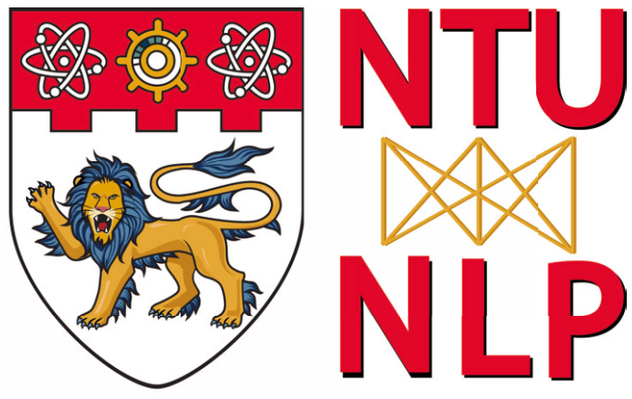
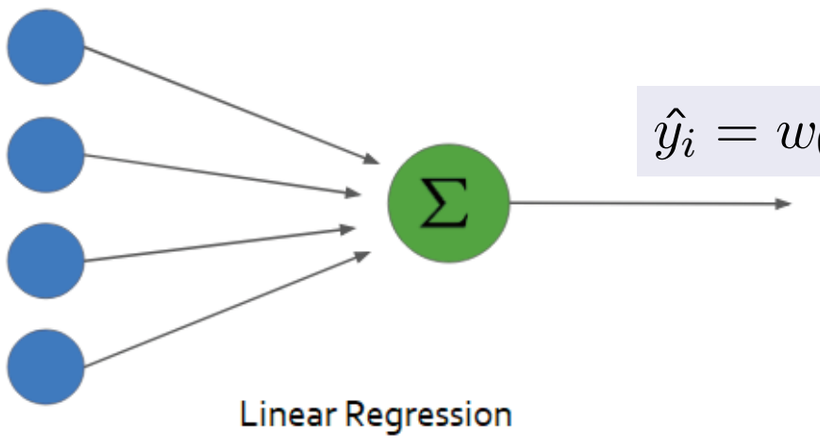
Deep Learning for Natural Language Processing

Shafiq Joty



Lecture 3: Neural Networks & Optimisation Basics

Linear Models (Recap)



Linear Regression

$$\hat{y}_i = w_0 + w_1 x_{i1} + ... + w_d x_{id}$$

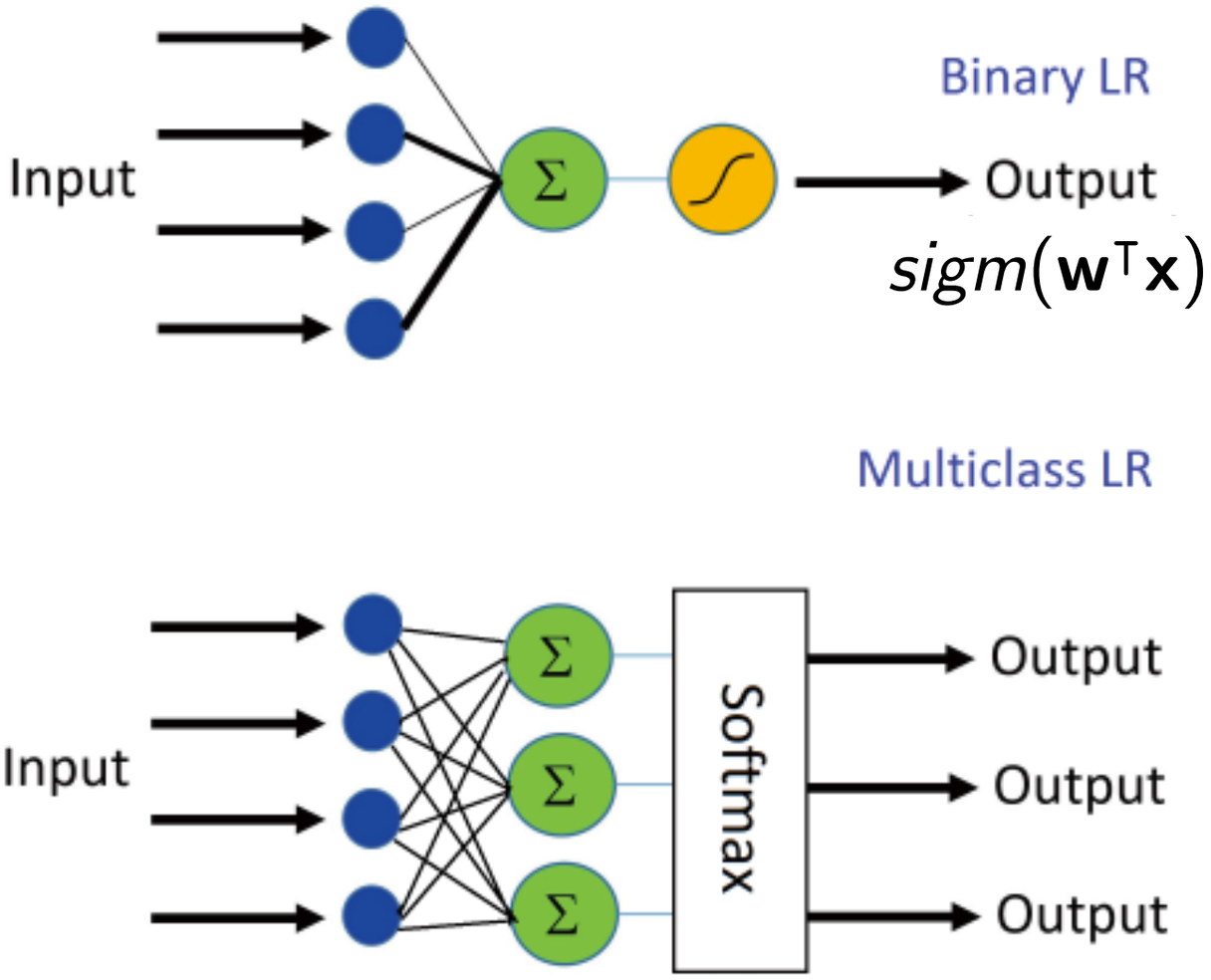
MSE loss

$$RSS(\mathbf{w}) \triangleq \sum_{i=1}^{N} (y_i - \mathbf{w}^\intercal \mathbf{x}_i)^2$$

Closed form solution

$$\mathbf{w} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal y$$

Linear Models (Recap)



Softmax $$\frac{\exp(\mathbf{w}_c^\mathsf{T}\mathbf{x})}{\sum_{c'=1}^{C}\exp(\mathbf{w}_{c'}^\mathsf{T}\mathbf{x})}$$

SGD

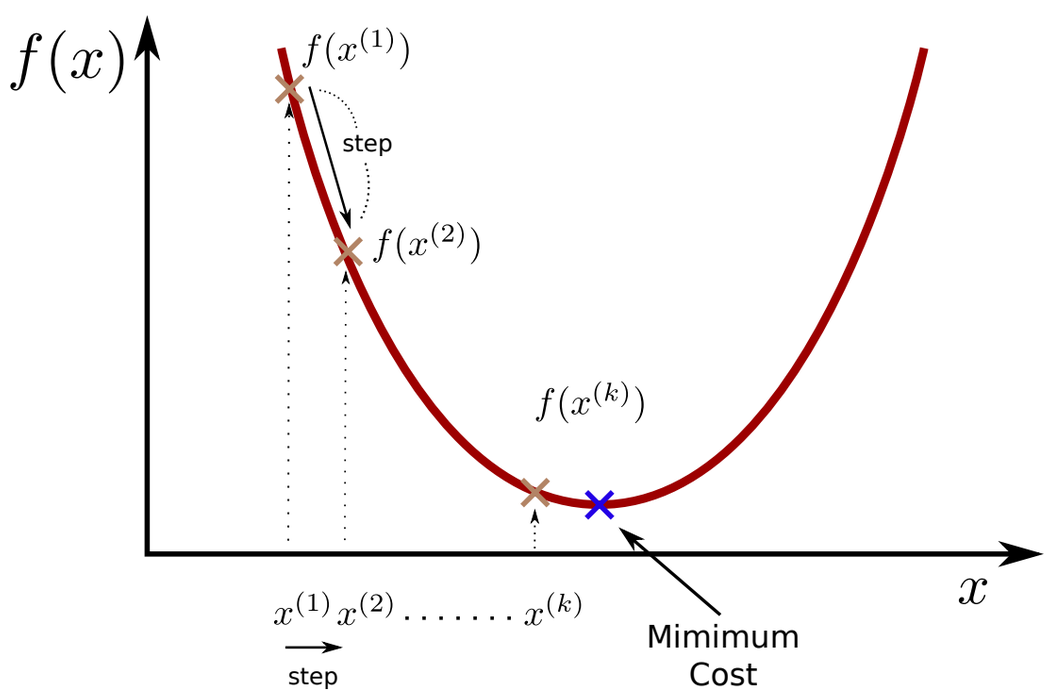


Figure: Illustration of gradient descent

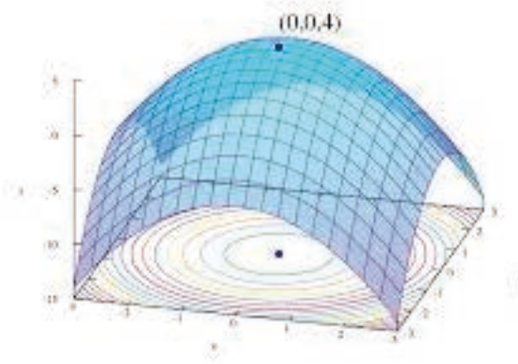
$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$$

Traditional Machine Learning

- Most ML models work well because of **human-designed representations/input features.**
- Machine learning becomes just optimizing feature weights to make a good prediction.

Feature	NER
Current Word	✓
Previous Word	✓
Next Word	✓
Current Word Character n-gram	all
Current POS Tag	✓
Surrounding POS Tag Sequence	✓
Current Word Shape	✓
Surrounding Word Shape Sequence	✓
Presence of Word in Left Window	size 4
Presence of Word in Right Window	size 4

Features for NER (Finkel et al., 2010)



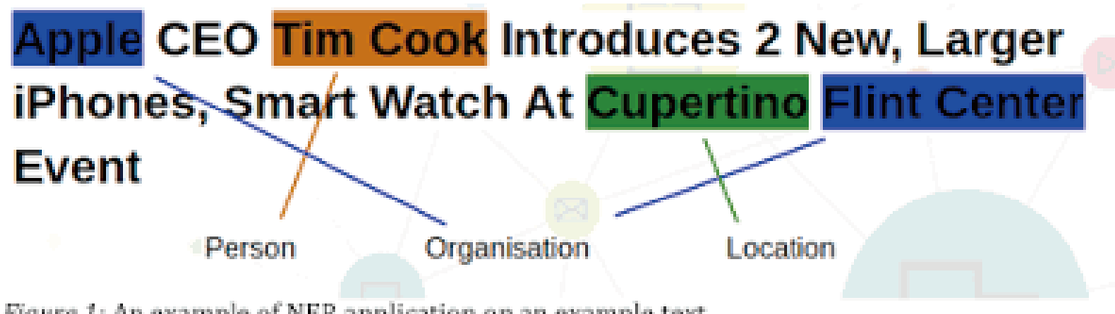
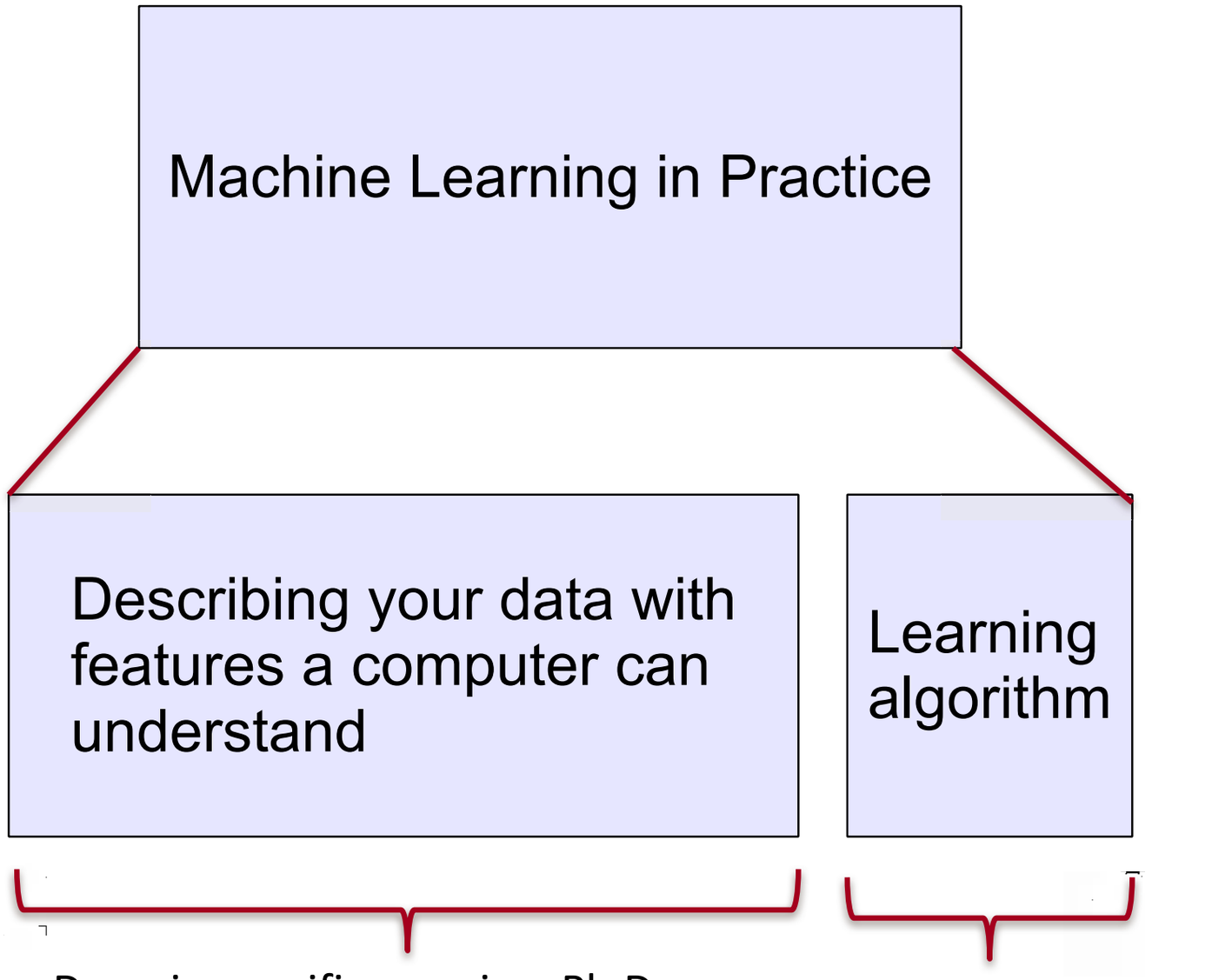


Figure 1: An example of NER application on an example text

Traditional Machine Learning



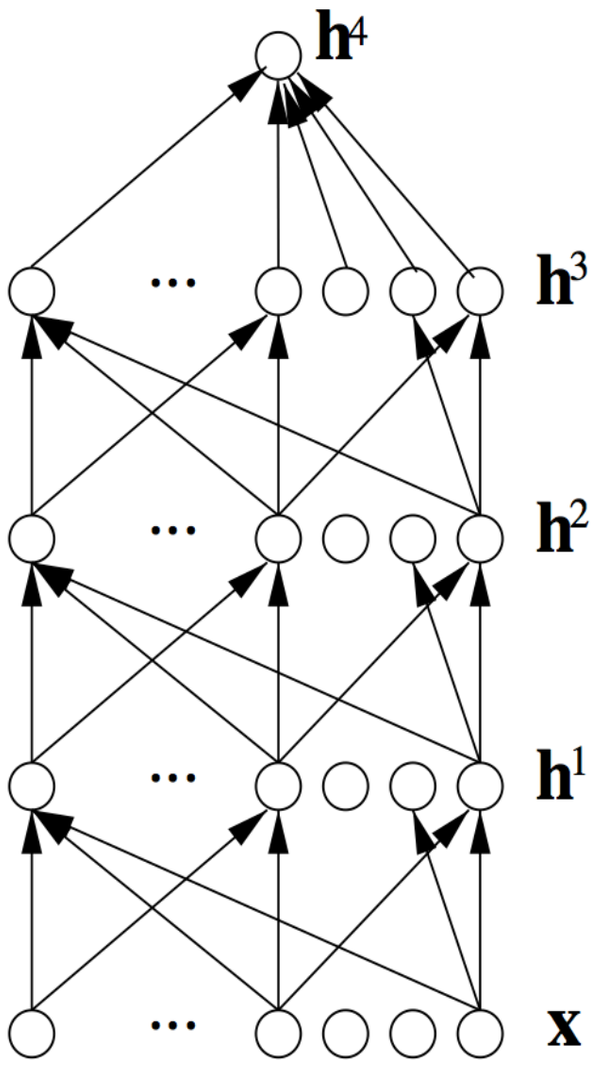
Deep Learning

NTU NLP

- Representation learning attempts to automatically learn good features or representations

- Deep learning algorithms attempt to learn (multiple levels of) representations (here: h^1,h^2,h^3) and an output (h^4)

- From "raw" inputs **x** (e.g. sound, pixels, characters, or words)



Source: Stanford 224n

Why Deep Learning?

