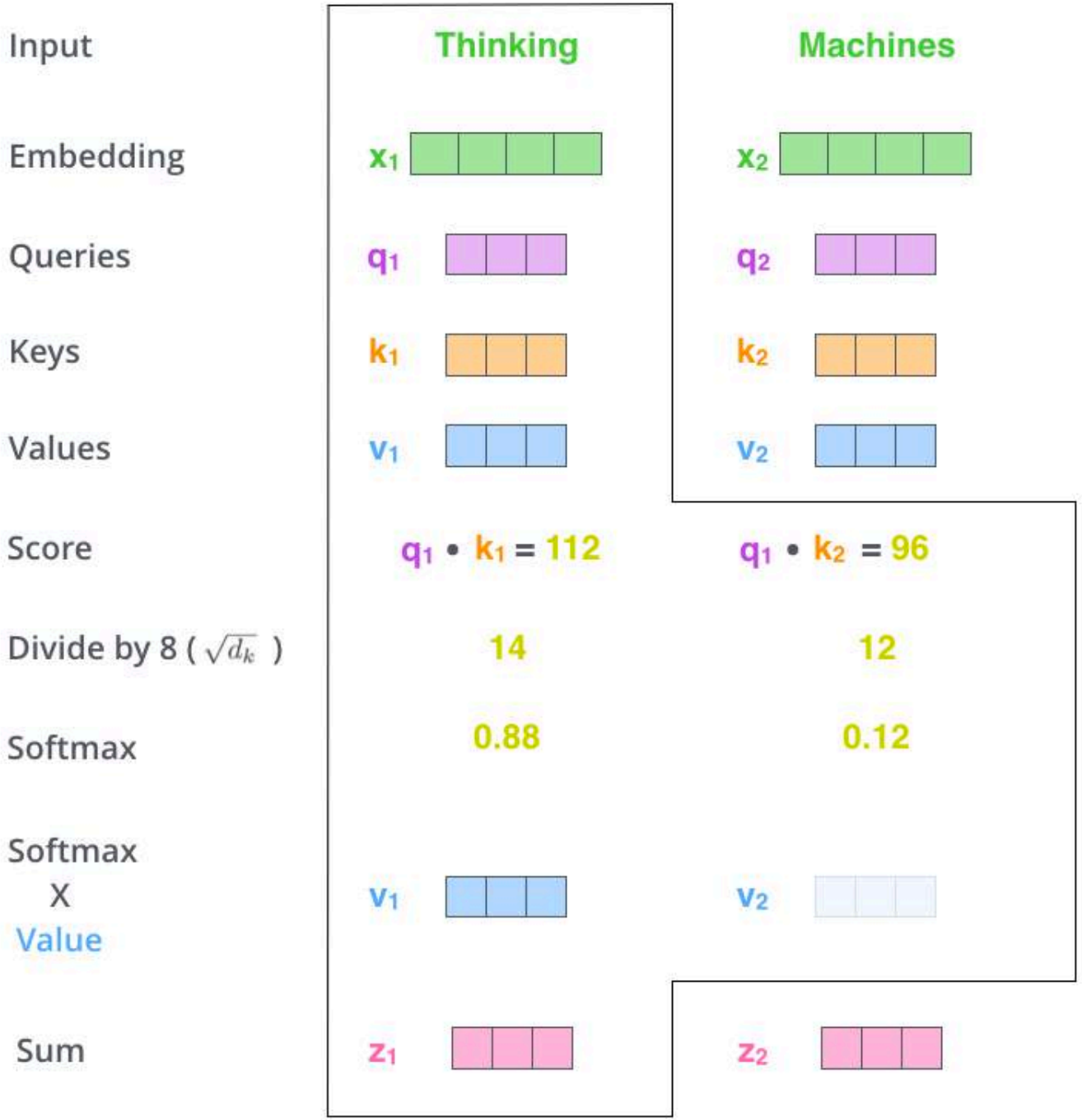
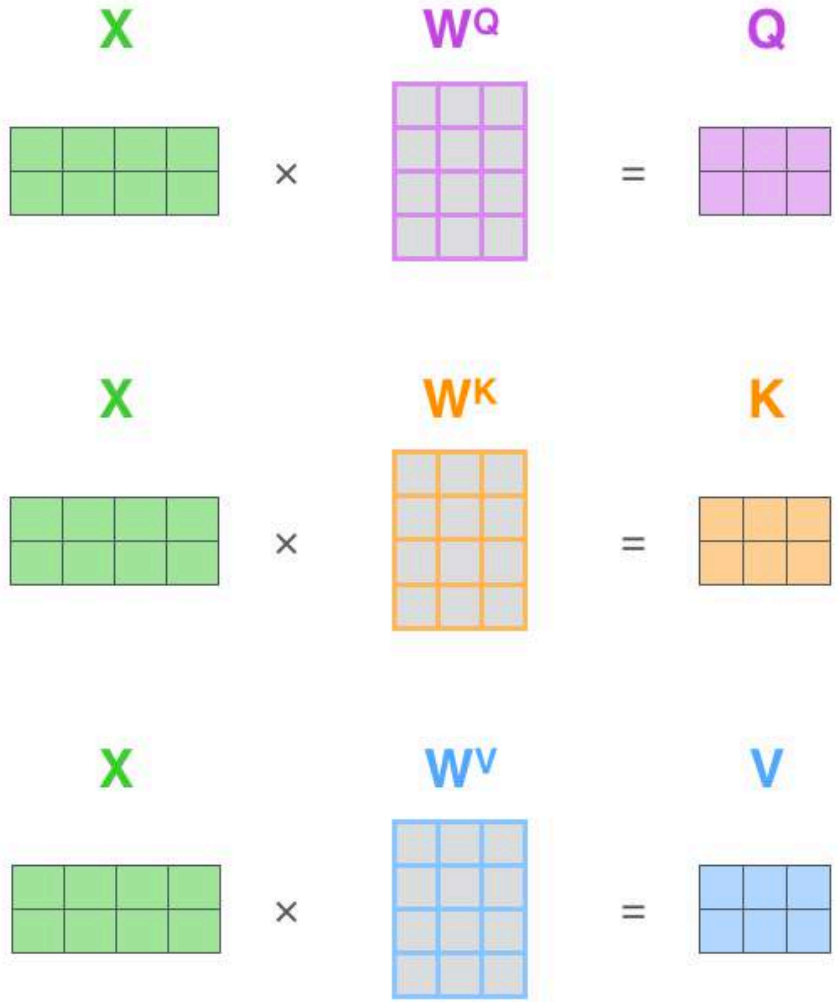


Image from http://jalammar.github.io/illustrated-transformer/

Transformer Self-Attention



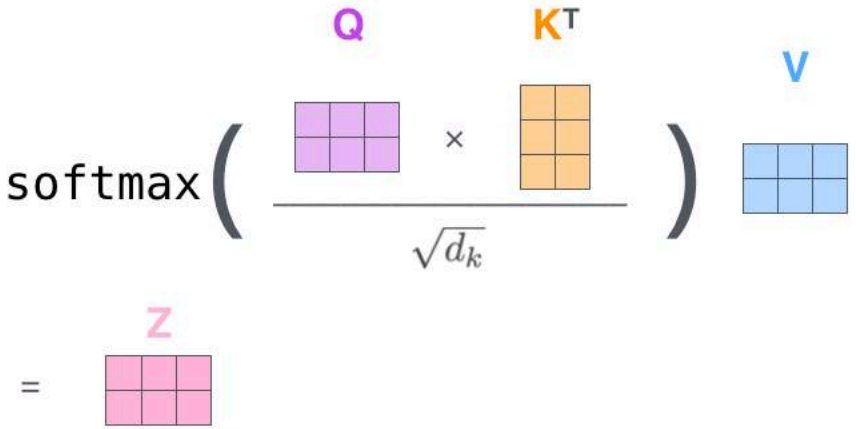


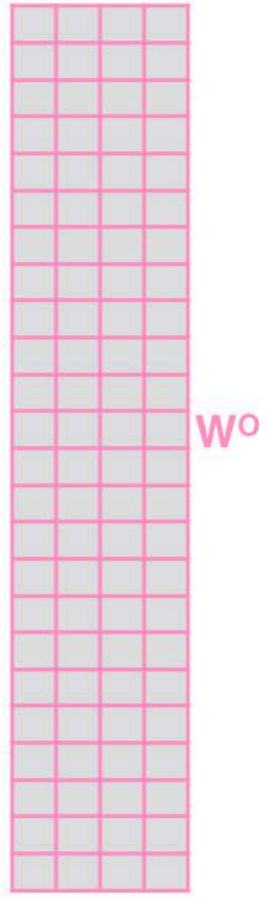
Image from http://jalammar.github.io/illustrated-transformer/