NTU NLP

#1. Flexible R&R System

- Manually designed features are often over-specified, incomplete and take a long time to design and validate
- This has to be done again for each task/domain/…
- **Learned Features** are easy to adapt, fast to learn
- Deep learning provides a very flexible, learnable framework for **representing the** world (e.g., visual and linguistic information), and perform **reasoning** with that representation.
- Deep learning can learn in **unsupervised** (from raw text) and **supervised** (with specific labels like positive/negative) settings

Source: Socher & Manning

Why Deep Learning?

NTU NLP

#2. Need for distributed representations

- Current NLP systems are incredibly fragile because of their atomic symbol representation.

- Deep learning uses **distributed representations** of words, phrases, and concepts that generalise better

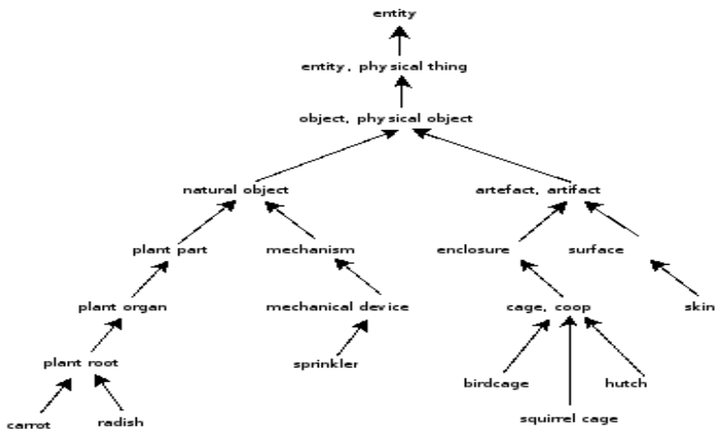
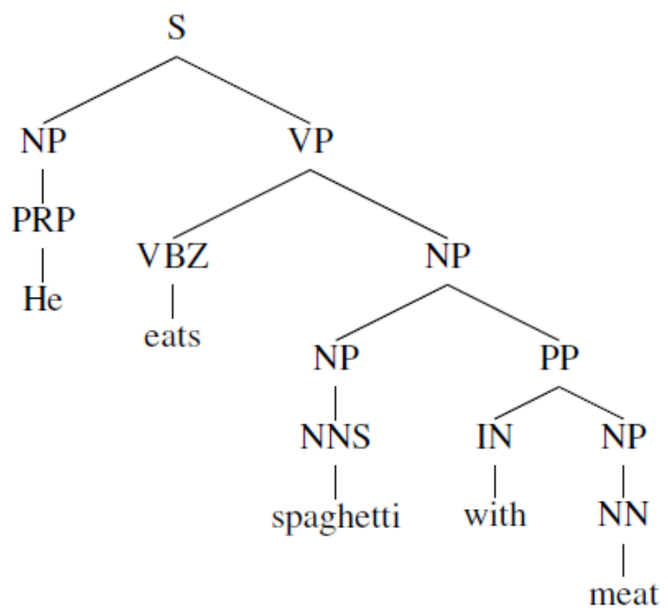
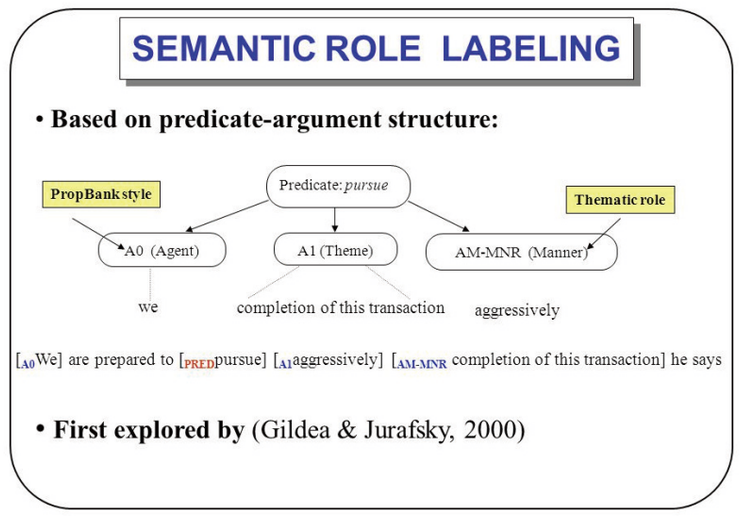


Figure 1. "is-a" relation example

WordNet



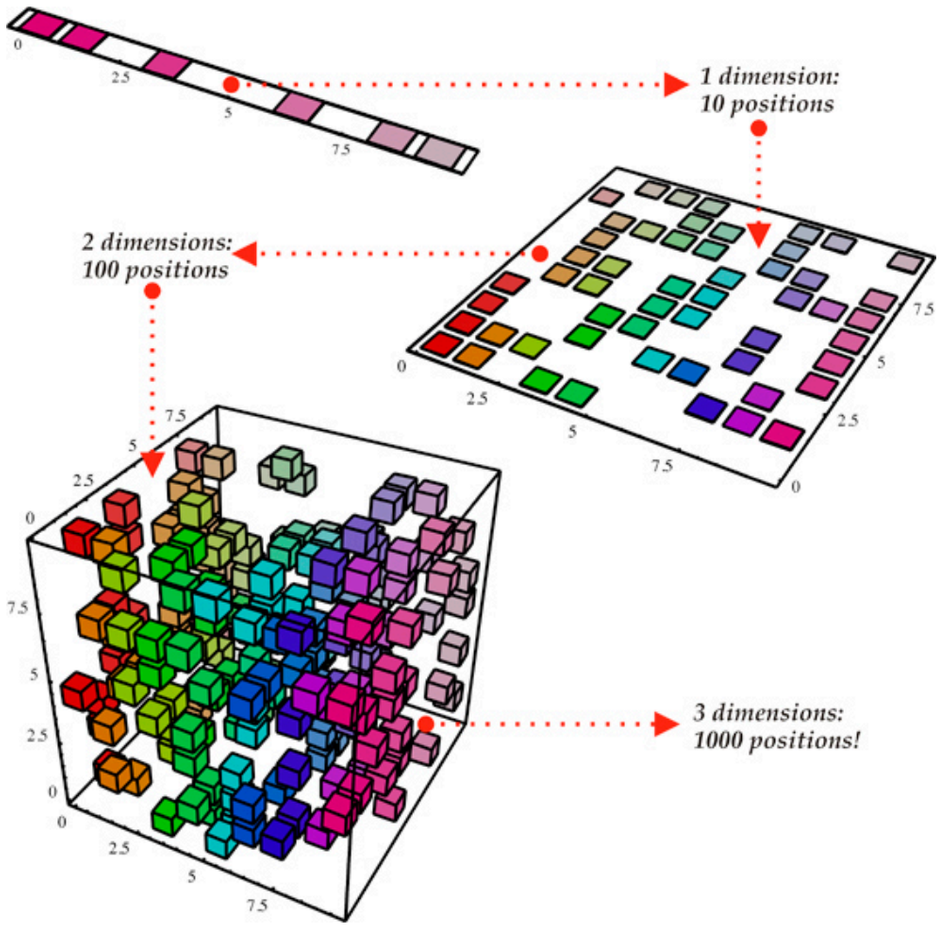


Why Deep Learning?

NTU NLP

#2. Need for distributed representations

- Generalizing locally (e.g., nearest neighbors (KNN)) requires representative examples for all relevant variations.
- Curse of dimensionality



Classic solutions:

- Manual feature design
- Assuming a smooth target function (e.g., linear models)
- Kernel methods (linear in terms of kernel based on data points)

- Neural networks parameterize and learn a "similarity" kernel

Source: Socher & Manning

Why Deep Learning?

NTU NLP

#3. Need for Unsupervised (or self-supervised) learning

- Most practical, good traditional ML methods require labeled training data (i.e., supervised learning).

- However, most information must be acquired **unsupervised**

When we're learning to see, nobody's telling us what the right answers are — we just look. Every so often, your mother says "that's a dog", but that's very little information. You'd be lucky if you got a few bits of information — even one bit per second — that way. The brain's visual system has 10^{14} neural connections. And you only live for 10^9 seconds. So it's no use learning one bit per second. You need more like 10^5 bits per second. And there's only one place you can get that much information: from the input itself. — Geoffrey Hinton, 1996 (quoted in (Gorder 2006)).

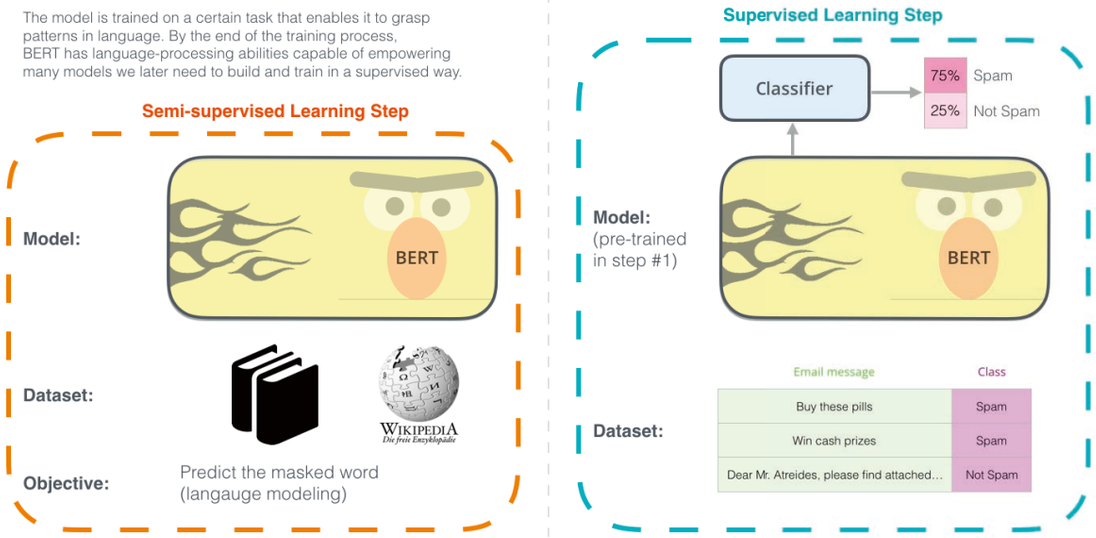
Source: Socher & Manning

Why Deep Learning?

NTU NLP

#3. Need for Unsupervised (or self-supervised) learning

- Models like BERT, XLM, ROBERTA, VideoBERT, VL-BERT are changing the entire landscape of NLP (and AI in general)



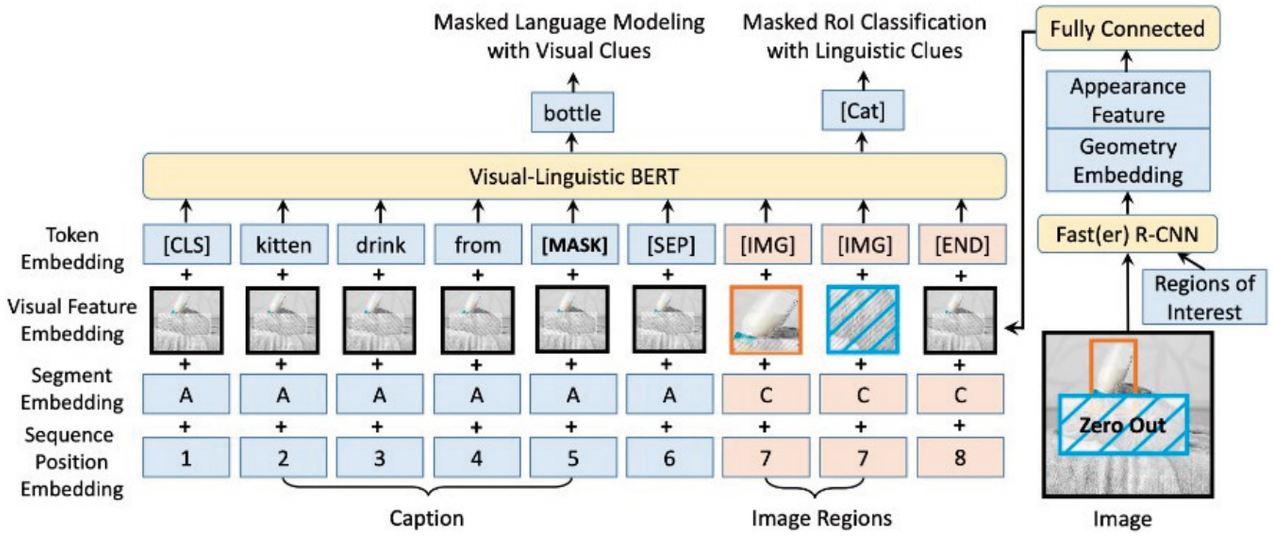


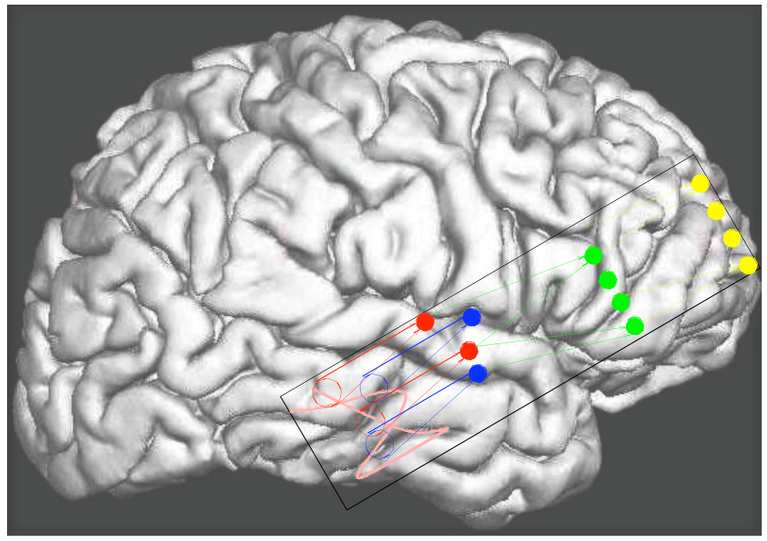
Figure 1. Architecture for Pre-training VL-BERT

Why Deep Learning?

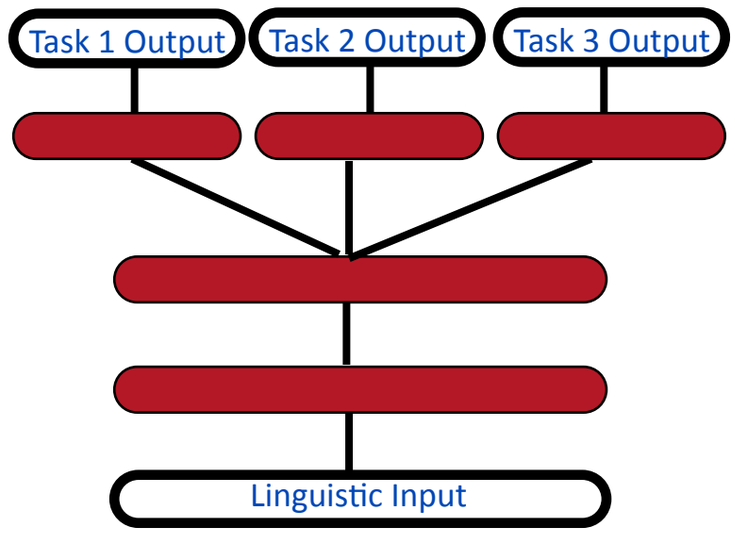
NTU NLP

#4. Learning multiple levels of representation

- The cortex seems to have a generic learning algorithm
- The brain has a deep architecture



- We need good intermediate representations that can be shared across tasks
- Multiple levels of latent variables allow combinatorial sharing of statistical strength
- Insufficient model depth can be exponentially inefficient



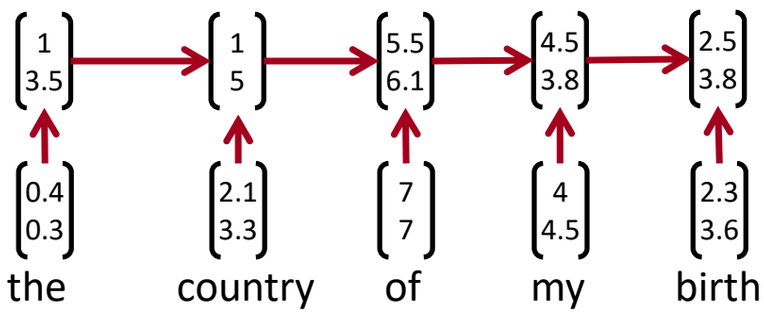
Source: Socher & Manning

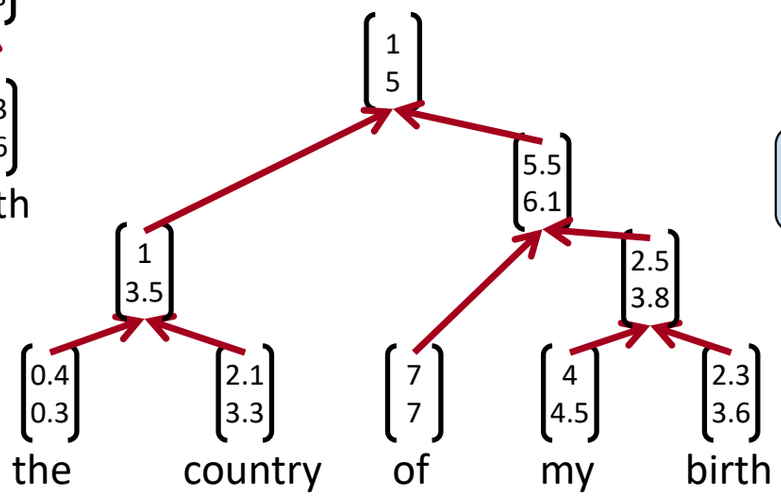
Why Deep Learning?

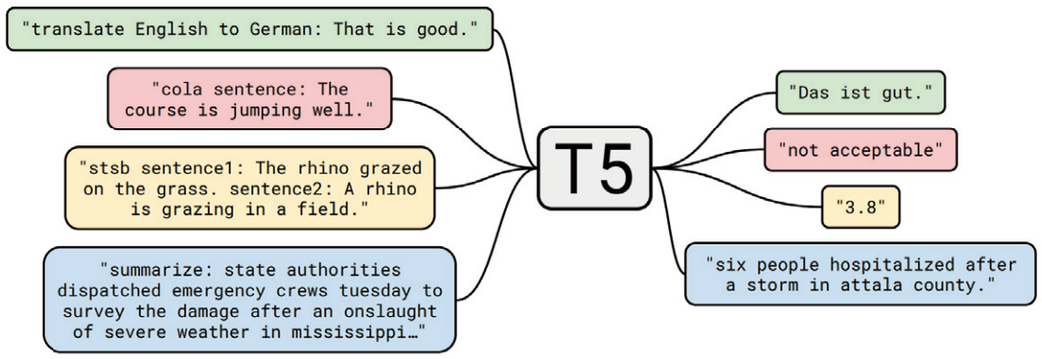
NTU NLP

#4. Learning multiple levels of representation

- We need good intermediate representations that can be shared across tasks
- We need compositionality in our ML models







Why Deep Learning?

NTU NLP

#5. Why now?

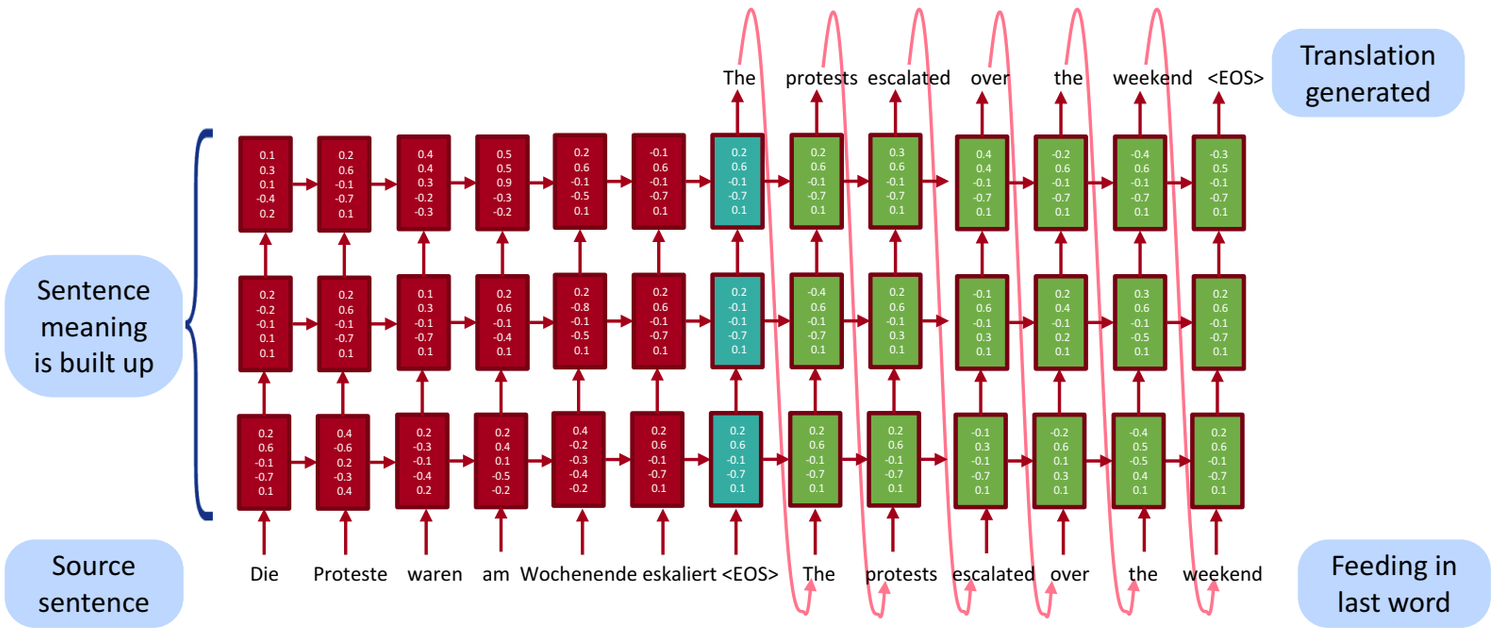
- In ~2010 **deep** learning techniques started outperforming other machine learning techniques. Why this decade?
  - Large amounts of training data favor deep learning
  - Faster machines and multicore CPU/GPUs favor deep learning
- New models, algorithms, ideas
  - Better, more flexible learning of intermediate representations
  - Effective end-to-end joint system learning
  - Effective learning methods for using contexts and transferring between tasks
  - Better regularization and optimization methods
- **Improved performance** (first in speech and vision, then NLP)

Source: Stanford 224n

Deep Learning in Language

NTU NLP

Machine Translation: first breakthrough in 2015



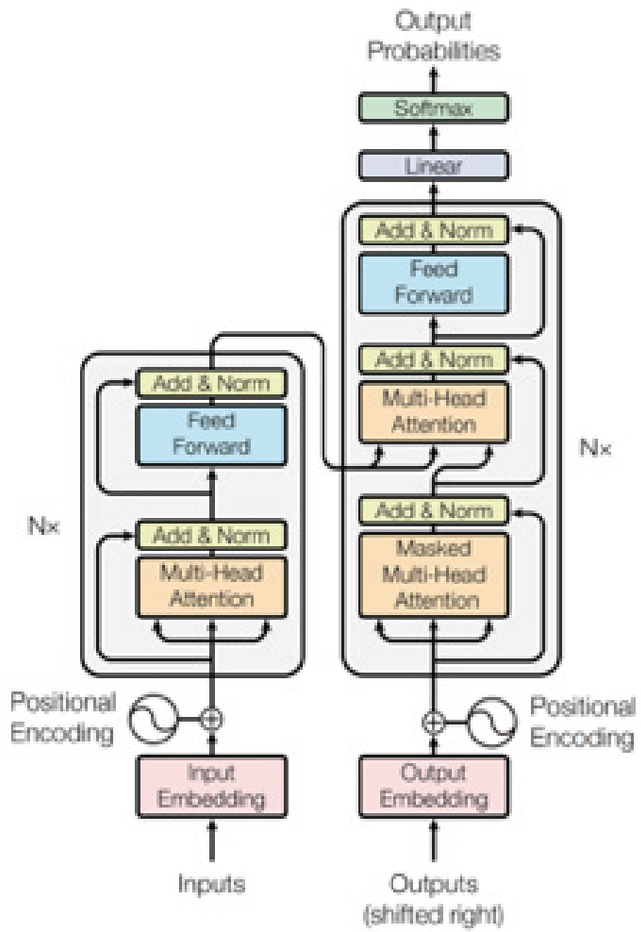


Figure 1: The Transformer - model architecture.

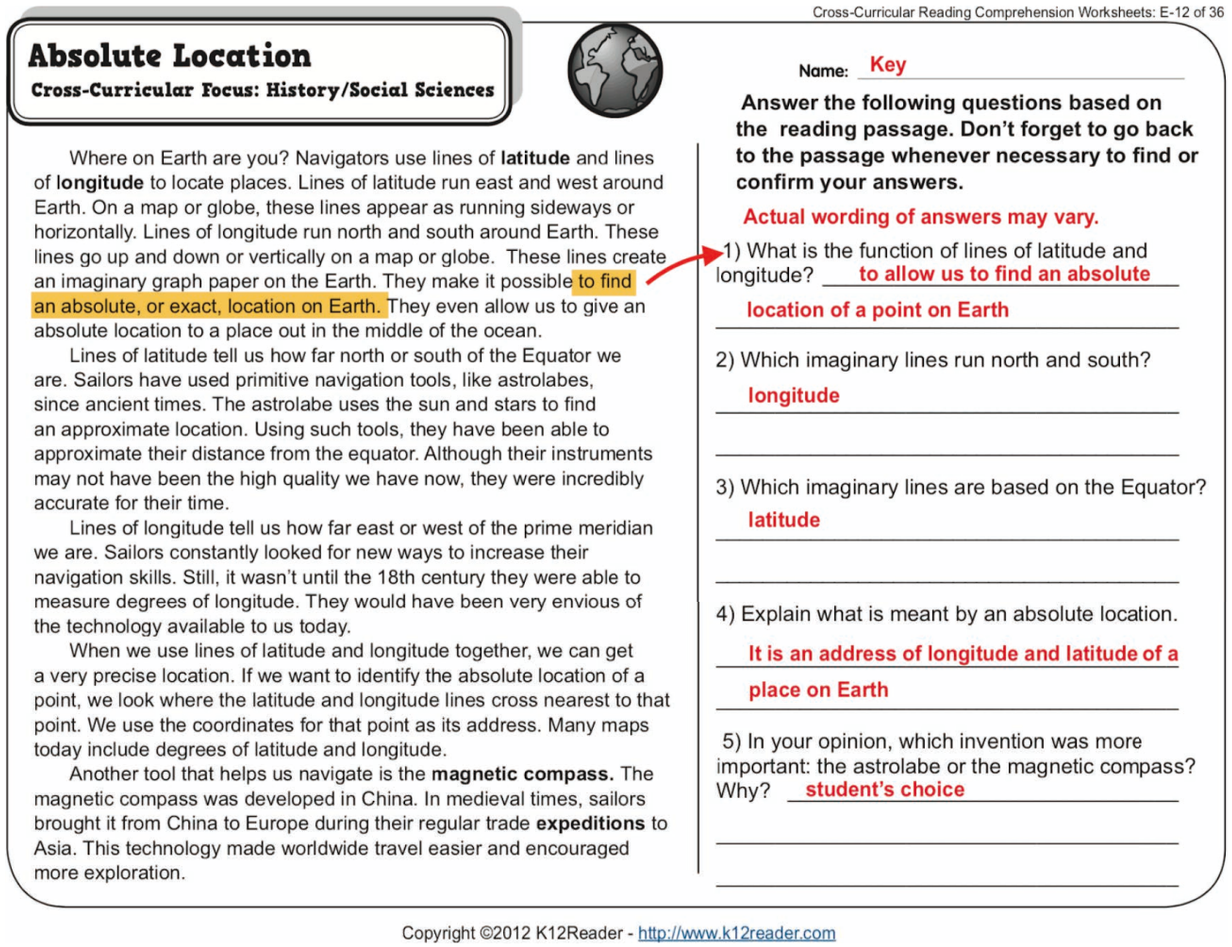
Now live for many languages in Google

Translate (etc.), with big error reductions!

Deep Learning in Language

NTU NLP

Machine Reading Comprehension (QA)



Cross-Curricular Reading Comprehension Worksheets: E-12 of 36

Absolute Location

Cross-Curricular Focus: History/Social Sciences

Where on Earth are you? Navigators use lines of **latitude** and lines of **longitude** to locate places. Lines of latitude run east and west around Earth. On a map or globe, these lines appear as running sideways or horizontally. Lines of longitude run north and south around Earth. These lines go up and down or vertically on a map or globe. These lines create an imaginary graph paper on the Earth. They make it possible to find an absolute, or exact, location on Earth. They even allow us to give an absolute location to a place out in the middle of the ocean.

Lines of latitude tell us how far north or south of the Equator we are. Sailors have used primitive navigation tools, like astrolabes, since ancient times. The astrolabe uses the sun and stars to find an approximate location. Using such tools, they have been able to approximate their distance from the equator. Although their instruments may not have been the high quality we have now, they were incredibly accurate for their time.

Lines of longitude tell us how far east or west of the prime meridian we are. Sailors constantly looked for new ways to increase their navigation skills. Still, it wasn't until the 18th century they were able to measure degrees of longitude. They would have been very envious of the technology available to us today.

When we use lines of latitude and longitude together, we can get a very precise location. If we want to identify the absolute location of a point, we look where the latitude and longitude lines cross nearest to that point. We use the coordinates for that point as its address. Many maps today include degrees of latitude and longitude.

Another tool that helps us navigate is the **magnetic compass.** The magnetic compass was developed in China. In medieval times, sailors brought it from China to Europe during their regular trade **expeditions** to Asia. This technology made worldwide travel easier and encouraged more exploration.

Name: Key

Answer the following questions based on the reading passage. Don't forget to go back to the passage whenever necessary to find or confirm your answers.

Actual wording of answers may vary.

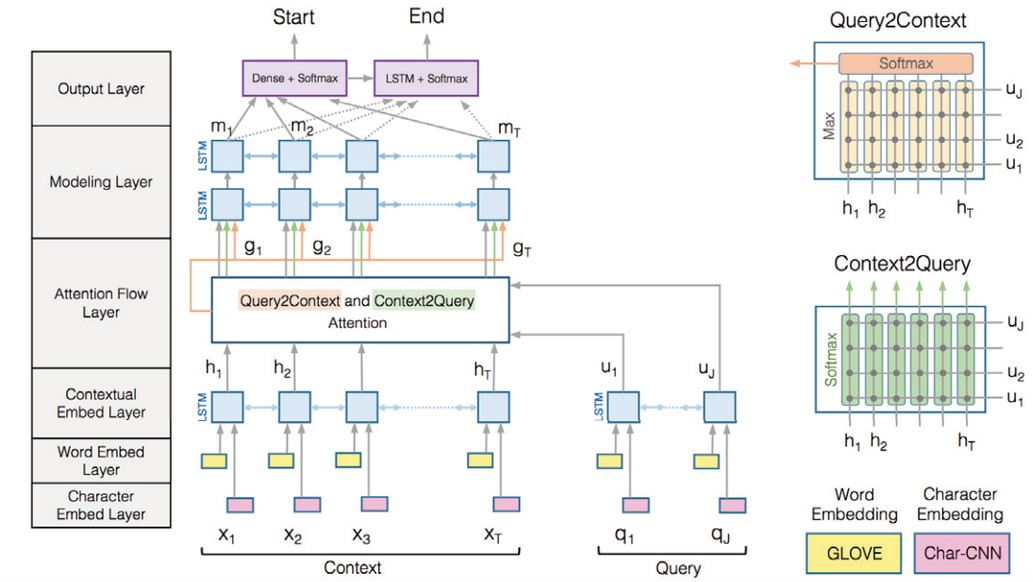
1) What is the function of lines of latitude and longitude? to allow us to find an absolute location of a point on Earth

2) Which imaginary lines run north and south? longitude

3) Which imaginary lines are based on the Equator? latitude

4) Explain what is meant by an absolute location. It is an address of longitude and latitude of a place on Earth

5) In your opinion, which invention was more important: the astrolabe or the magnetic compass? Why? student's choice