Transformer Self-Attention

1) Concatenate all the attention heads

2) Multiply with a weight matrix W^O that was trained jointly with the model

X



3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

=

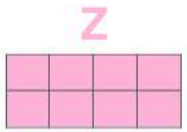


Image from http://jalammar.github.io/illustrated-transformer/

Transformer Self-Attention

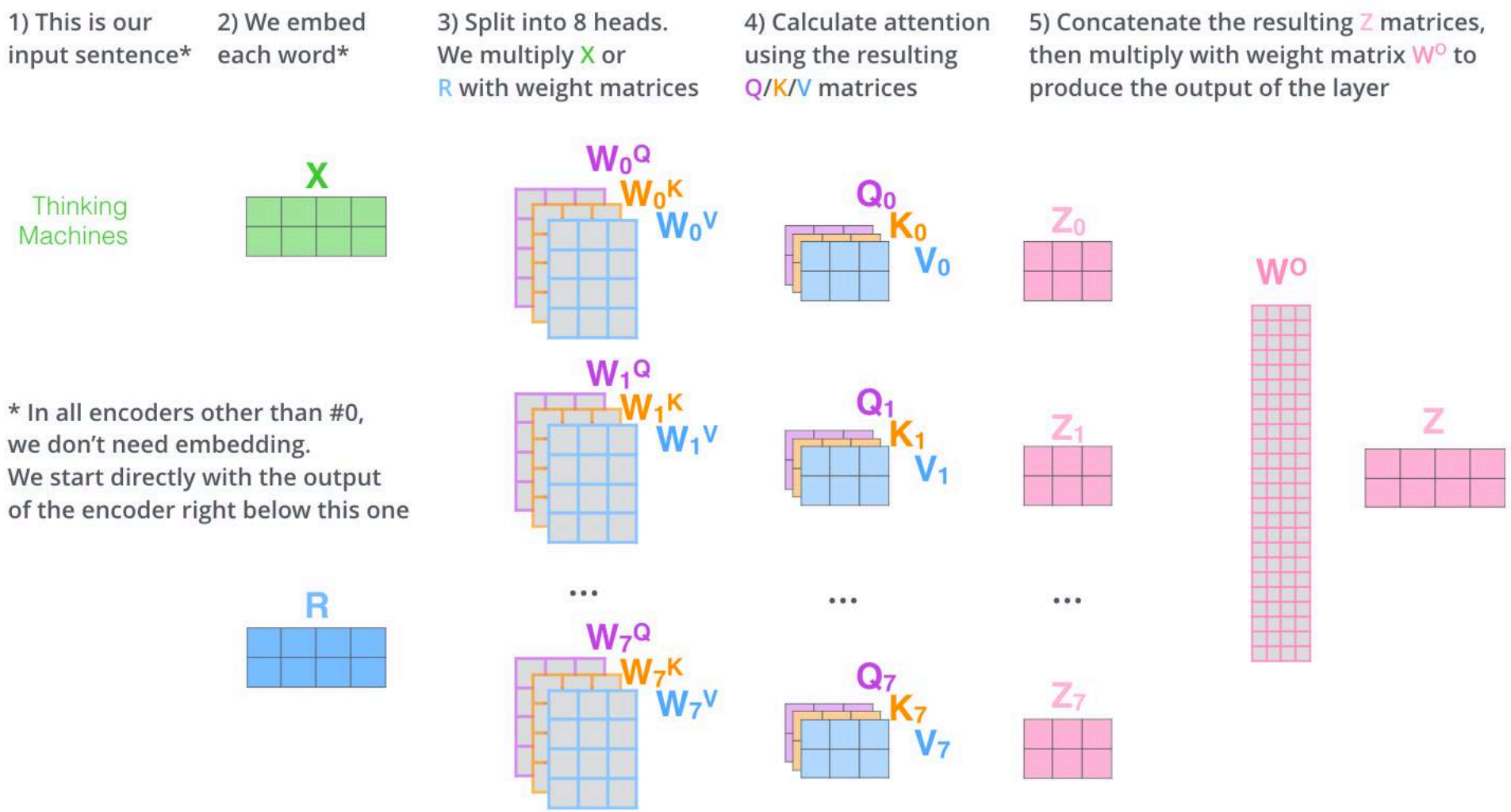


Image from http://jalammar.github.io/illustrated-transformer/

Attention Tricks

- **Self Attention:** Each layer combines words with every other word in sentence
- **Multi-headed Attention:** Multiple attention heads learnt independently
- **Normalized dot product attention:** removes bias
- **Positional Encodings:** Can distinguish positive without stepwise recurrence

Cite from Neubig (2018) CMU NN4NLP Course

Training Tricks

- **Layer Normalization:** Ensure layer outputs remains in range
- **Specialized Training Schedule:** Warm-up and cool-up scheduler adjust defaults Adam learning rates
- **Label Smoothing:** Insert some uncertainty during training to prevent overfitting

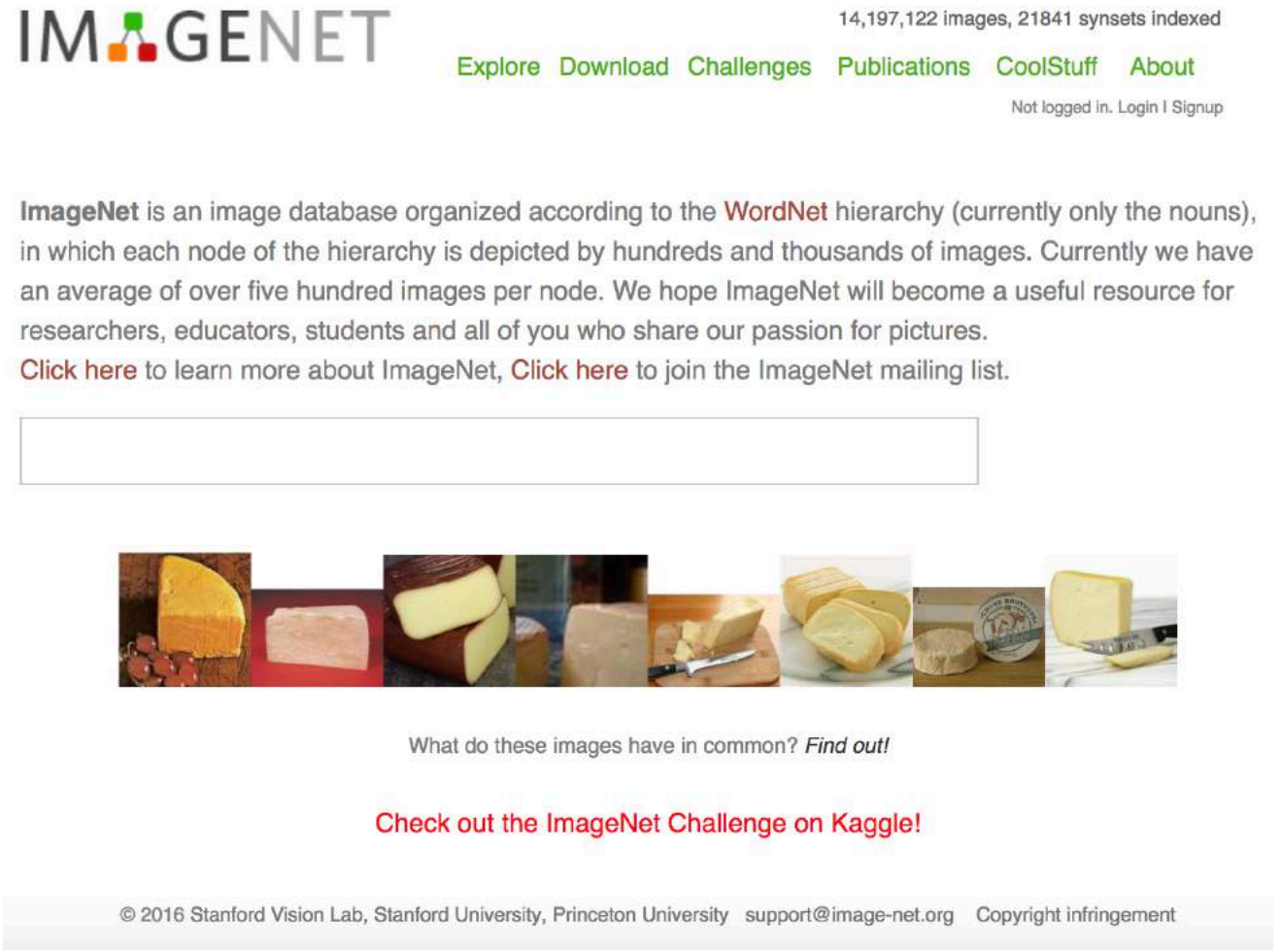
Cite from Neubig (2018) CMU NN4NLP Course

Lesson 8: Sentence Representations

The "ImageNet" Moment

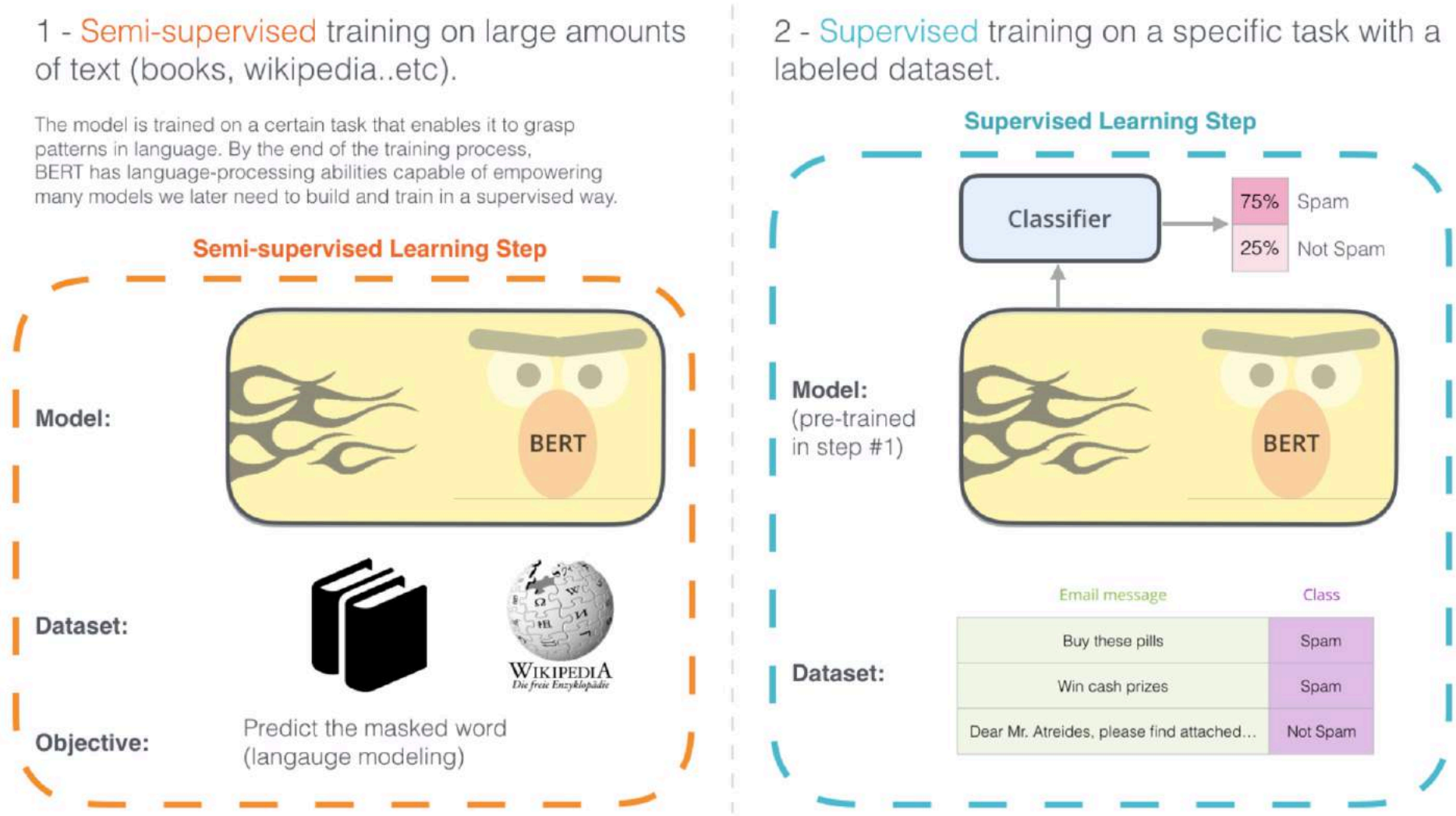
Various Computer Vision Challenges (ImageNet, MS Coco, etc.) started a wave of groups **training models and sharing pre-trained models.**

Fine-tuning / transfer learning for these pre-trained models are faster and *"usually better"* when training new models for other computer vision task.



(*Image from Stanford Vision Lab)

The "ImageNet" Moment for NLP



The two steps of how BERT is developed. You can download the model pre-trained in step 1 (trained on un-annotated data), and only worry about fine-tuning it for step 2. [Source for book icon].

(*Image from Jay Alammar's blog)

Types of Learning

- **Multi-Task Learning:** Training on multiple datasets/tasks

- **Transfer Learning:** Type of Multi-Task Learning, where learning is multi-task but evaluation focus on is on a "downstream" single task

(Image from Burpple.com)

Plethora of Tasks in NLP

- In NLP, there are a plethora of tasks, each requiring different varieties of data
  - **Only text:** e.g. language modeling
  - **Naturally occurring data:** e.g. machine translation
  - **Hand-labeled data:** e.g. most analysis tasks
- And each in many languages, many domains!

This slide is taken from CMU NN4NLP (Neubig, 2019)

Why Multi-Task Learning?

- Ideally, we want a "Strong NLP"
- Honestly, **making labelled dataset is time-consuming and requires lots of resources,** so the resultant dataset created are often small-ish
- By combining multiple tasks, we have more data =)

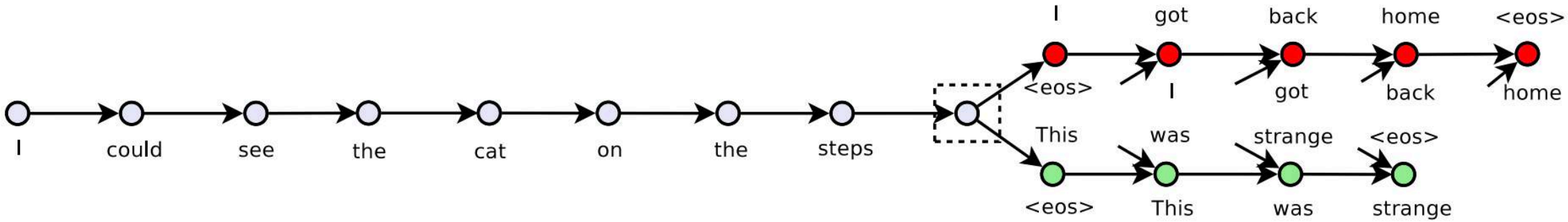
Why Transfer-Learning?

- Even with Multi-Task Learning **dataset with labels are still small relative to the size of text-only dataset** (e.g. Wikipedia, Common Crawl, News articles)

- What if we can learn a generalized language model then reuse it to fine-tune for downstream tasks when necessary?

- How do we represent a sentence?

Skip-Thought Vectors (Kiros et al. 2017)



- Get a sentence triplets, from the focus sentence, generate both sentence before (red) and sentence after (green)
- Learn a Seq2Seq model, use the learnt encoder as a sentence encoder

Objective. Given a tuple (s_{i-1}, s_i, s_{i+1}), the objective optimized is the sum of the log-probabilities for the forward and backward sentences conditioned on the encoder representation:

$$\sum_t \log P(w_{i+1}^t | w_{i+1}^{<t}, \mathbf{h}_i) + \sum_t \log P(w_{i-1}^t | w_{i-1}^{<t}, \mathbf{h}_i) \quad (10)$$

Semantic Textual Similarity

Dataset	Domain	Score	Sent1	Sent2
STS2012-gold	surprise.OnWN	5.0	render one language in another language ...	restate (words) from one language into another ...
STS2012-gold	surprise.OnWN	3.25	nations unified by shared interests, history or ...	a group of nations having common interests. ...
STS2012-gold	surprise.OnWN	3.25	convert into absorbable substances, (as if) with ...	soften or disintegrate by means of chemical act ...
STS2012-gold	surprise.OnWN	4.0	devote or adapt exclusively to an skill, ...	devote oneself to a special area of work. ...
STS2012-gold	surprise.OnWN	3.25	elevated wooden porch of a house ...	a porch that resembles the deck on a ship. ...
STS2012-gold	surprise.OnWN	4.0	either half of an archery bow ...	either of the two halves of a bow from handle to ...
STS2012-gold	surprise.OnWN	3.333	a removable device that is an accessory to la ...	a supplementary part or accessory. ...
STS2012-gold	surprise.OnWN	4.75	restrict or confine	place limits on (extent or access). ...
STS2012-gold	surprise.OnWN	0.5	orient, be positioned	be opposite.
STS2012-gold	surprise.OnWN	4.75	Bring back to life, return from the dead ...	cause to become alive again. ...