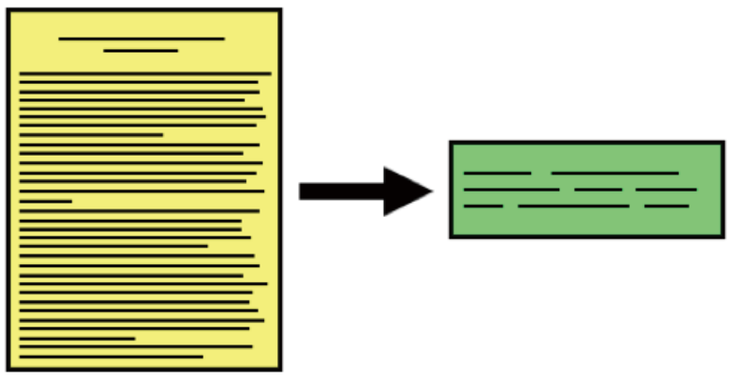
Allen AI's BiDAF

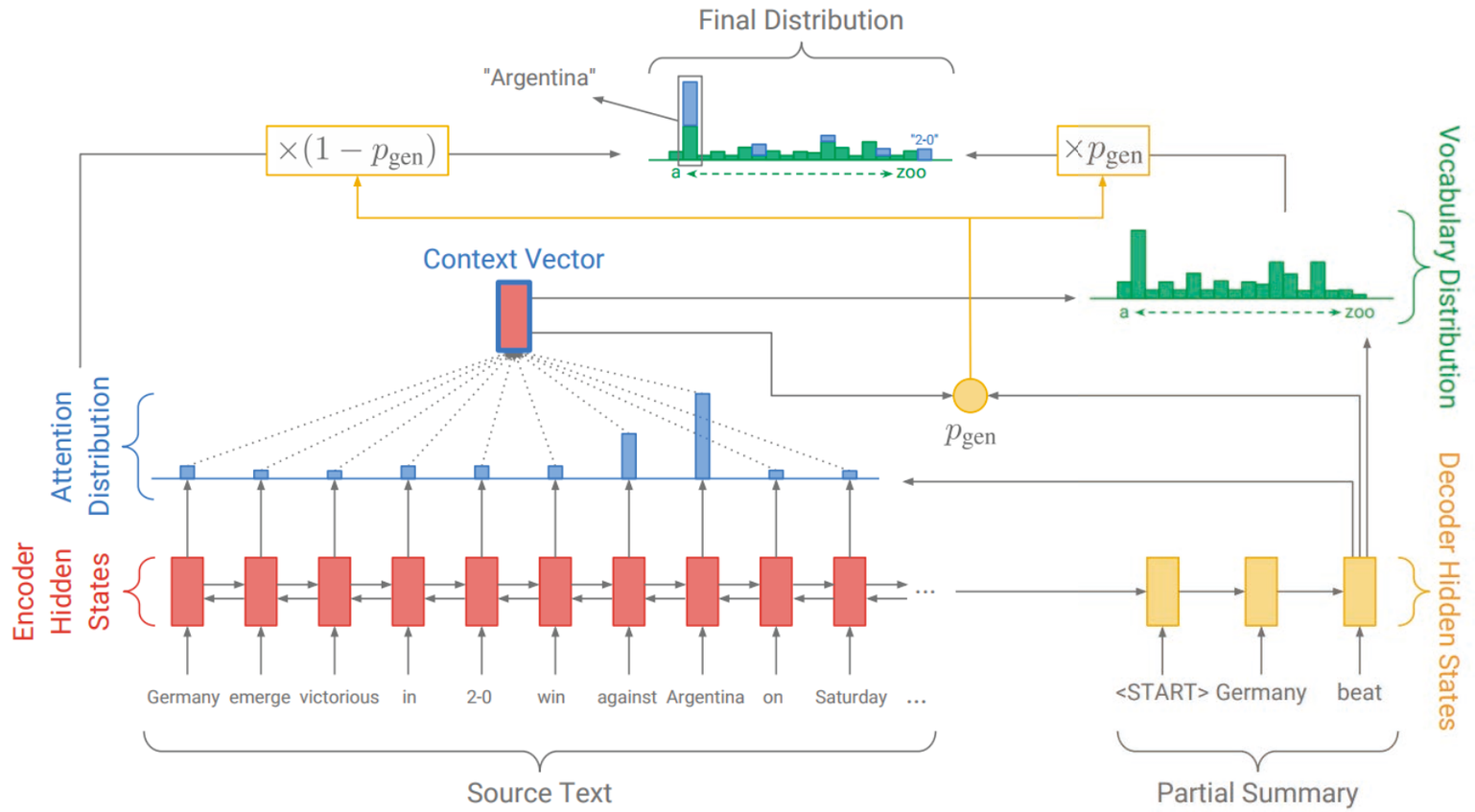
Source: Allen AI

Deep Learning in Language

NTU NLP

Text Summarization





Pointer Generator

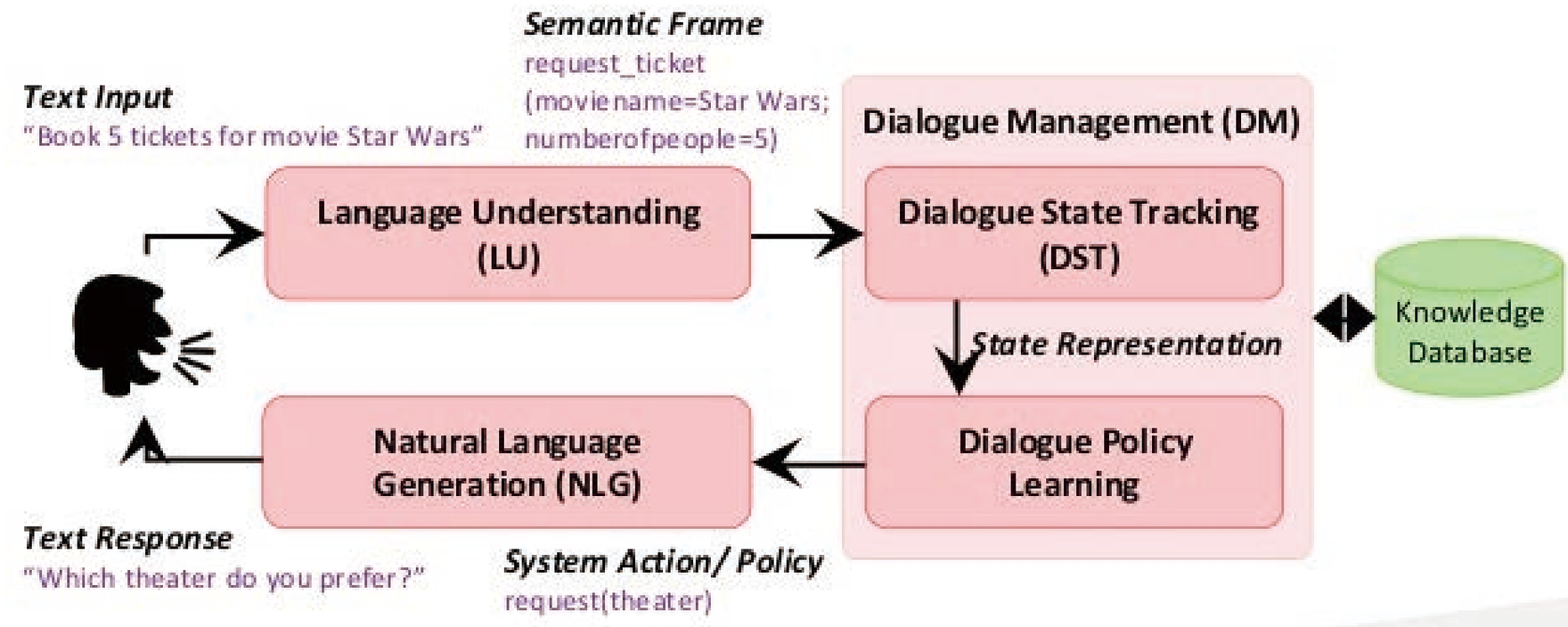
Source: See et al (2017)

Deep Learning in Language

NTU NLP

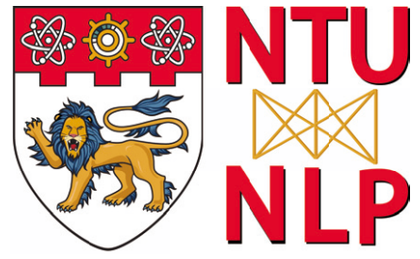
Dialogue Systems

9 Task-Oriented Dialogue System Framework



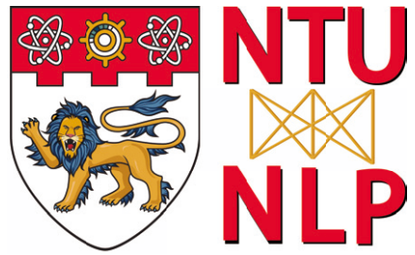
Source: MSRA

Lecture Plan



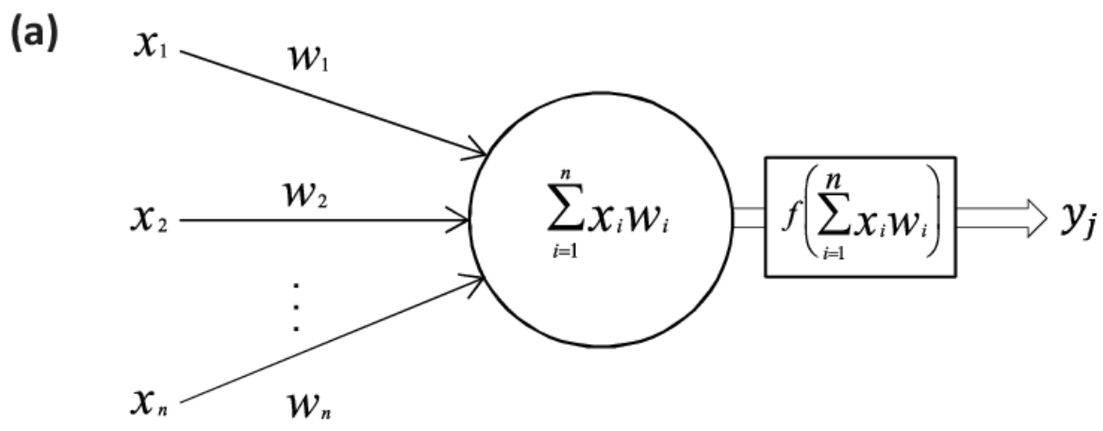
- Why Deep Learning for NLP
- From Logistic/Linear Regression to NN
  - Activation functions
- SGD with Backpropagation
- Adaptive SGD (adagrad, adam, RMSProp)
- Regularisation (dropout, gradient clipping)
- Introduction to word vectors

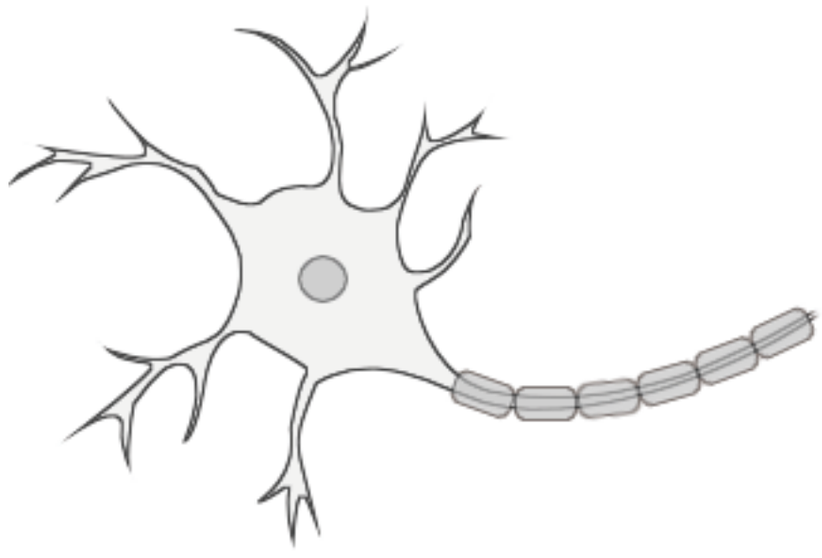
Neurons

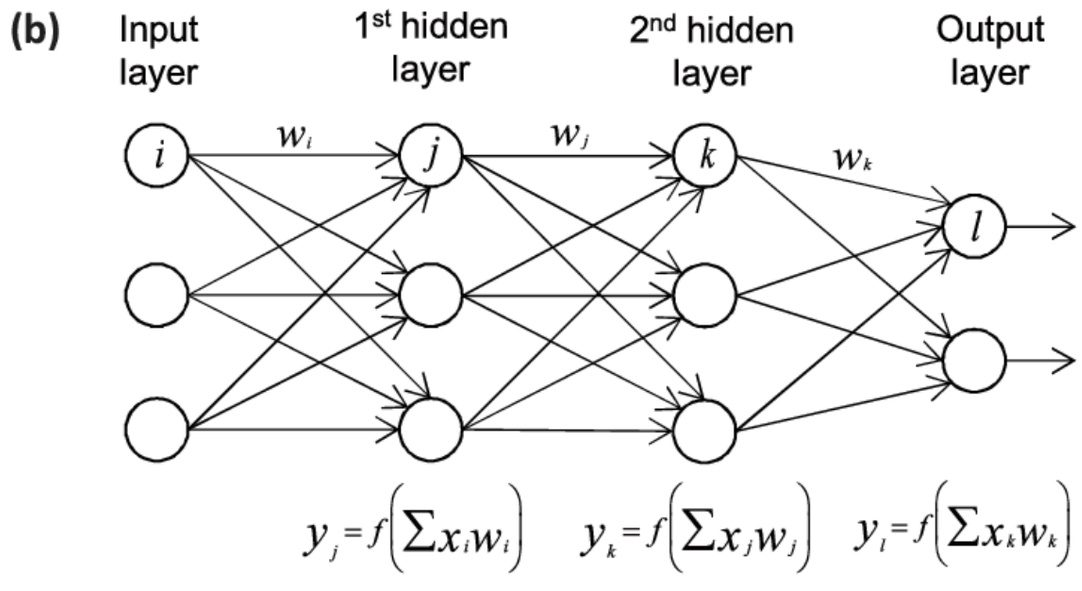


if you understand how logistic regression works

Then **you already understand** the operation of a basic neural network neuron!







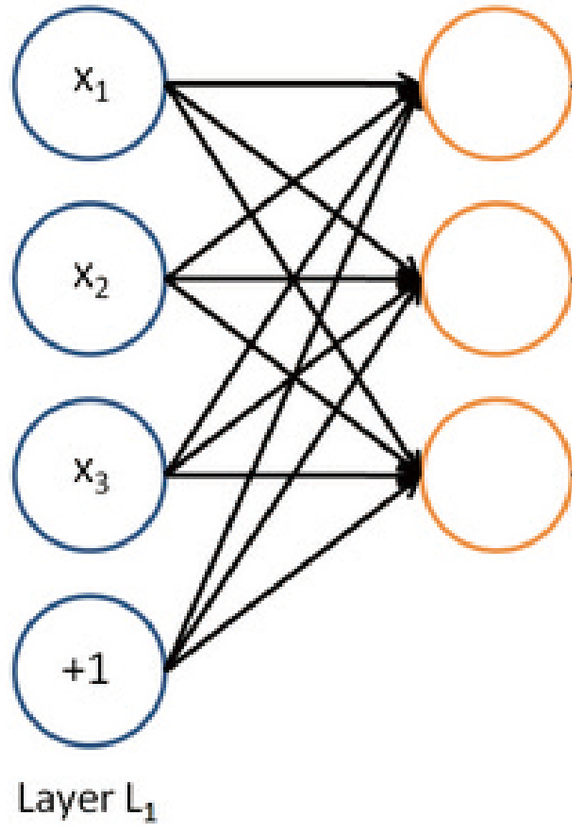
From LR to NN

NTU NLP

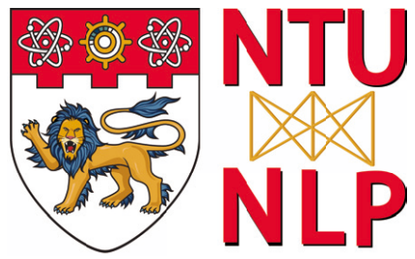
A neural network = running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …

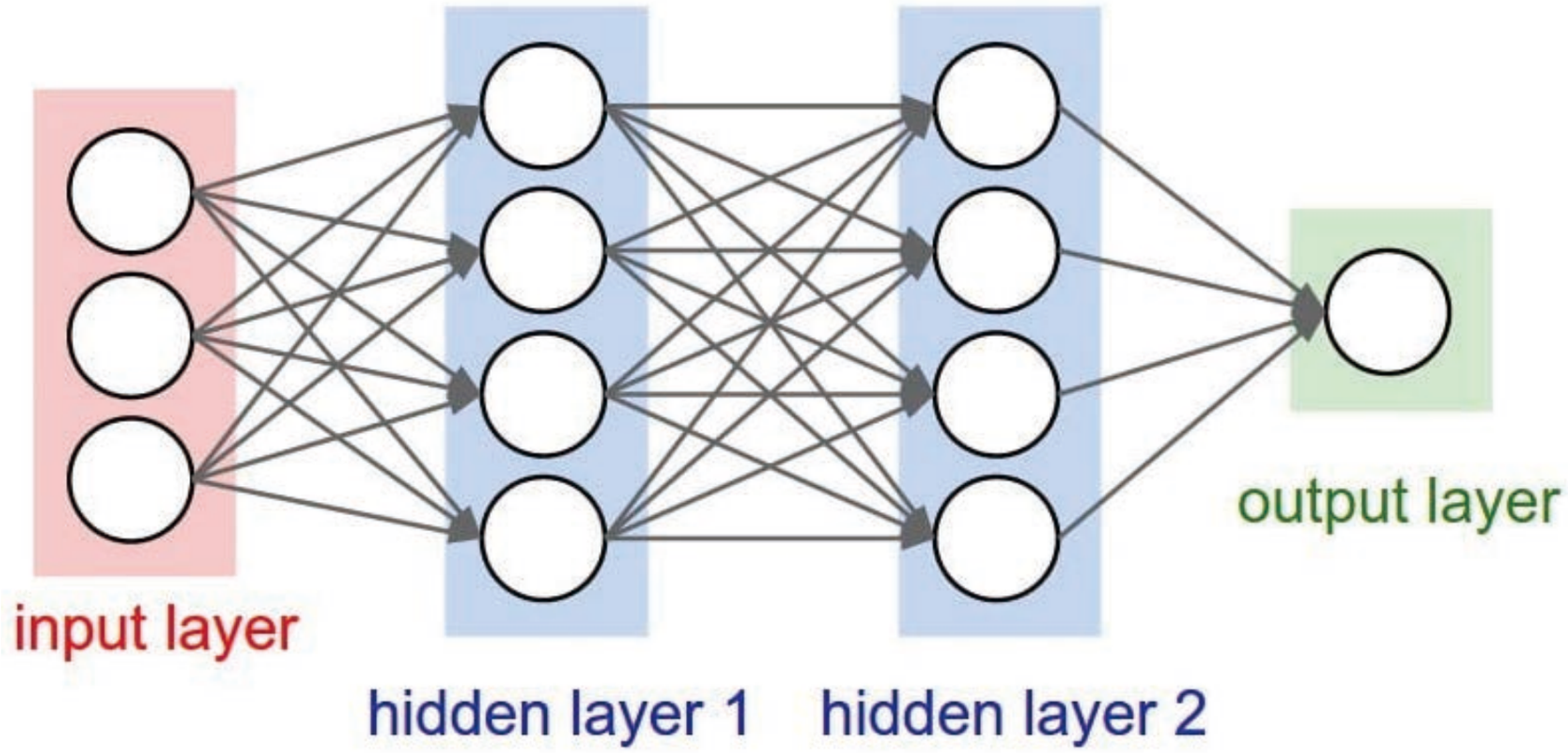
But these are not random variables



From LR to NN



... which we can feed into another logistic regression function



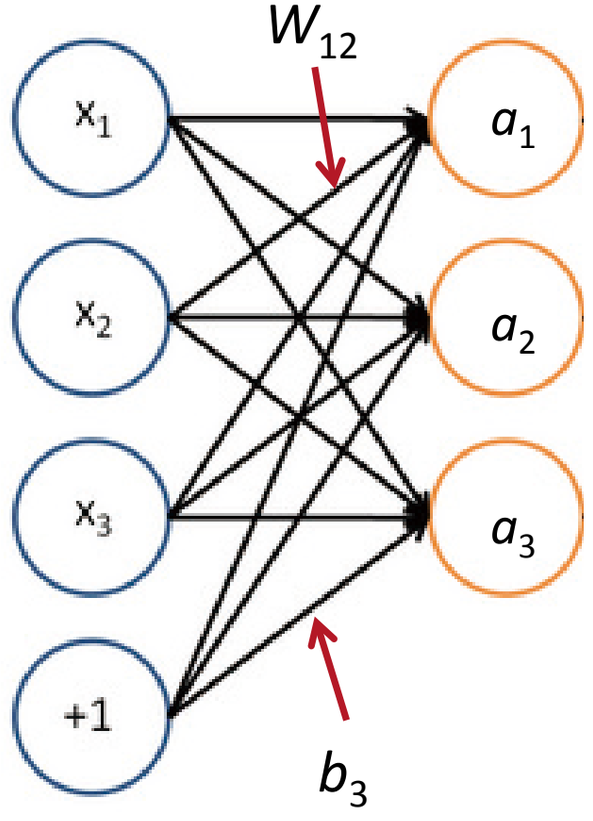
Output layer can be a classification layer or a regression layer

From LR to NN

NTU NLP

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$

.....



In Matrix Notation

$$z = Wx + b$$
$$a = f(z)$$

Where f() is applied element-wise

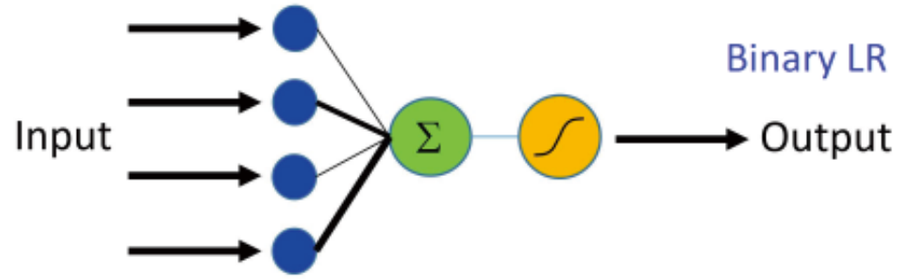
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

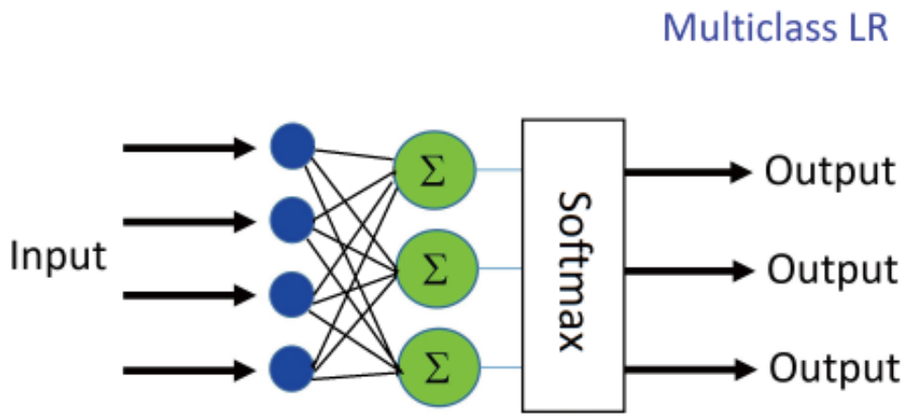
How Do We Train?

If we have only one layer (linear)

It is same as Linear/Logistic Regression

- Use Gradient Descent
- Or L-BFGS (2nd order)
- Or SGD





See Lecture 2 for details

How Do We Train?

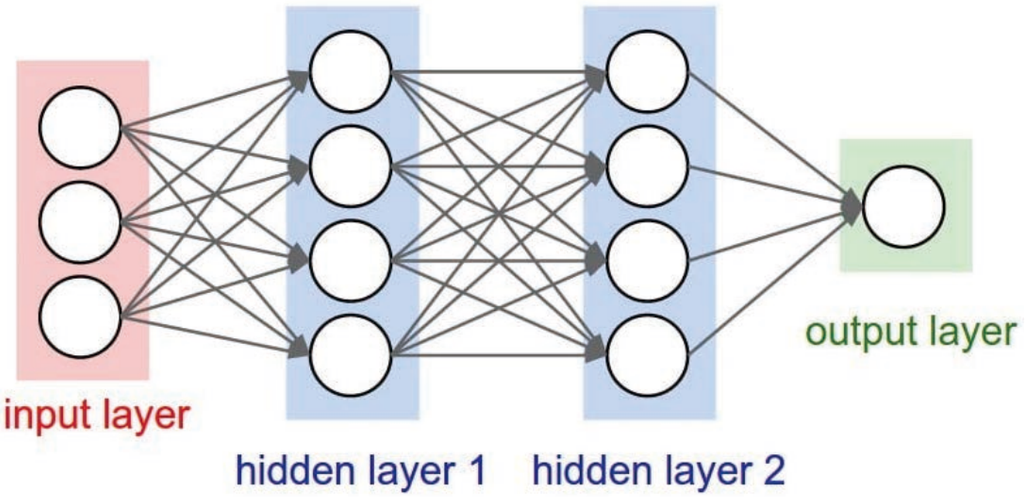
NTU NLP

But we have internal (hidden) layers in a multi-layer NN

This makes the loss/objective function non-convex

- But, we can still use the same ideas/algorithms (just without guarantees)
- Use SGD or its variants

- We "backpropagate" error derivatives through the model



Algorithm 2 Stochastic Gradient Descent Algorithm

initialize θ, η
repeat
  Randomly permute data
  for $i = 1 : N$ **do**
    $\mathbf{g} = \nabla f(\theta, \mathbf{z}_i)$
    $\theta \leftarrow proj_{\Theta}(\theta - \eta \mathbf{g})$
    Update η
  end for
until *converged*

Why Non-Linearity?

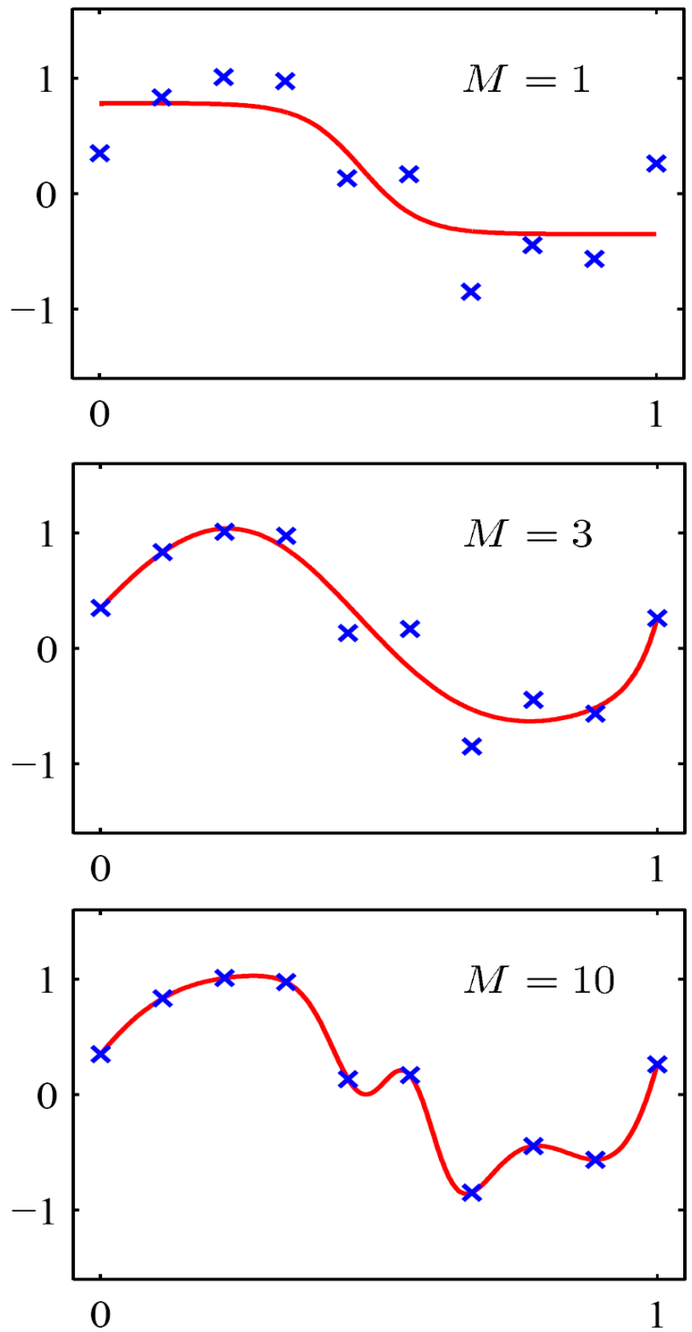
NTU NLP

- For logistic regression: map to probabilities

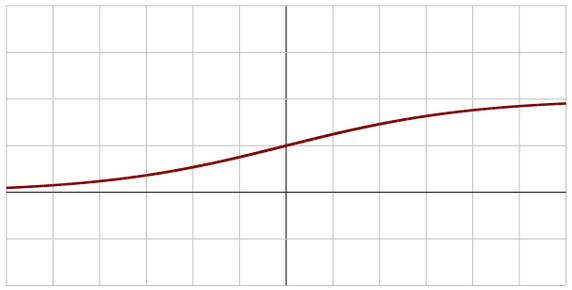
- For NN: function approximation, e.g., regression or classification

- Without non-linearities, NNs can't do anything more than a linear transform
- Extra layers could just be compiled down into a single linear transform W(Wx) = Vx

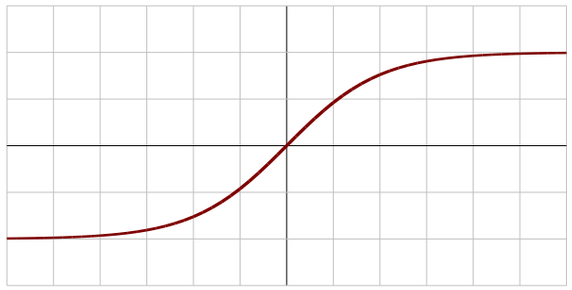
$sigm(\mathbf{w}^\top \mathbf{x})$



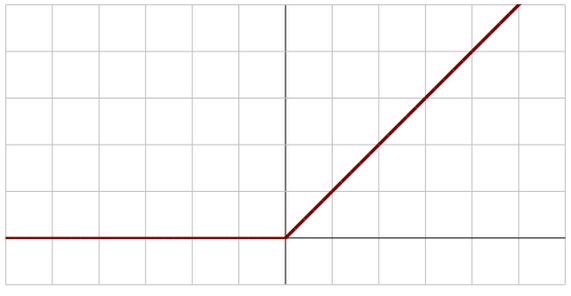
Activation Functions



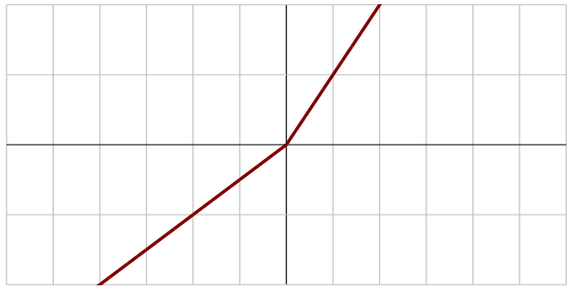
(a) Sigmoid



(b) tanh



(c) ReLU



(d) Leaky ReLU

- Sigmoid

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

- Hyperbolic Tangent:

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

- Rectified Linear Unit (ReLU):

$$f(x) = \max(0, x)$$

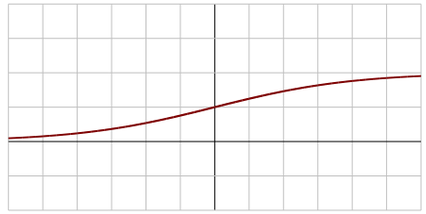
- Leaky ReLU:

$$f(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases}$$

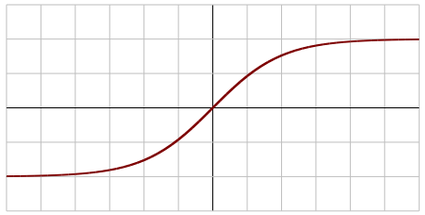
, where α is small constant

Activation Functions

- Sigmoid
  - Biological analogy. Very popular before DL era.
  - Range: $[0, 1]$
  - Gradient vanishing $\rightarrow$ losing popularity in DL era, but still used a lot e.g., LSTM
- Hyperbolic Tangent
  - Centered at 0
  - Gradient vanishing (slightly better than sigmoid function)
- Rectified Linear Unit (ReLU)
  - Alleviate gradient vanishing problem. Safe choice in general.
  - Piece-wise linear
- Leaky ReLU: variant of ReLU

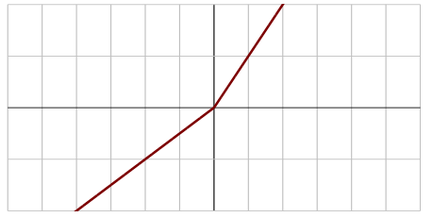


(a) Sigmoid



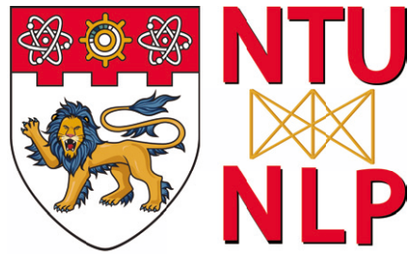
(b) tanh

(c) ReLU



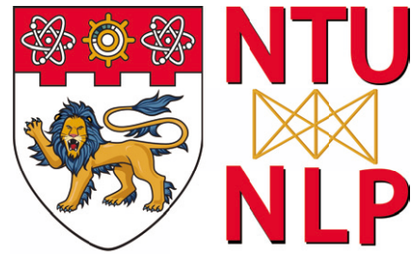
(d) Leaky ReLU

Neural Nets Taxonomy



- Neuron = logistic regression or similar function
- Input layer = input training/test vector
- Bias unit = intercept term/always on feature
- Activation = response
- Activation function is a logistic (or similar "sigmoid" nonlinearity)

Lecture Plan



- Why Deep Learning for NLP
- From Logistic/Linear Regression to NN
  - Activation functions
- SGD with Backpropagation
- Adaptive SGD (adagrad, adam, RMSProp)
- Regularisation (dropout, gradient clipping)
- Introduction to word vectors

Gradient Descent (Recall)

The gradient tells us how to change x (parameters) in order to make a small improvement in our goal.