Given a dataset of **pairs of sentences and a similarity score** assigned by humans, learn a model to assign a score given two sentences

Metric: Correlation score with human annotations

Famous datasets:

- SemEval STS
- SICK
- MS Paraphrase Corpus
- Quora Question Pairs
- Corpus of Linguistics Acceptability

Siamese Network for STS

Muller and Thyagarajan (2016) trained a network with two separate LSTM layers and the **last layer is a Mahattan distance between the output of the two LSTMs**

Works well on the SICK dataset, outperforms the Skip-Thought

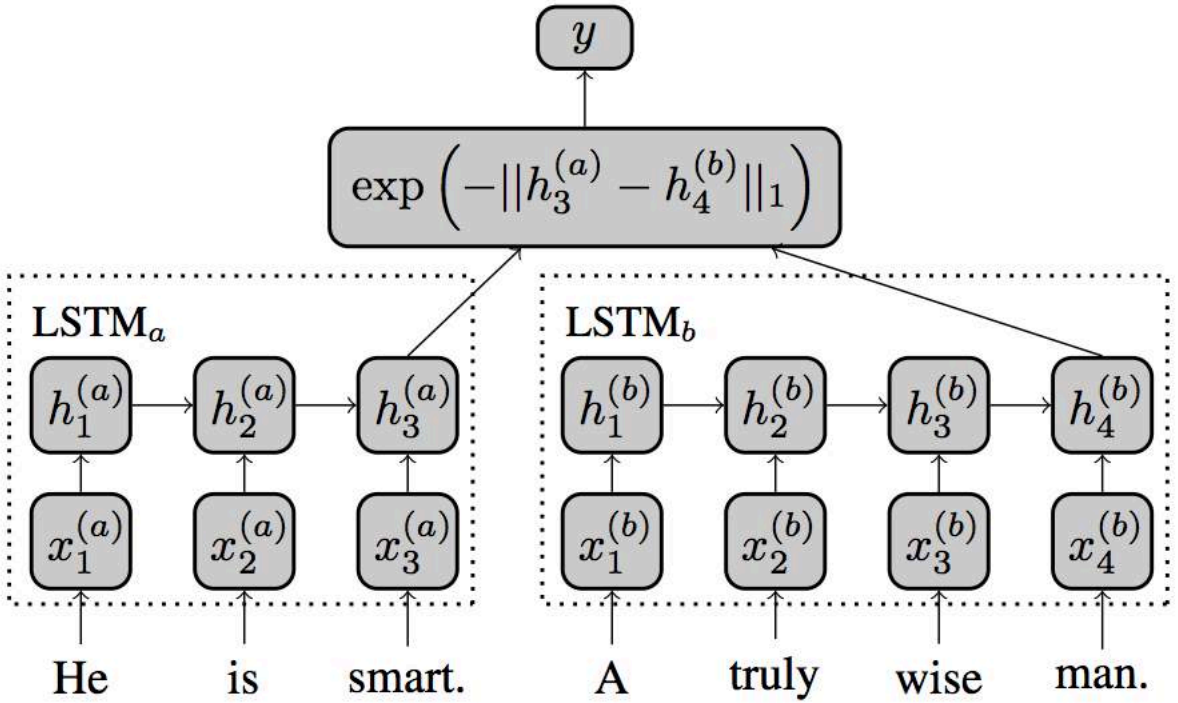


Figure 1: Our model uses an LSTM to read in word-vectors representing each input sentence and employs its final hidden state as a vector representation for each sentence. Subsequently, the similarity between these representations is used as a predictor of semantic similarity.

Stanford Natural Language Inference Dataset

Text	Judgments	Hypothesis
A man inspects the uniform of a figure in some East Asian country.	contradiction C C C C C	The man is sleeping
An older and younger man smiling.	neutral N N E N N	Two men are smiling and laughing at the cats playing on the floor.
A black race car starts up in front of a crowd of people.	contradiction C C C C C	A man is driving down a lonely road.
A soccer game with multiple males playing.	entailment E E E E E	Some men are playing a sport.
A smiling costumed woman is holding an umbrella.	neutral N N E C N	A happy woman in a fairy costume holds an umbrella.

The SNLI corpus (version 1.0) is a collection of 570k human-written English sentence pairs manually labeled for balanced classification with the labels ***entailment***, ***contradiction***, **and** ***neutral***, supporting the task of natural language inference (NLI), also known as recognizing textual entailment (RTE).

InferSent: Supervised Learning of Universal Sentence Representations from Natural Language Inference

1. Initialize a model for sentence encoder,

2. Put the pair of sentences through the encoder and generate the sentence vector, u and v.

3. Compute the two distance / similarity metrics |u-v| and u*v

4. Concat output of 2 and 3 put it through feed-forward net,

5. End up with a 3 way softmax

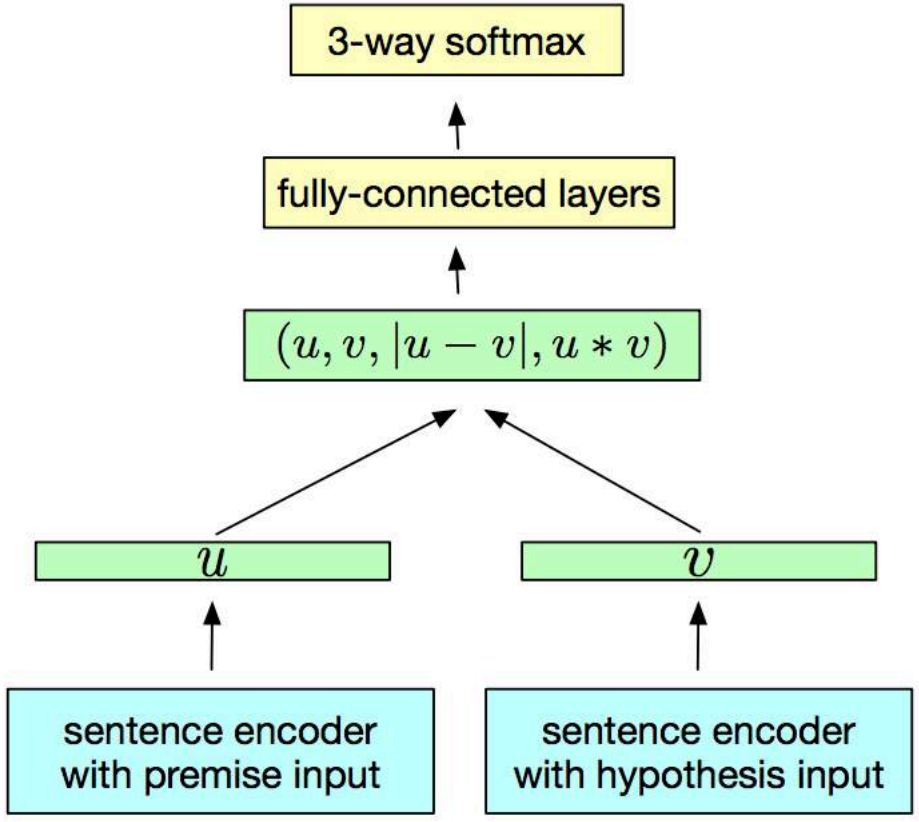


Figure 1: **Generic NLI training scheme.**

InferSent: Supervised Learning of Universal Sentence Representations from Natural Language Inference

y = [0, 0, 1]

z = FFN(u, v, | u-v | , u*v)

z has shape [1 x 3]

Last layer options:

(i) sigmoid(z) , multi-label, multi-class
(ii) softmax(z) , single-label, multi-class

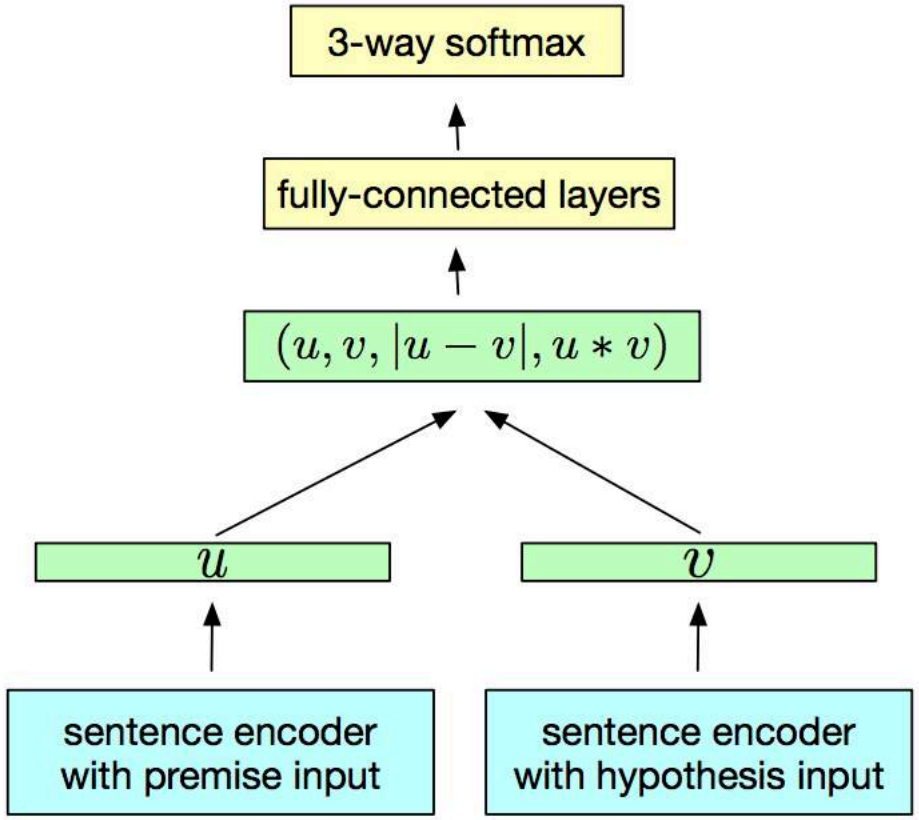


Figure 1: **Generic NLI training scheme.**

InferSent: Supervised Learning of Universal Sentence Representations from Natural Language Inference

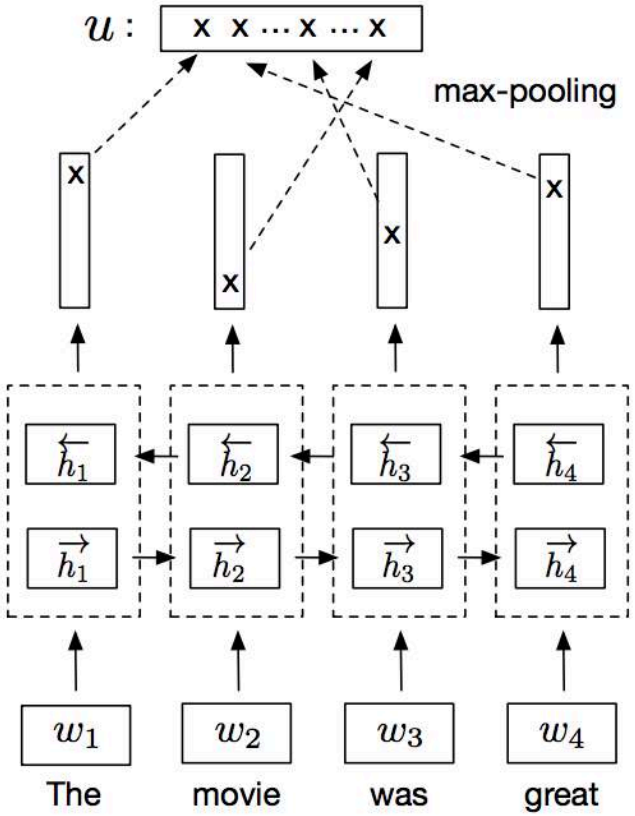


Figure 2: **Bi-LSTM max-pooling network.**

Model	dim	NLI dev	NLI test	Transfer micro	Transfer macro
LSTM	2048	81.9	80.7	79.5	78.6
GRU	4096	82.4	81.8	81.7	80.9
BiGRU-last	4096	81.3	80.9	82.9	81.7
BiLSTM-Mean	4096	79.0	78.2	83.1	81.7
Inner-attention	4096	82.3	82.5	82.1	81.0
HConvNet	4096	83.7	83.4	82.0	80.9
BiLSTM-Max	4096	**85.0**	**84.5**	**85.2**	**83.7**

Table 3: **Performance of sentence encoder architectures** on SNLI and (aggregated) transfer tasks. Dimensions of embeddings were selected according to best aggregated scores (see Figure 5).

Model	MR	CR	SUBJ	MPQA	SST	TREC	MRPC	SICK-R	SICK-E	STS14
Unsupervised representation training (unordered sentences)										
Unigram-TFIDF	73.7	**79.2**	90.3	82.4	-	85.0	73.6/81.7	-	-	.58/.57
ParagraphVec (DBOW)	60.2	66.9	76.3	70.7	-	59.4	72.9/81.1	-	-	.42/.43
SDAE	74.6	78.0	90.8	86.9	-	78.4	**73.7**/80.7	-	-	.37/.38
SIF (GloVe + WR)	-	-	-	-	82.2	-	-	-	**84.6**	**.69**/ -
word2vec BOW†	77.7	79.8	90.9	88.3	79.7	83.6	72.5/81.4	0.803	78.7	.65/.64
fastText BOW†	78.3	81.0	**92.4**	87.8	**81.9**	84.8	**73.9/82.0**	0.815	78.3	.63/.62
GloVe BOW†	**78.7**	78.5	91.6	87.6	79.8	83.6	72.1/80.9	0.800	78.6	.54/.56
GloVe Positional Encoding†	78.3	77.4	91.1	87.1	80.6	83.3	72.5/81.2	0.799	77.9	.51/.54
BiLSTM-Max (untrained)†	77.5	**81.3**	89.6	**88.7**	80.7	**85.8**	73.2/81.6	**0.860**	83.4	.39/.48
Unsupervised representation training (ordered sentences)										
FastSent	70.8	78.4	88.7	80.6	-	76.8	72.2/80.3	-	-	**.63/.64**
FastSent+AE	71.8	76.7	88.8	81.5	-	80.4	71.2/79.1	-	-	.62/.62
SkipThought	76.5	80.1	93.6	87.1	82.0	**92.2**	**73.0/82.0**	**0.858**	82.3	.29/.35
SkipThought-LN	**79.4**	**83.1**	**93.7**	**89.3**	82.9	88.4	-	**0.858**	79.5	.44/.45
Supervised representation training										
CaptionRep (bow)	61.9	69.3	77.4	70.8	-	72.2	73.6/81.9	-	-	.46/.42
DictRep (bow)	76.7	78.7	90.7	87.2	-	81.0	68.4/76.8	-	-	**.67**/**.70**
NMT En-to-Fr	64.7	70.1	84.9	81.5	-	82.8	69.1/77.1	-		.43/.42
Paragram-phrase	-	-	-	-	79.7	-	-	0.849	83.1	**.71**/ -
BiLSTM-Max (on SST)†	(*)	83.7	90.2	89.5	(*)	86.0	72.7/80.9	0.863	83.1	.55/.54
BiLSTM-Max (on SNLI)†	79.9	84.6	92.1	**89.8**	83.3	**88.7**	75.1/82.3	**0.885**	**86.3**	.68/.65
BiLSTM-Max (on AllNLI)†	**81.1**	**86.3**	**92.4**	**90.2**	**84.6**	88.2	**76.2/83.1**	**0.884**	**86.3**	**.70/.67**
Supervised methods (directly trained for each task – no transfer)										
Naive Bayes - SVM	79.4	81.8	93.2	86.3	83.1	-	-	-	-	-
AdaSent	83.1	86.3	95.5	93.3	-	92.4	-	-	-	-
TF-KLD	-	-	-	-	-	-	80.4/85.9	-	-	-
Illinois-LH	-	-	-	-	-	-	-	-	84.5	-
Dependency Tree-LSTM	-	-	-	-	-	-	-	0.868	-	-