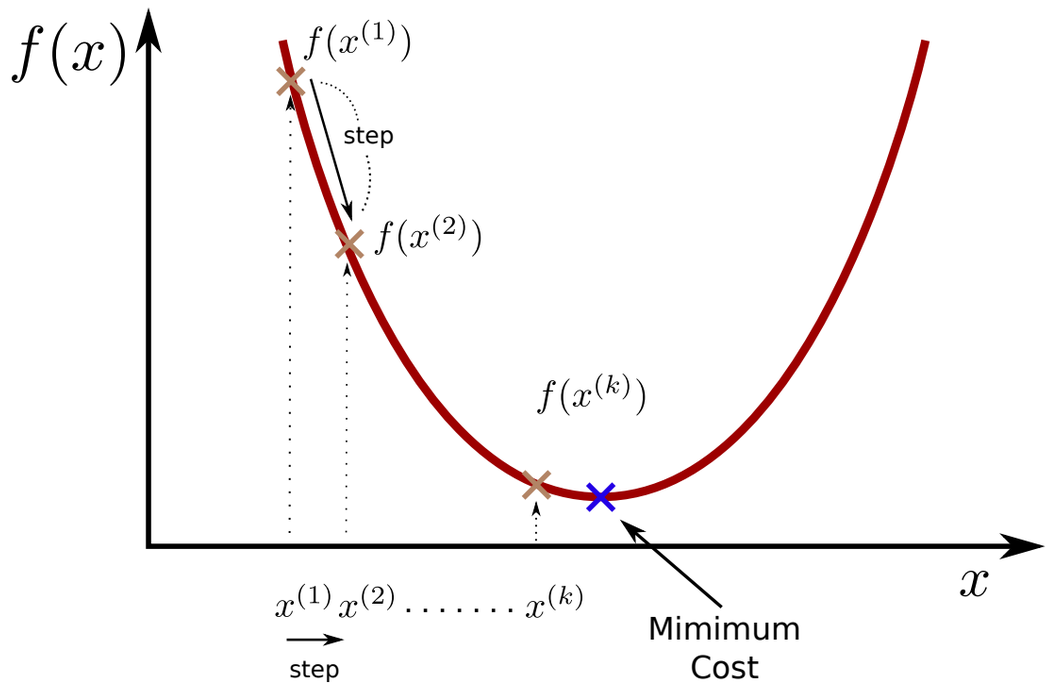


Figure: Illustration of gradient descent

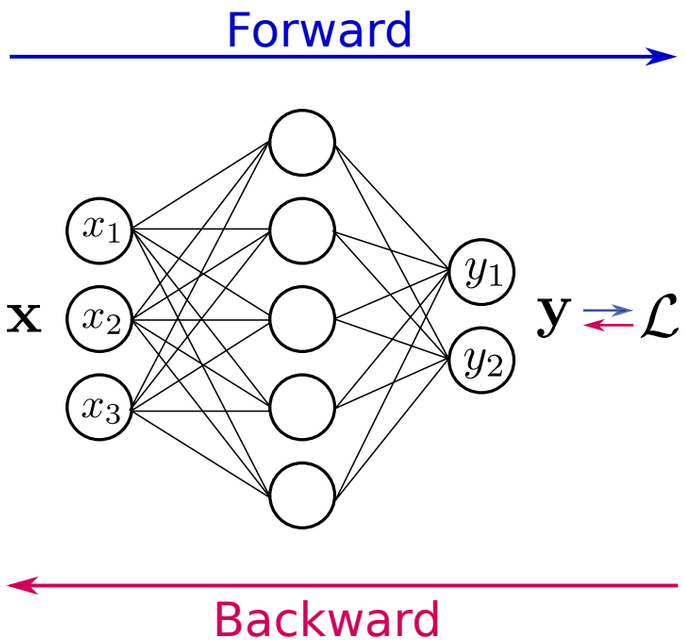
Forward- & Back-propagation

NTU NLP

Two information flow directions:

Forward propagation

- NN accepts an input **x** and produces an output **y**
- During training, it continues onward until it produces a scalar cost **L**



Back-propagation

- Information (**gradients** with respect to the parameters) from the cost flows backward through the network

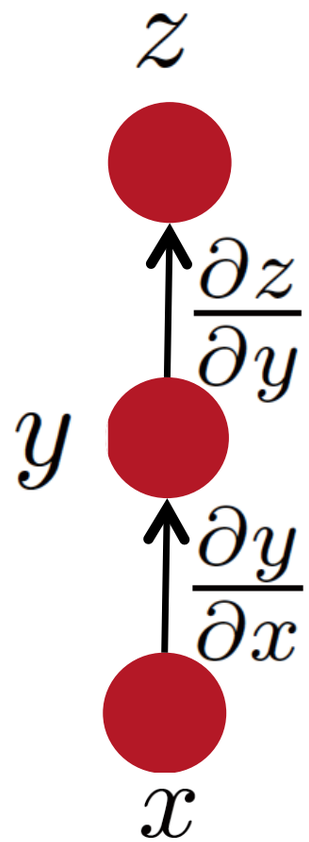
Chain Rule of Derivatives

NTU NLP

Compute gradient of example-wise loss wrt parameters

- Simply apply the chain rule of derivative

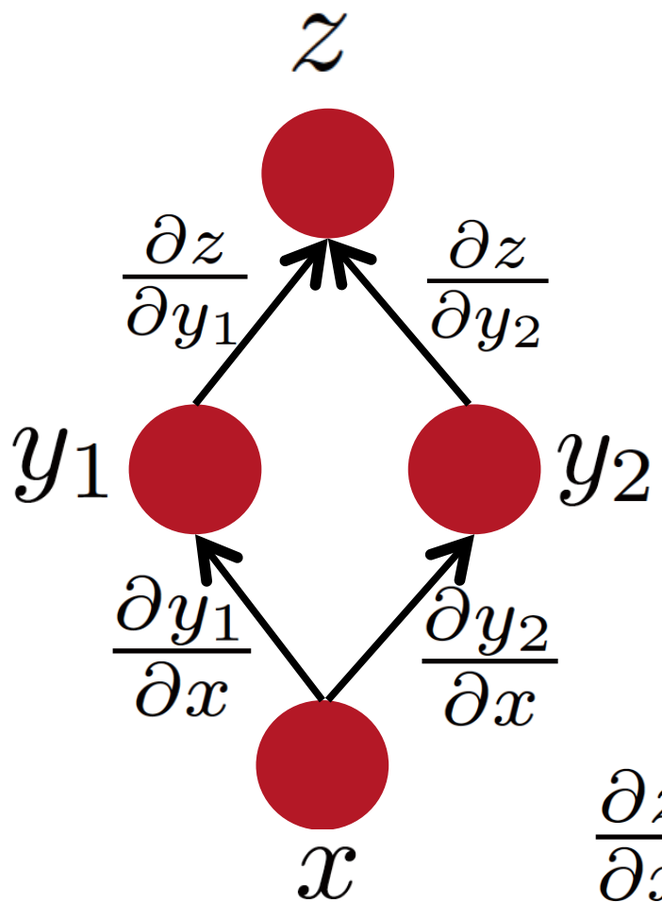
$$z = f(y) \quad y = g(x) \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$



Chain Rule of Derivatives

NTU NLP

- Multiple-path chain rule

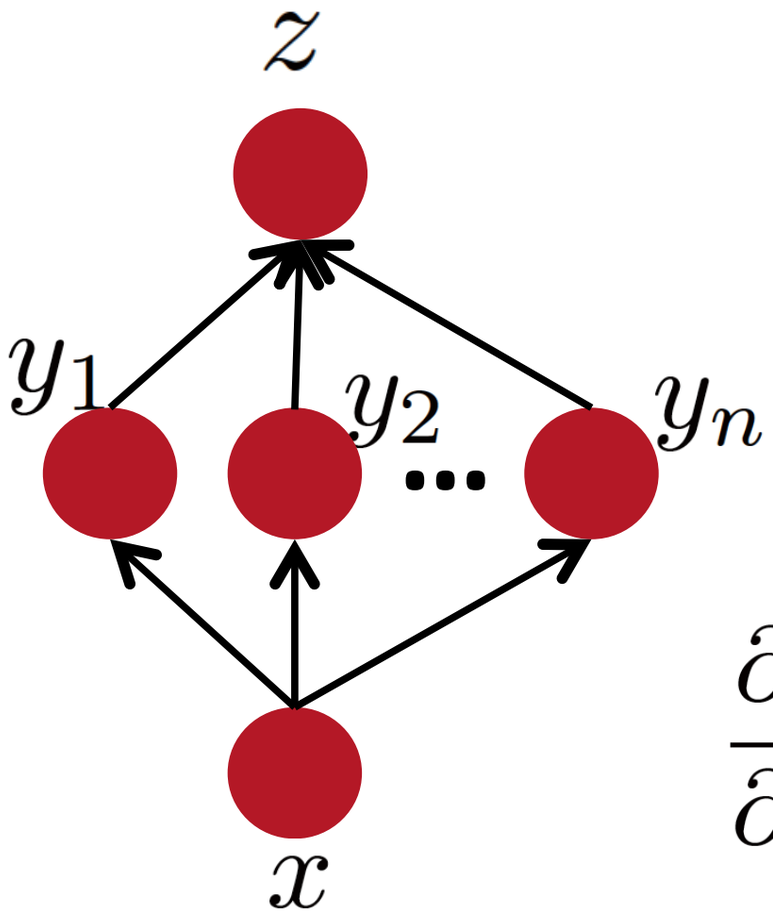


$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Chain Rule of Derivatives

NTU NLP

- Multiple-path chain rule (general)



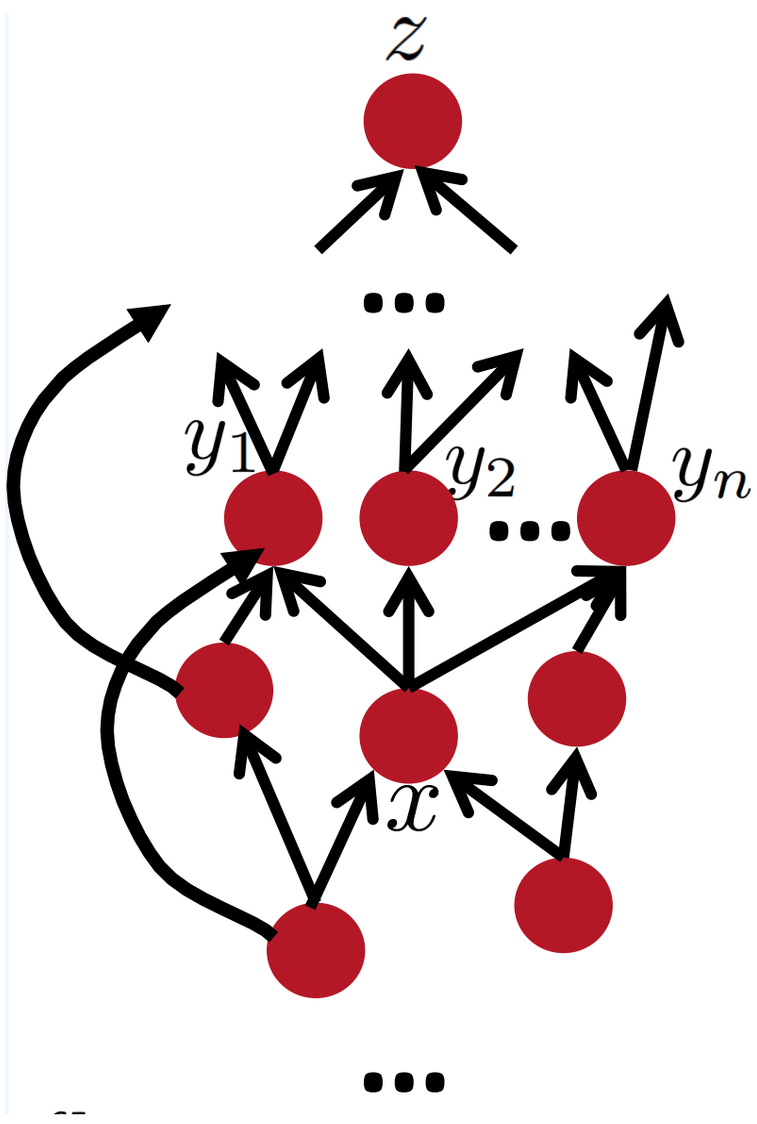
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Source: Socher & Manning

Chain Rule of Derivatives

NTU NLP

- Chain rule in computational graph



Flow graph: any directed acyclic graph
node = computation result
arc = computation dependency

$\{y_1, y_2, \ldots y_n\}$ = successors of x

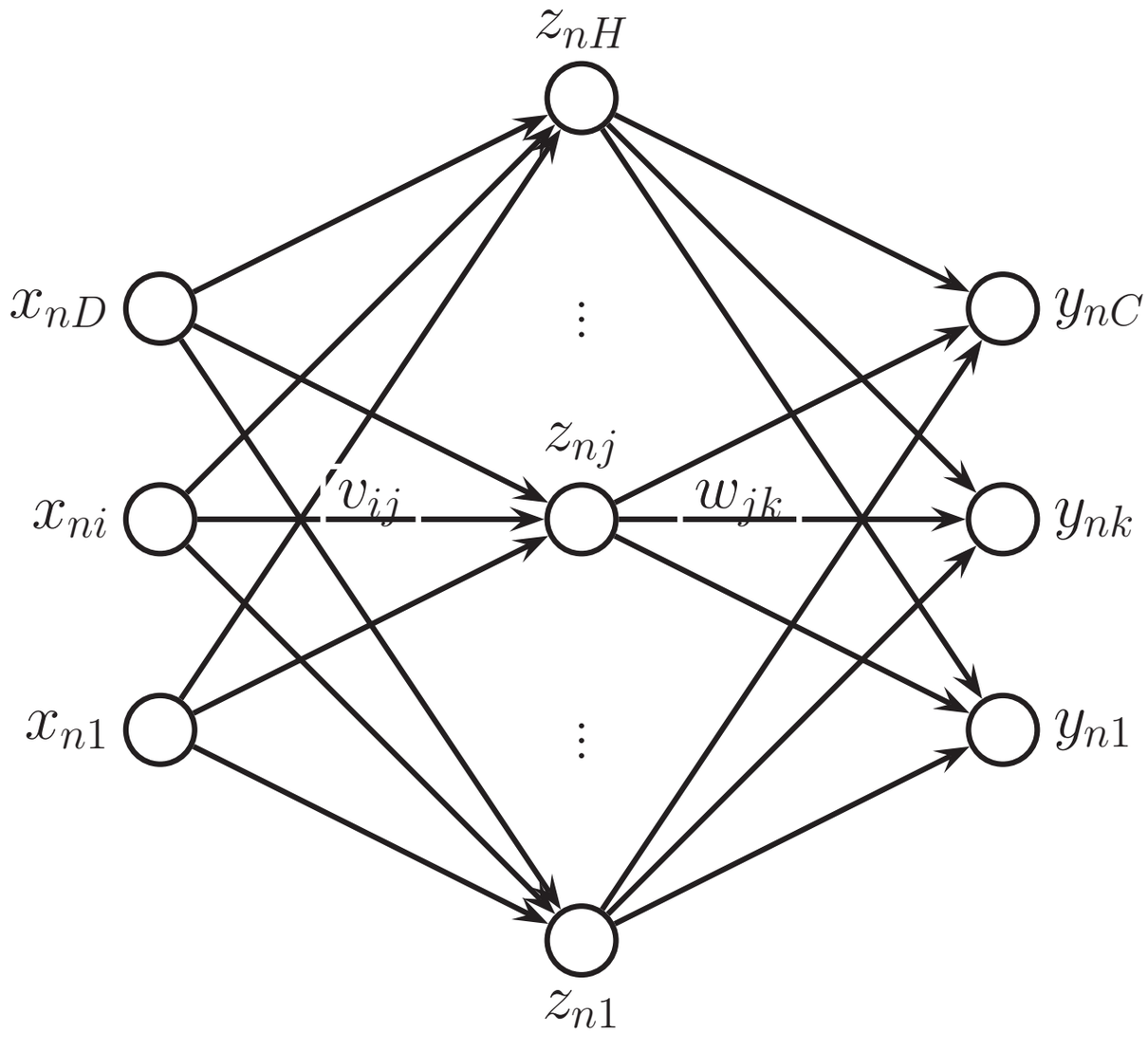
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Source: Socher & Manning

Back-propagation Derivation

- Single Layer NN

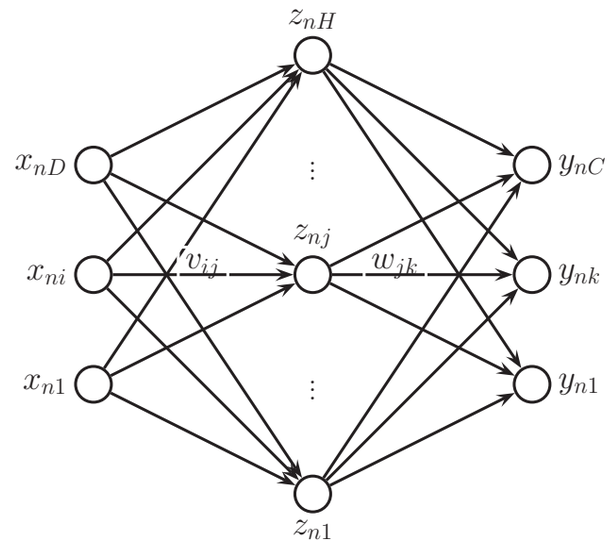
$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \mathbf{y}_n$$



Back-propagation Derivation

NTU NLP

- Gradient of the loss at the last layer



$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \mathbf{y}_n$$

$$J\ \boldsymbol{\theta} \quad -\sum_n \sum_k y_{nk} \log y_{nk}\ \boldsymbol{\theta}$$

where

$$\boldsymbol{\theta} \quad \mathbf{V}, \mathbf{W}$$

Our task is to compute: $\nabla_{\boldsymbol{\theta}} J)$

$$\nabla_{\mathbf{w}_k} J_n \quad \frac{\partial J_n}{\partial b_{nk}} \nabla_{\mathbf{w}_k} b_{nk}$$

Chain rule

For any GLM: $\frac{\partial J_n}{\partial b_{nk}} \triangleq \delta_{nk}^{w}$ $y_{nk} - y_{nk}$