Deep Averaging Network

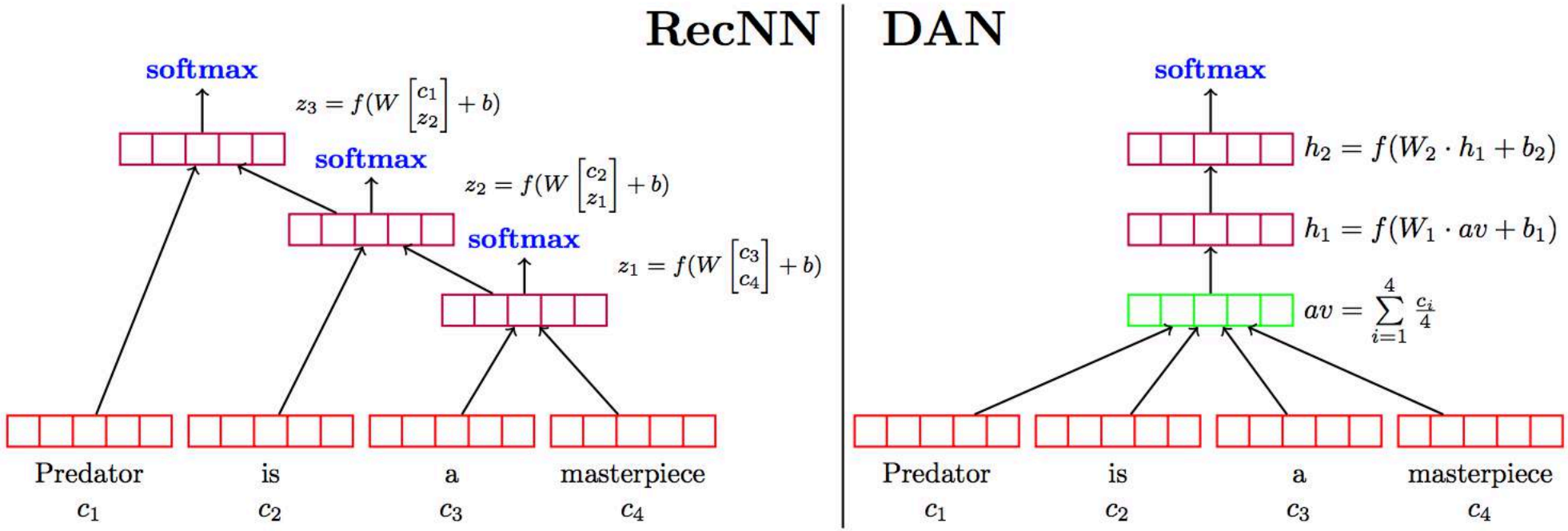


Figure 1: On the left, a **RecNN** is given an input sentence for sentiment classification. Softmax layers are placed above every internal node to avoid vanishing gradient issues. On the right is a two-layer **DAN** taking the same input. While the **RecNN** has to compute a nonlinear representation (purple vectors) for every node in the parse tree of its input, this **DAN** only computes two nonlinear layers for every possible input.

Universal Sentence Encoder (Transfer Learning Datasets)

MR : Movie review snippet sentiment on a five star scale (Pang and Lee, 2005).

CR : Sentiment of sentences mined from customer reviews (Hu and Liu, 2004).

SUBJ : Subjectivity of sentences from movie reviews and plot summaries (Pang and Lee, 2004).

MPQA : Phrase level opinion polarity from news data (Wiebe et al., 2005).

TREC : Fine grained question classification sourced from TREC (Li and Roth, 2002).

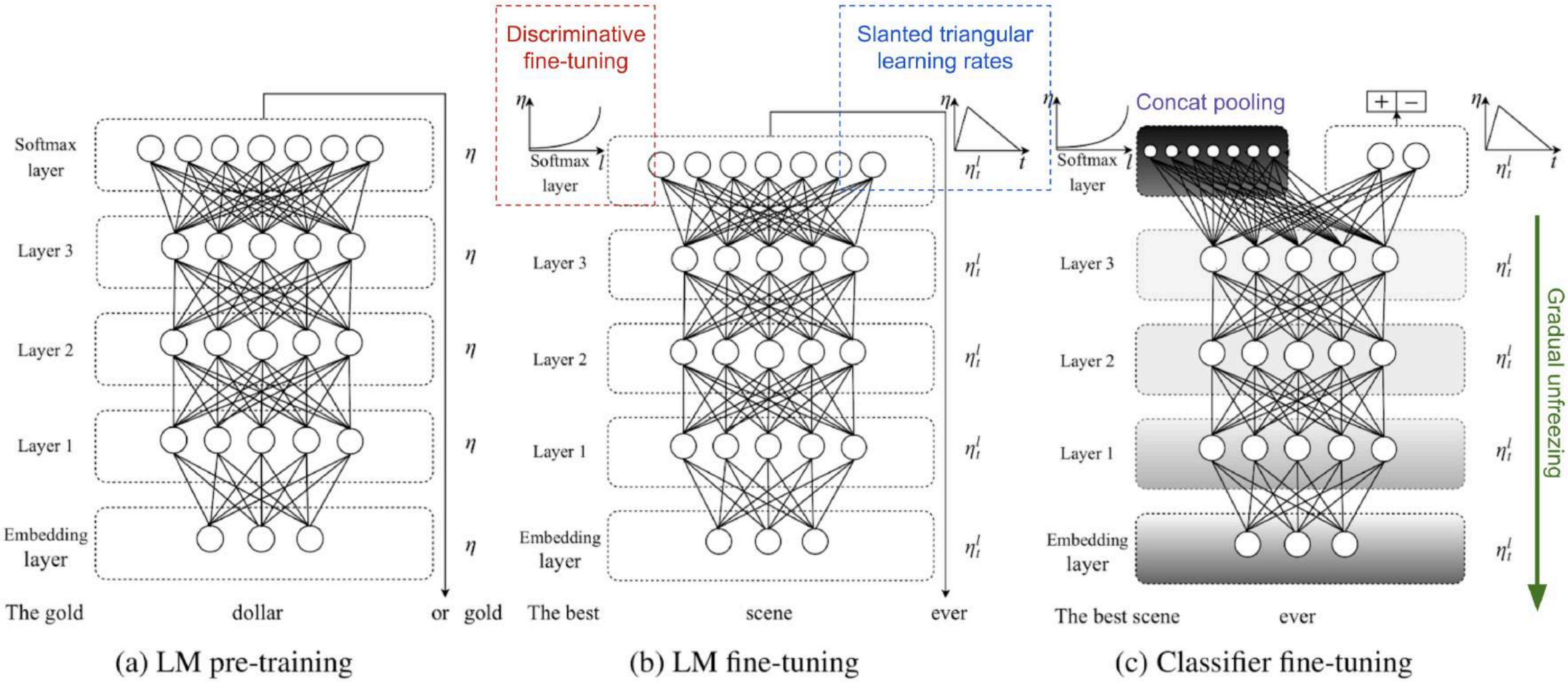
SST : Binary phrase level sentiment classification (Socher et al., 2013).

STS Benchmark : Semantic textual similarity (STS) between sentence pairs scored by Pearson correlation with human judgments (Cer et al., 2017).

Dataset	Train	Dev	Test
SST	67,349	872	1,821
STS Bench	5,749	1,500	1,379
TREC	5,452	-	500
MR	-	-	10,662
CR	-	-	3,775
SUBJ	-	-	10,000
MPQA	-	-	10,606

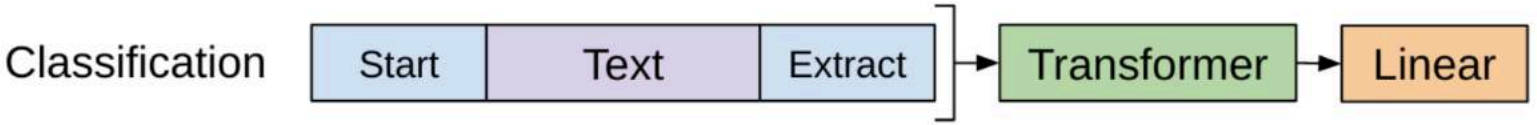
Table 1: Transfer task evaluation sets

ULMFit


Discriminative fine-tuning
Slanted triangular learning rates
Concat pooling
Softmax layer
Layer 3
Layer 2
Layer 1
Embedding layer
The gold
dollar
or
gold
The best
scene
ever
The best scene
ever
Gradual unfreezing
(a) LM pre-training
(b) LM fine-tuning
(c) Classifier fine-tuning

OpenAI GPT

There's no post-training FFN!!!



$$P(y \mid x_1, \ldots, x_n) = \text{softmax}(\mathbf{h}_L^{(n)} \mathbf{W}_y)$$

The loss is to minimize the negative log-likelihood for true labels. In addition, adding the LM loss as an auxiliary loss is found to be beneficial, because:

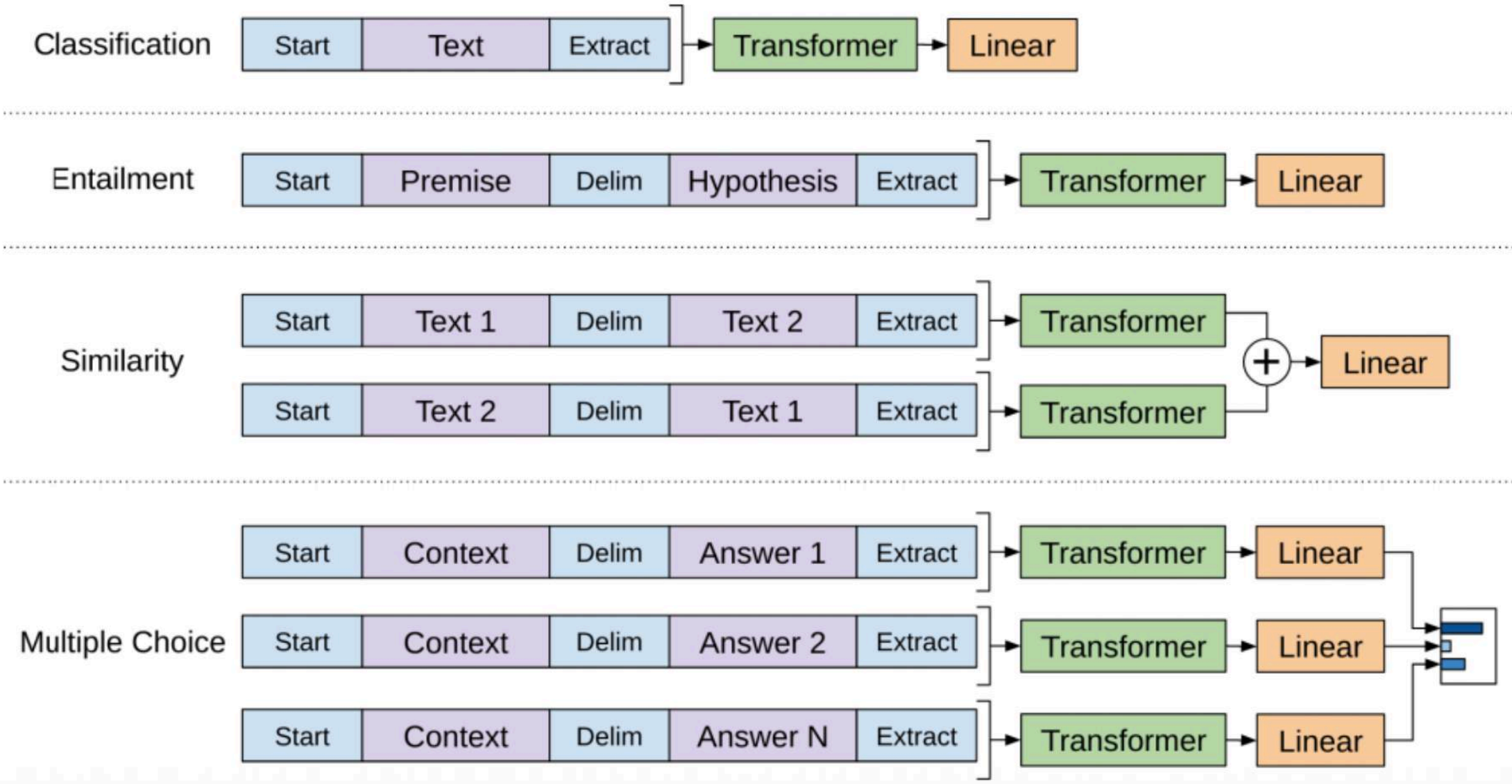
- (1) it helps accelerate convergence during training and
- (2) it is expected to improve the generalization of the supervised model.

$$\mathcal{L}_{\text{cls}} = \sum_{(\mathbf{x},y)\in\mathcal{D}} \log P(y \mid x_1, \ldots, x_n) = \sum_{(\mathbf{x},y)\in\mathcal{D}} \log \text{softmax}(\mathbf{h}_L^{(n)}(\mathbf{x})\mathbf{W}_y)$$

$$\mathcal{L}_{\text{LM}} = -\sum_i \log p(x_i \mid x_{i-k}, \ldots, x_{i-1})$$

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \lambda \mathcal{L}_{\text{LM}}$$

OpenAI GPT


Classification
Start
Text
Extract
Transformer
Linear
Entailment
Start
Premise
Delim
Hypothesis
Extract
Transformer
Linear
Similarity
Start
Text 1
Delim
Text 2
Extract
Start
Text 2
Delim
Text 1
Extract
Transformer
Transformer
Linear
Multiple Choice
Start
Context
Delim
Answer 1
Extract
Start
Context
Delim
Answer 2
Extract
Start
Context
Delim
Answer N
Extract
Transformer
Transformer
Transformer
Linear
Linear
Linear

BERT: Input Hacking

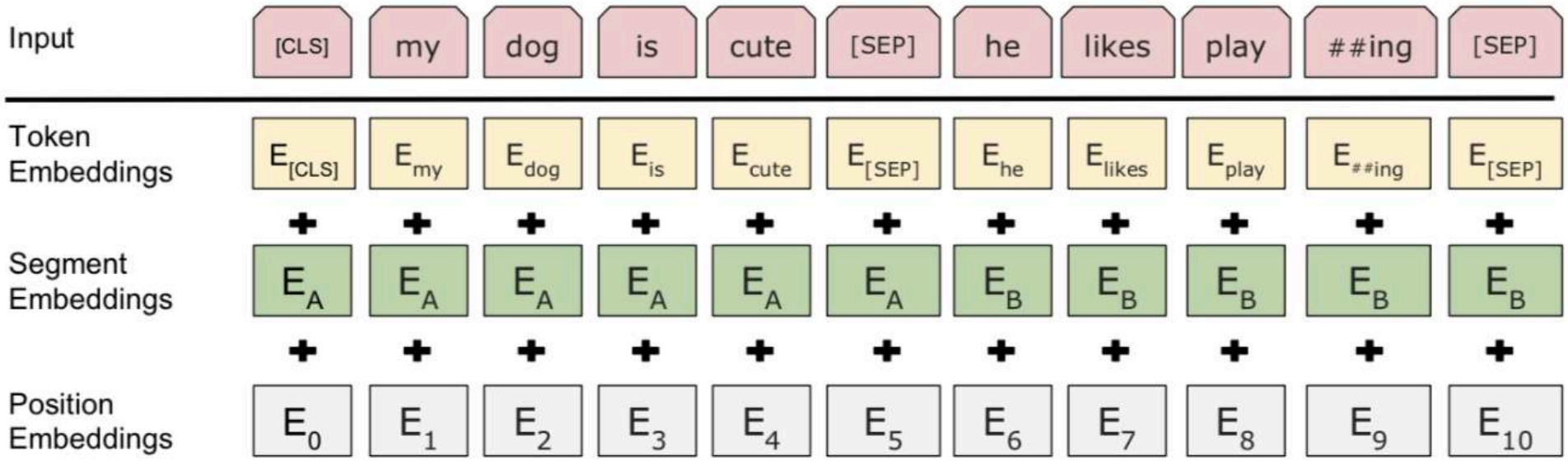
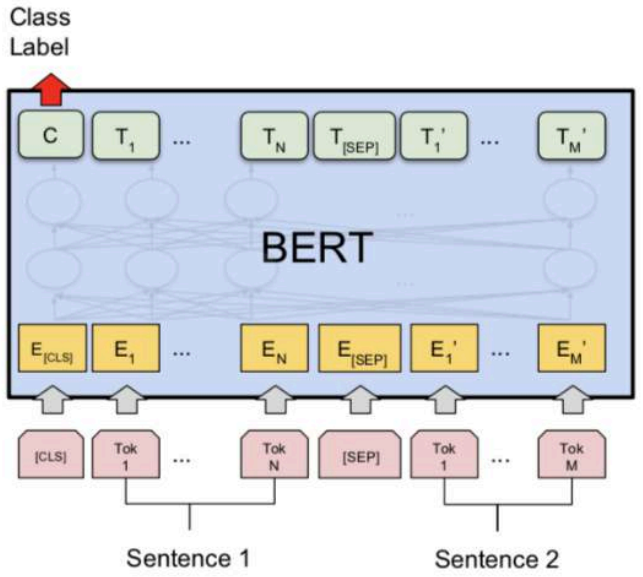


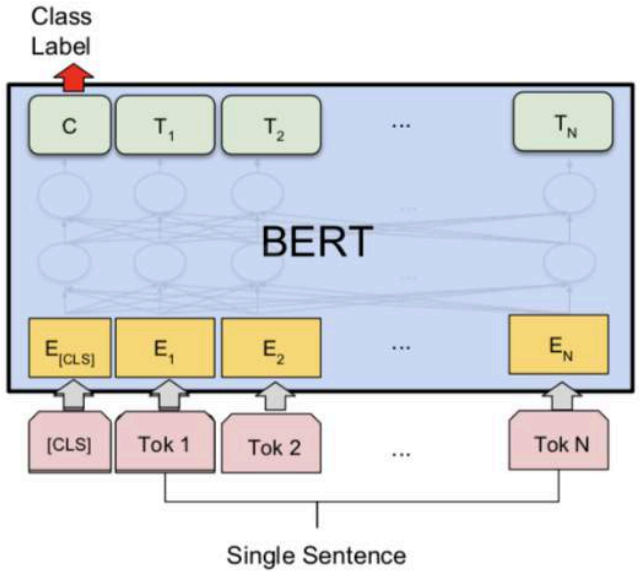
Fig. 11. BERT input representation. (Image source: original paper)

Note that the first token is always forced to be `[CLS]` — a placeholder that will be used later for prediction in downstream tasks.