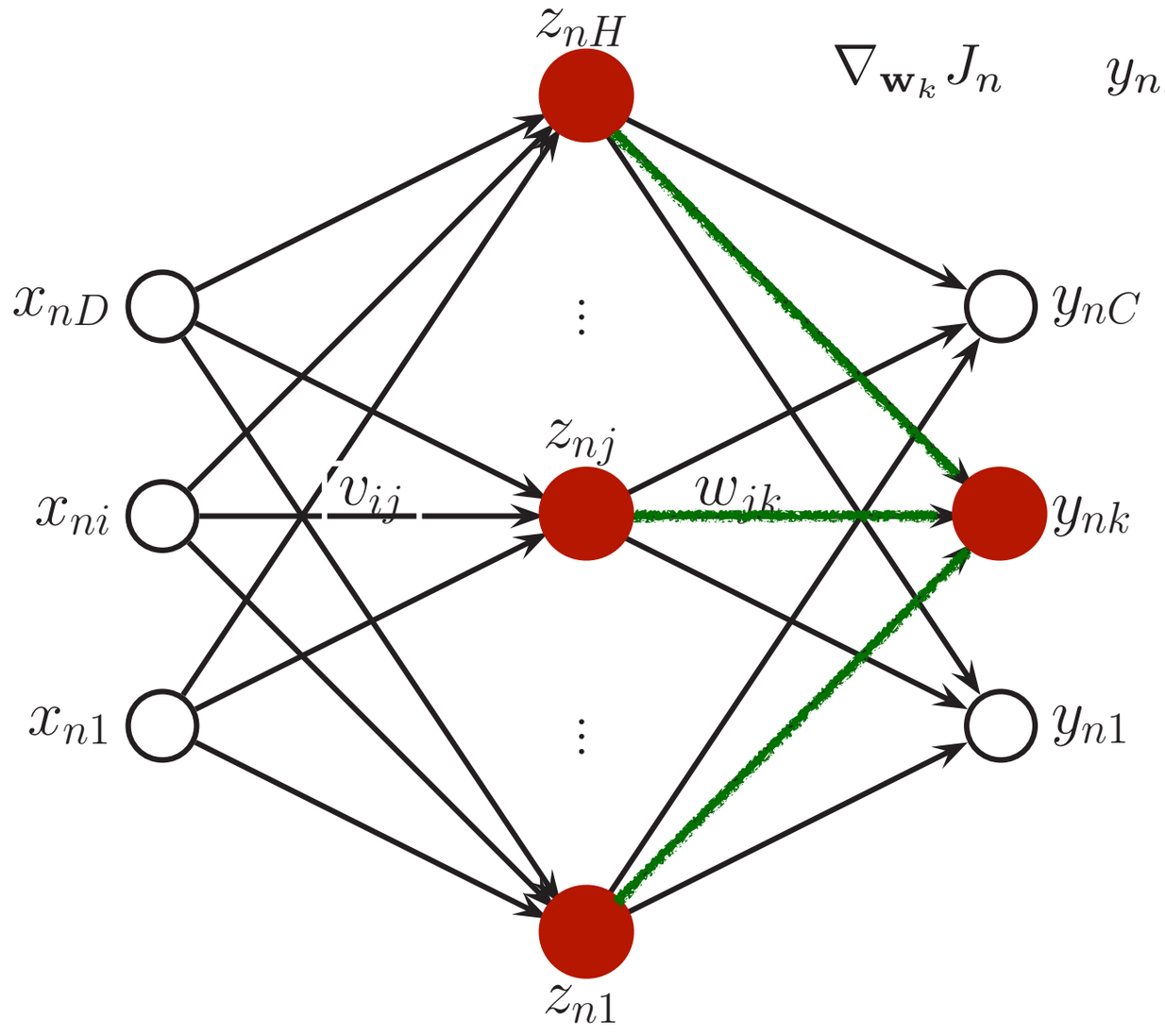
Assignment 1 for Softmax

$$\nabla_{\mathbf{w}_k} b_{nk} \quad \mathbf{z}_n \qquad \nabla_{\mathbf{w}_k} J_n \quad y_{nk} - y_{nk}\ \mathbf{z}_n$$

Back-propagation Derivation

- Gradient of the loss at the last layer

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \mathbf{y}_n$$



$\nabla_{\mathbf{w}_k} J_n$ $y_{nk} - y_{nk}$ $\mathbf{z}_n$

$y_{nk} - y_{nk}$

Error signal

Back-propagation Derivation

NTU NLP

- Gradient of the loss at the first layer

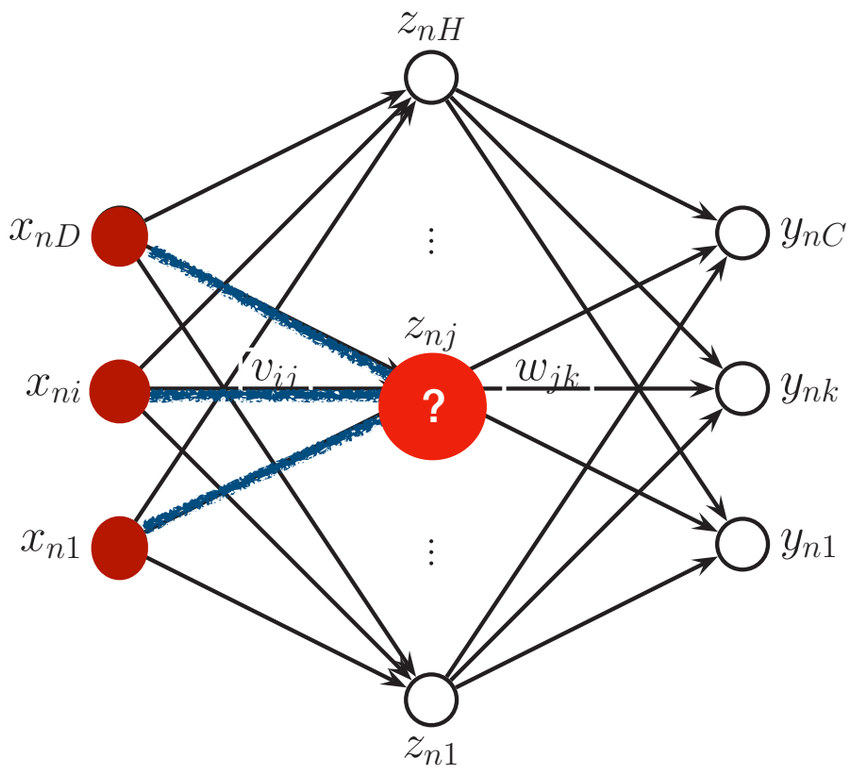
$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \mathbf{y}_n$$

Gradient wrt V

$$\nabla_{\mathbf{v}_j} J_n \quad \frac{\partial J_n}{\partial a_{nj}} \nabla_{\mathbf{v}_j} a_{nj} \triangleq \delta_{nj}^{v} \mathbf{x}_n$$

Since

$$a_{nj} \quad \mathbf{v}_j^T \mathbf{x}_n$$



All that remains is to compute the first level error signal δ_{nj}^{v}

Back-propagation Derivation

All that remains is to compute the first level error signal δ_{nj}^{v}

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \mathbf{y}_n$$

δ_{nj}^{v} $\frac{\partial J_n}{\partial a_{nj}}$ $\sum_{k=1}^{K} \frac{\partial J_n}{\partial b_{nk}} \frac{\partial b_{nk}}{\partial a_{nj}}$ We know: $\frac{\partial J_n}{\partial b_{nk}} \triangleq \delta_{nk}^{w}$ $y_{nk} - y_{nk}$

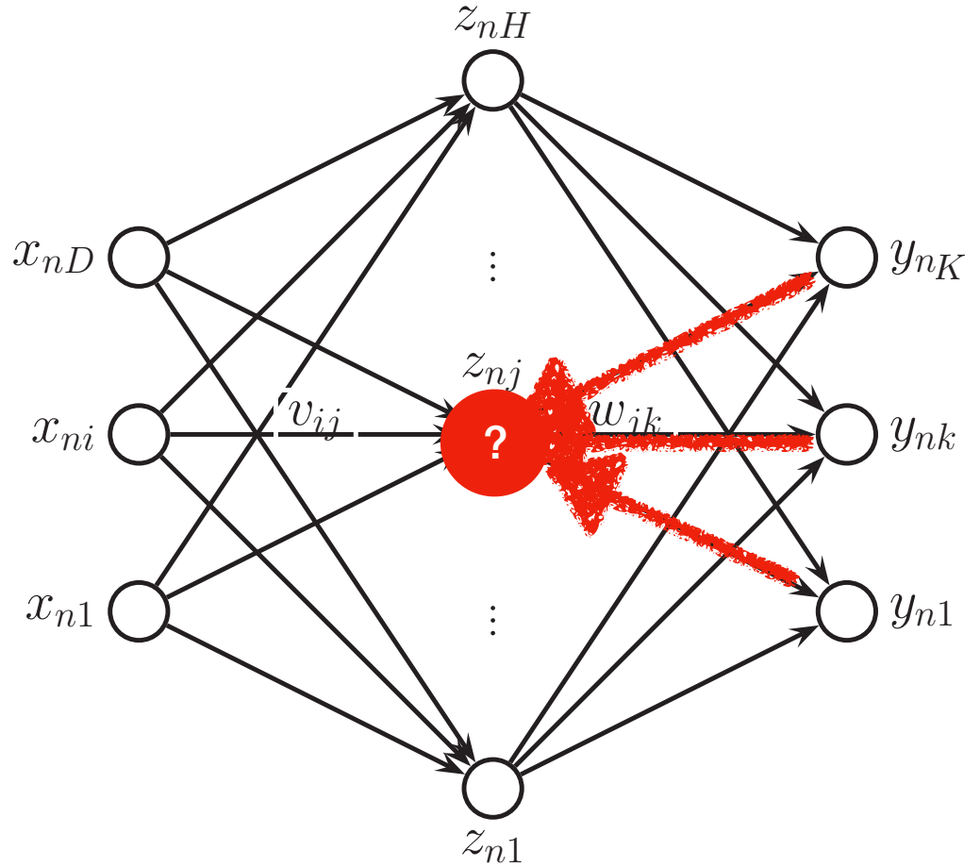
Multiple-path chain rule

b_{nk} $\sum_{j} w_{kj} g\ a_{nj}$ $w_{kj} g'\ a_{nj}$

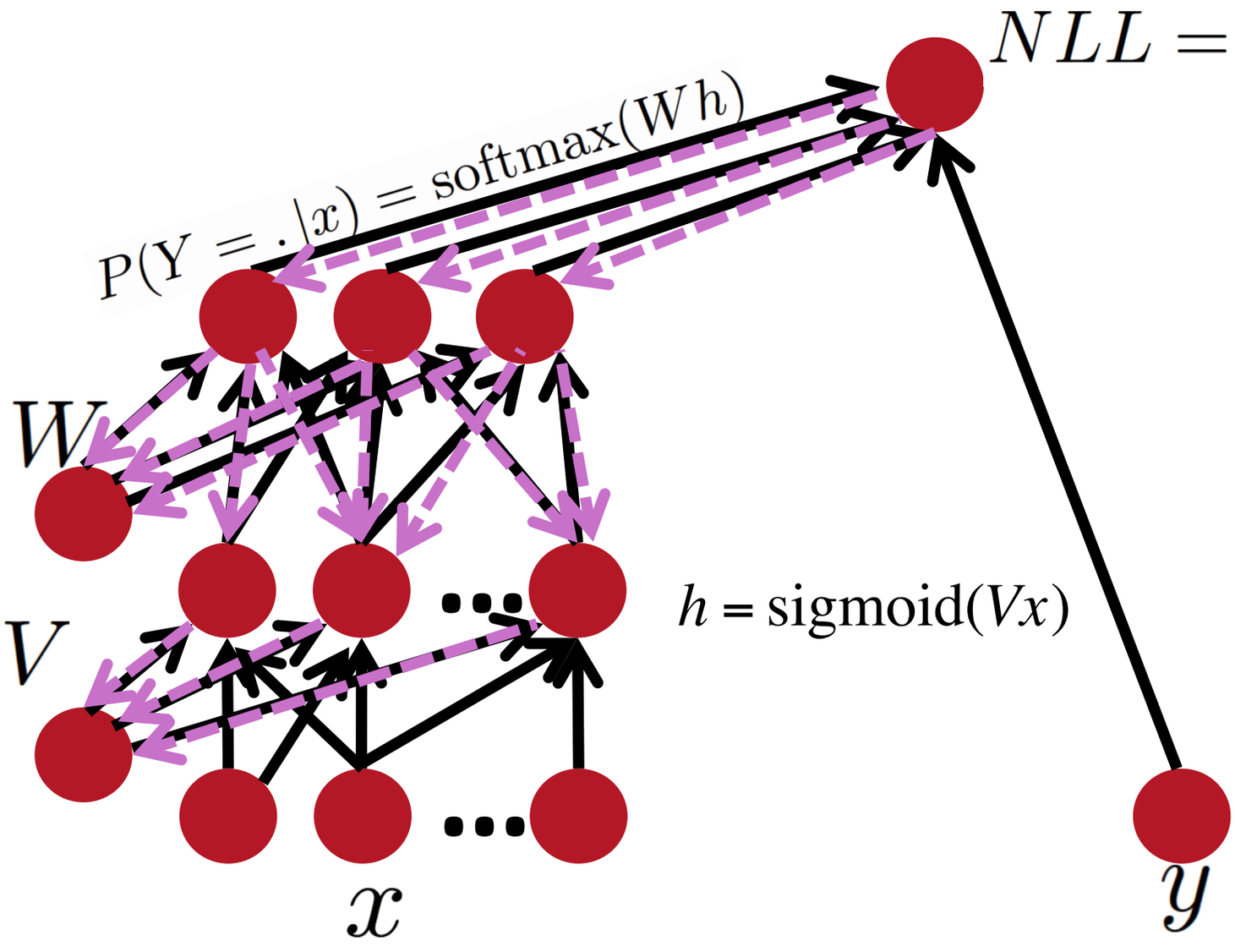
Finally,

δ_{nj}^{v} $\sum_{k=1}^{K} \delta_{nk}^{w} w_{kj} g'\ a_{nj}$

$\nabla_{\boldsymbol{\theta}} J\ \boldsymbol{\theta}$ $\sum_{n} [\boldsymbol{\delta}_n^{v} \mathbf{x}_n,\ \boldsymbol{\delta}_n^{w} \mathbf{z}_n]$