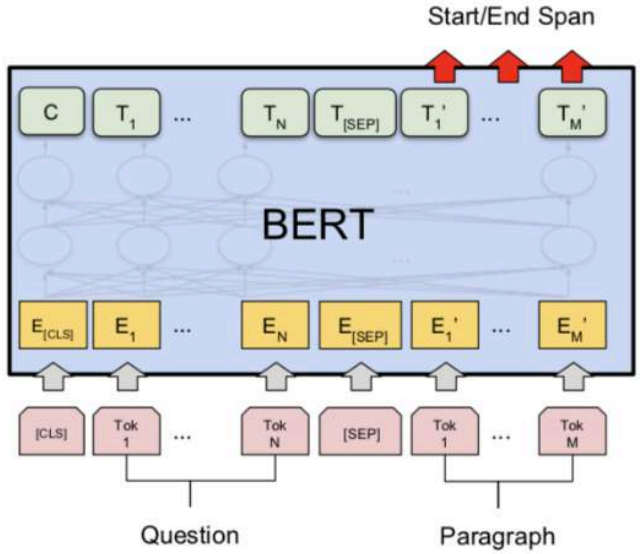
BERT



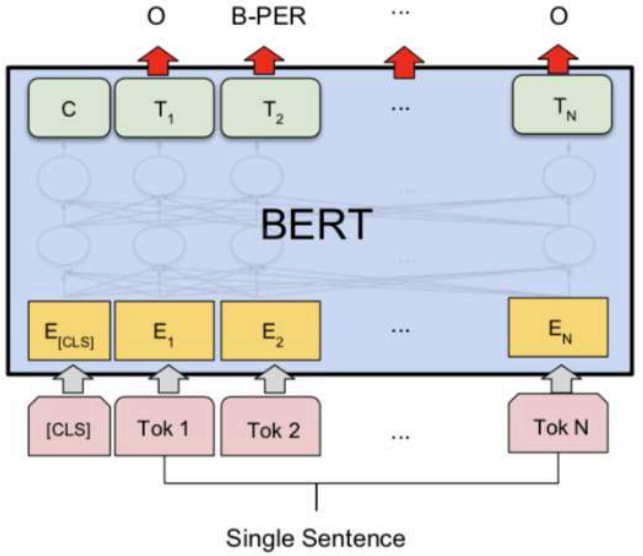
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA

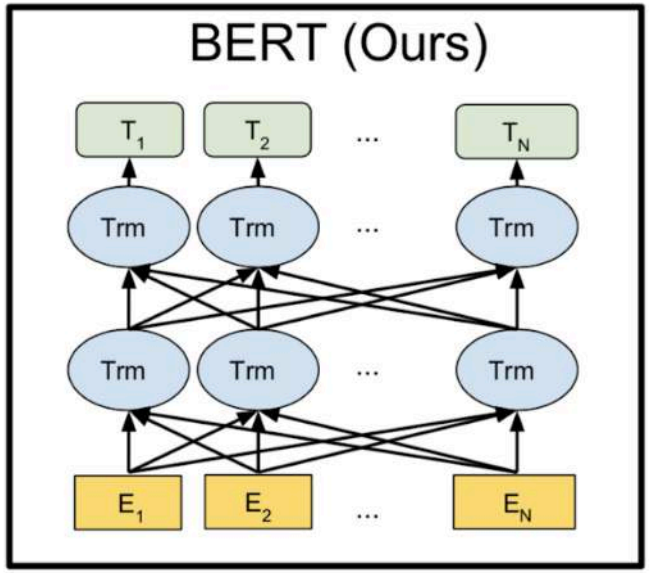


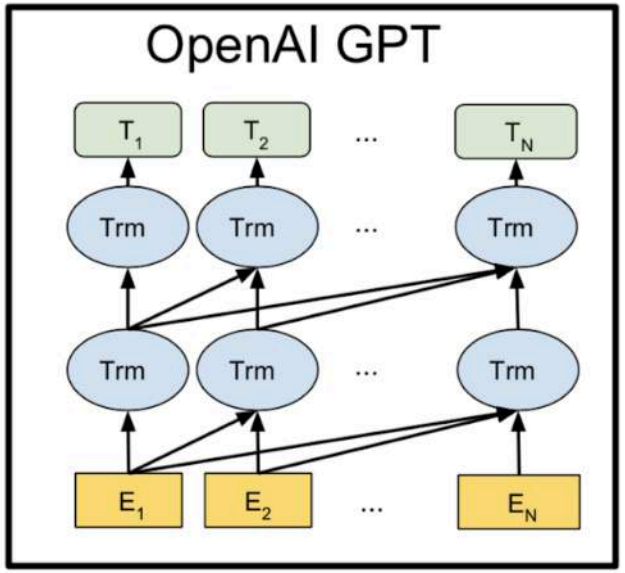
(c) Question Answering Tasks: SQuAD v1.1

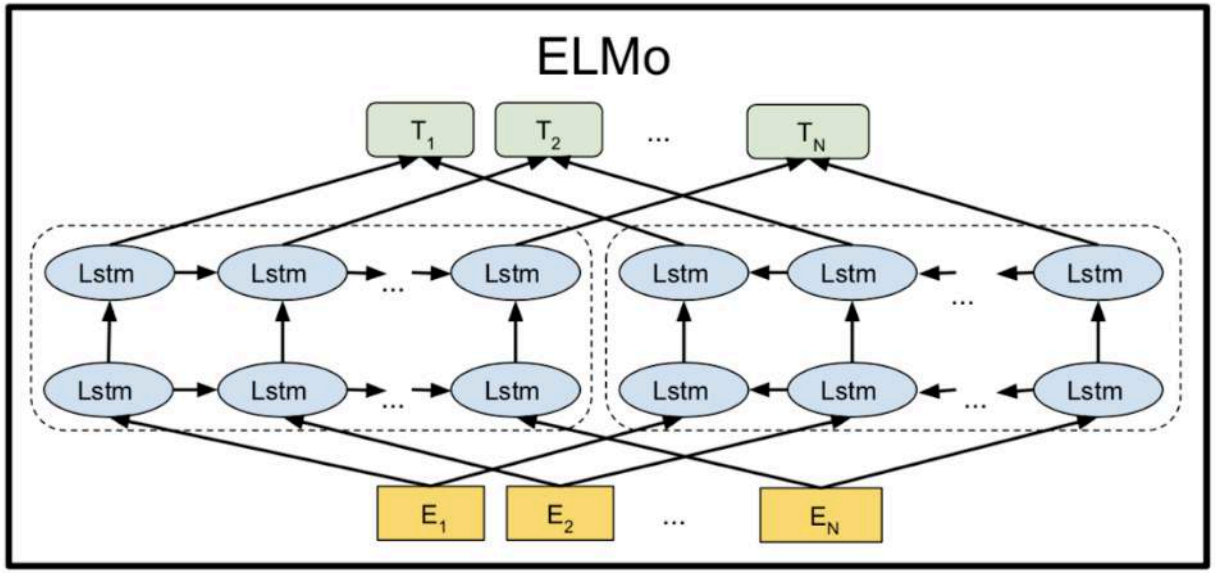


(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

BERT vs OpenAI GPT vs ELMo







What was this course about?

- **Recent advance techniques for NLP**
  - esp. since the "*deep learning tsunami*"
- Getting a **good grasp of PyTorch for NLP**
  - Similar knowledge can be applied to any tensor libraries for machine/deep learning
- **Understanding the underlying techniques and knowing how to implement them**
  - Hopefully, you'll find them less "magical"

What was this course NOT about?

- **Understanding *everything* in NLP.**
  - There's a lot more to *computational linguistics* than what's in this course
- Using **libraries to achieve results for specific NLP tasks**
  - (e.g. SpaCy, AllenNLP, TorchText, etc.)
- No, **we won't build chatbots** in this course.

Thank you for pardoning my quirks…

- **Sometimes I let the code do the talking and I just read out the code** (*Stop me if it's not intuitive enough*)

- **I have more code than math in my slides**

- **Feel free to stop me at any time to ask questions**

Fin