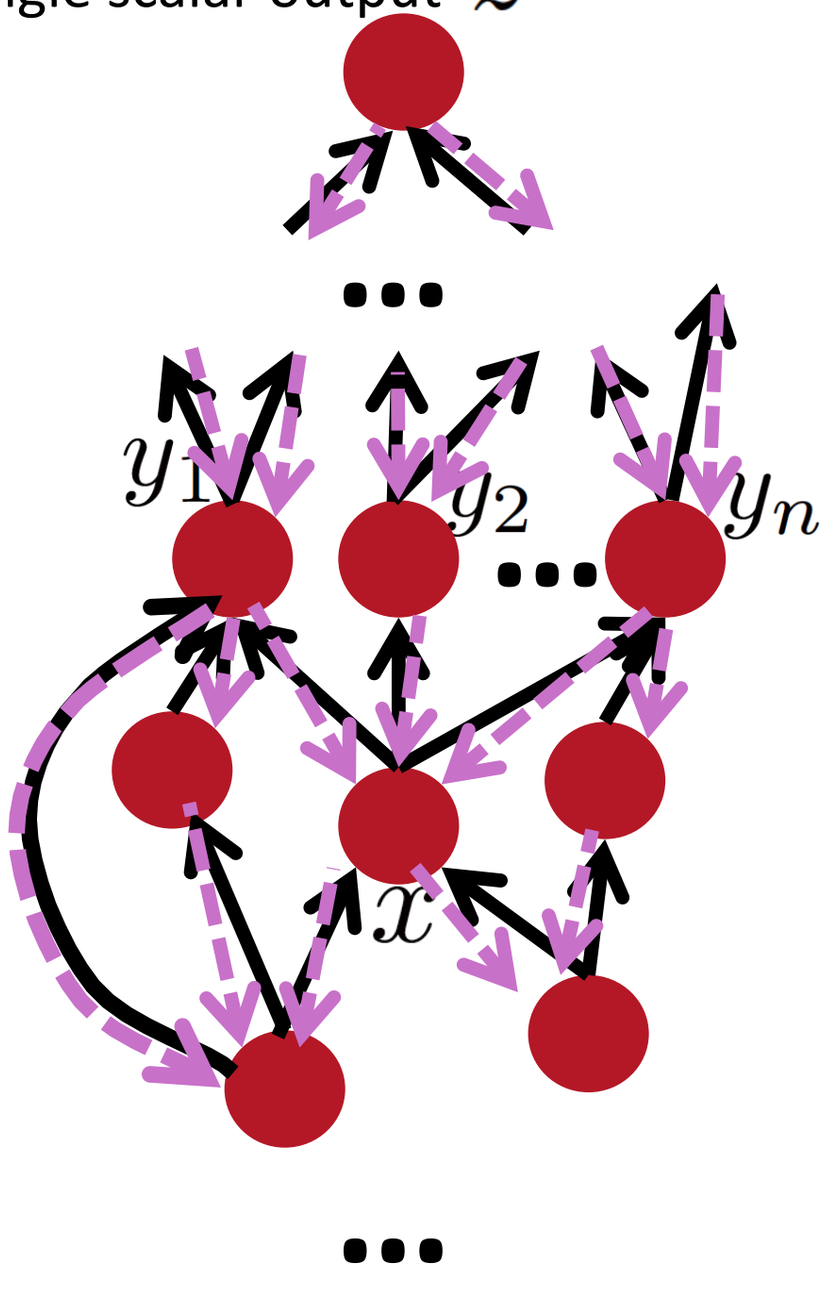
Back-propagation in an MLP



Source: Socher & Manning

Back-propagation in a General Computational Graph

Single scalar output z



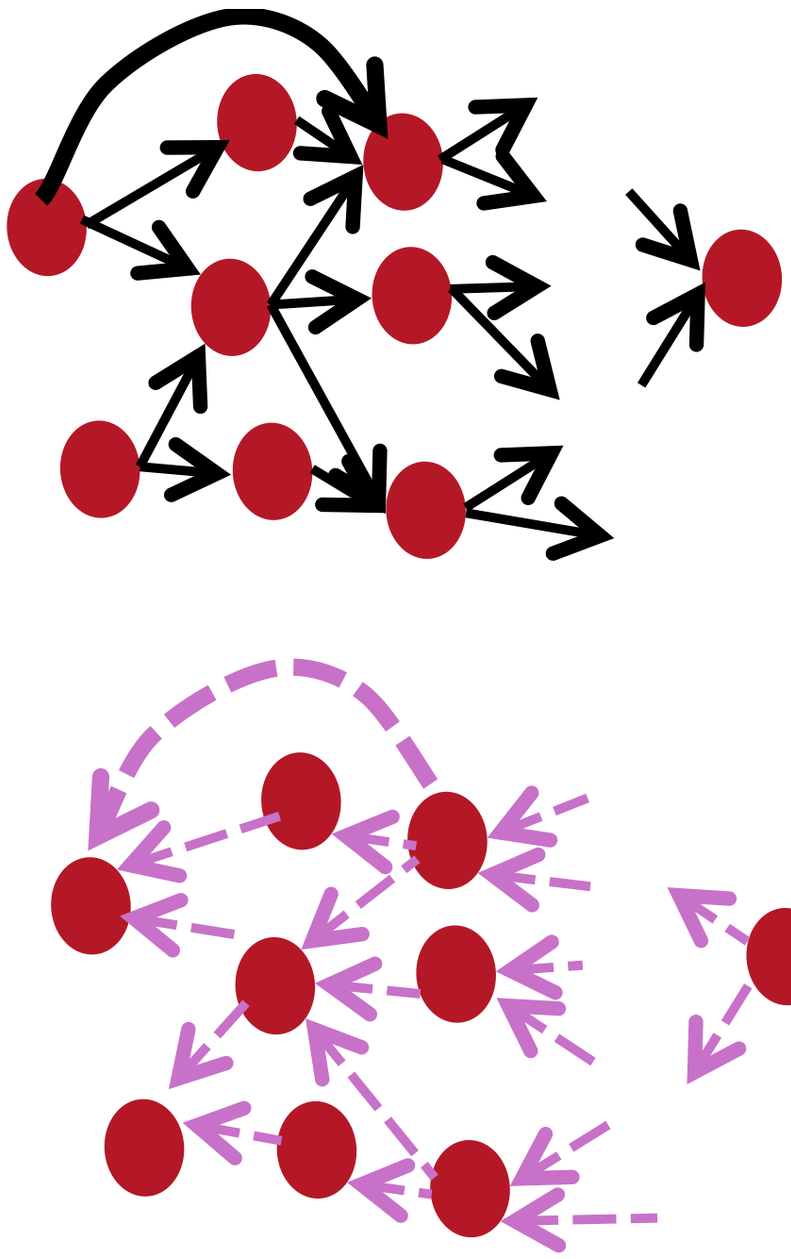
1. Fprop: visit nodes in topo-sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \ldots y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Source: Socher & Manning

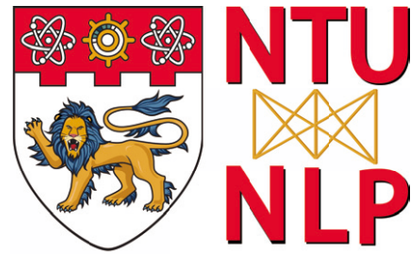
Automatic Gradient Computation



- The gradient computation can be automatically inferred from the symbolic expression of the fprop.
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.
- Easy and fast prototyping

Source: Socher & Manning

Lecture Plan



- Why Deep Learning for NLP
- From Logistic/Linear Regression to NN
  - Activation functions
  - SGD with Backpropagation
- Adaptive SGD (adagrad, adam, RMSProp)
- Regularisation (dropout, gradient clipping)
- Introduction to word vectors

Gradient descent variants

$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$$

1. Batch gradient descent
2. Stochastic gradient descent
3. Mini-batch gradient descent

Difference: Amount of data used per update

Source: Ruder, 2018

Batch gradient descent

- Computes gradient with the **entire** dataset.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(
    loss_function, data, params)
  params = params - learning_rate * params_grad
```

Source: Ruder, 2018

Batch gradient descent

- Pros:
  - Guaranteed to converge to **global** minimum for **convex** error surfaces and to a **local** minimum for **non-convex** surfaces.
- Cons:
  - **Very slow**.
  - Intractable for datasets that **do not fit in memory**.
  - **No online learning**.

Source: Ruder, 2018

Stochastic gradient descent

- Computes update for **each** example $x^{(i)}y^{(i)}$.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(
      loss_function, example, params)
    params = params - learning_rate * params_grad
```

Source: Ruder, 2018

Stochastic gradient descent

- Pros
  - **Much faster** than batch gradient descent.
  - Allows **online learning**.
- Cons
  - **High variance** updates.

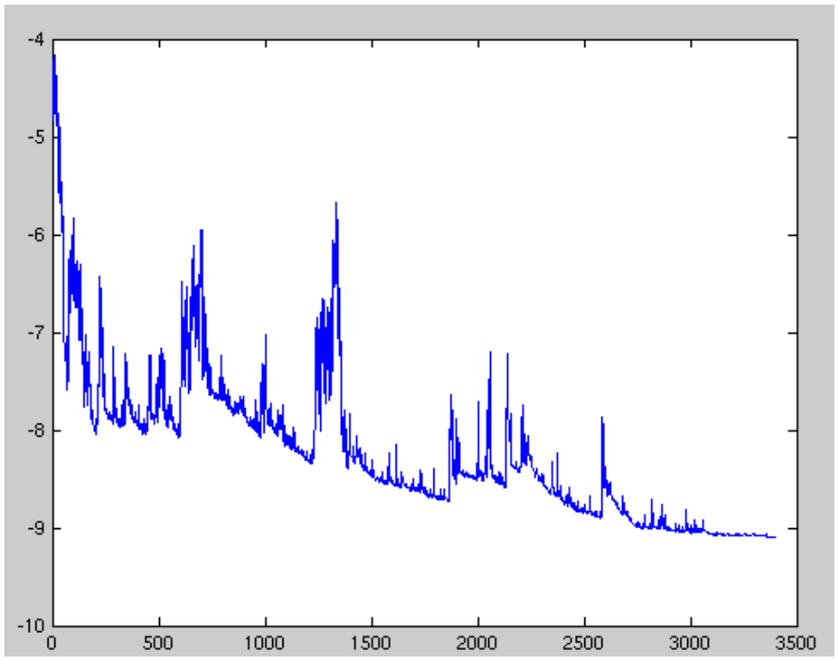


Figure: SGD fluctuation (Source: Wikipedia)

Minibatch gradient descent

- Performs update for every **mini-batch** of n examples.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(
      loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Source: Ruder, 2018

Minibatch gradient descent

- Pros
  - **Reduces variance** of updates.
  - Can exploit **matrix multiplication** primitives.
- Cons
  - **Mini-batch size** is a hyperparameter. Common sizes are 50-256.
- Typically the algorithm of choice.
- Usually referred to as SGD even when mini-batches are used.

Source: Ruder, 2018

Comparison

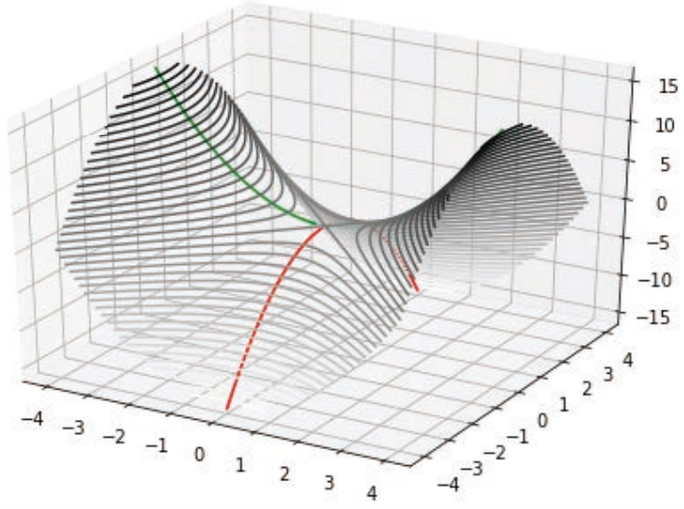
Method	Accuracy	Update Speed	Memory Usage	Online Learning
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

Table: Comparison of trade-offs of gradient descent variants

Source: Ruder, 2018

Challenges

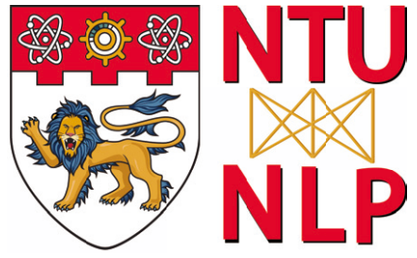
- Choosing a **learning rate**.
- Defining an **annealing schedule**.
- Updating features to **different extent**.
- **Avoiding suboptimal minima**.



Saddle Point

Source: Ruder, 2018

SGD Variants



1. Momentum
2. Nesterov accelerated gradient
3. Adagrad
4. Adadelta
5. RMSprop
6. Adam
7. Adam extensions

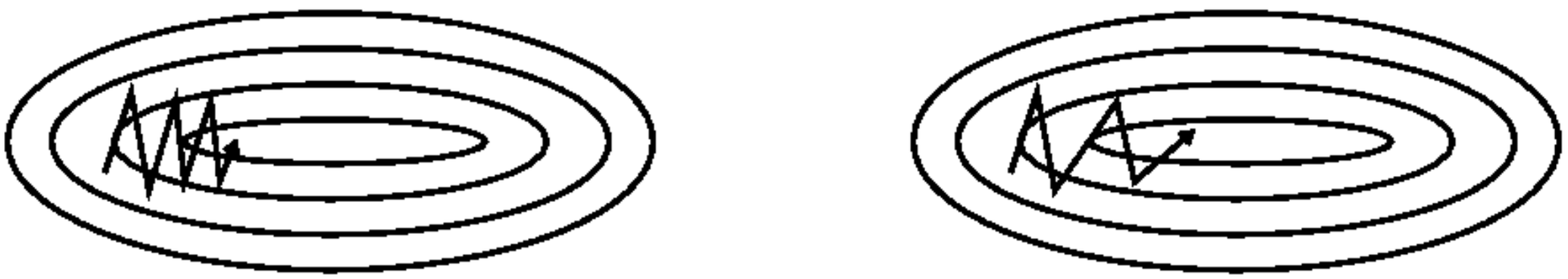
See https://ruder.io/optimizing-gradient-descent/

Source: Ruder, 2018

SGD with Momentum

- SGD has trouble navigating **ravines**.
- Momentum [Qian, 1999] helps SGD **accelerate**.
- Adds a fraction γ of the update vector of the past step v_{t-1} to current update vector v_t. Momentum term γ is usually set to 0.9.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{1}$$



(a) SGD without momentum

(b) SGD with momentum

Figure: Source: Genevieve B. Orr

Source: Ruder, 2018

SGD with Momentum

- **Reduces updates** for dimensions whose gradients **change directions**.
- **Increases updates** for dimensions whose gradients **point in the same directions**.

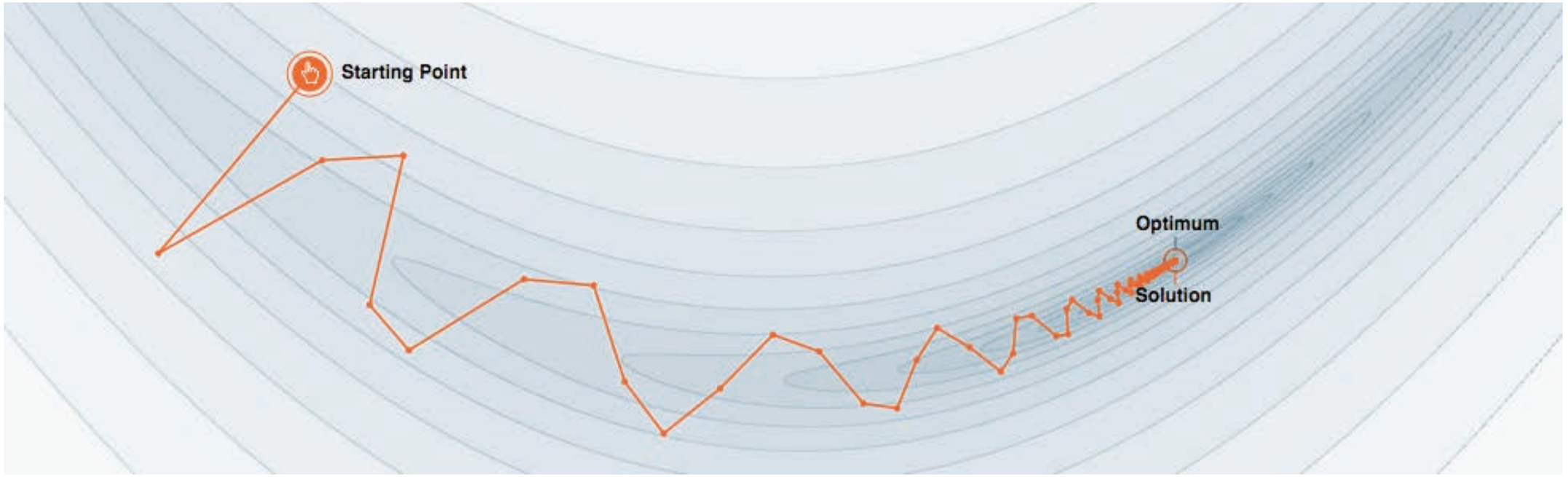


Figure: Optimization with momentum (Source: distill.pub)

Source: Ruder, 2018

Adagrad

- Previous methods: **Same learning rate** η for all parameters θ.
  - SGD update: $\theta_{t+1} = \theta_t - \eta \cdot g_t$
    - $g_t = \nabla_{\theta_t} J(\theta_t)$

- Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (**large** updates for **infrequent** parameters, **small** updates for **frequent** parameters).

- Adagrad divides the learning rate by the **square root of the sum of squares of historic gradients**.
  - Adagrad update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{3}$$

    - $G_t \in \mathbb{R}^{d \times d}$: diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t
    - ϵ: smoothing term to avoid division by zero

Source: Ruder, 2018

Adagrad

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

- Pros
  - Well-suited for dealing with **sparse data**.
  - Significantly **improves robustness** of SGD.
  - Lesser need to manually tune learning rate.
- Cons
  - **Accumulates squared gradients** in denominator. Causes the learning rate to **shrink** and become **infinitesimally small**.

Pennington et al. [11] used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Source: Ruder, 2018

Adadelta

- Adadelta [Zeiler, 2012] restricts the window of accumulated past gradients to a **fixed size**. SGD update:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{4}$$

- Defines **running average** of squared gradients $E[g^2]_t$ at time t:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{5}$$

  - γ: fraction similarly to momentum term, around 0.9

- Adagrad update:

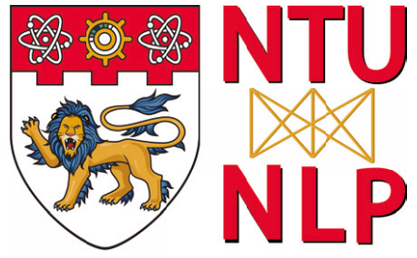
$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

- Preliminary Adadelta update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Source: Ruder, 2018

Adadelta



- Preliminary Adadelta update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- Denominator is just root mean squared (RMS) error of gradient:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \tag{9}$$

- Note: **Hypothetical units do not match**.
- Define **running average of squared parameter updates** and RMS:

$$\begin{aligned} E[\Delta\theta^2]_t &= \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \\ RMS[\Delta\theta]_t &= \sqrt{E[\Delta\theta^2]_t + \epsilon} \end{aligned} \tag{10}$$

- Approximate with $RMS[\Delta\theta]_{t-1}$, replace η for **final Adadelta update**:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \tag{11}$$

RMSProp

- Developed independently from Adadelta around the same time by Geoff Hinton.
- Also divides learning rate by a **running average of squared gradients**.
- RMSprop update:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{12}$$

  - γ: decay parameter; typically set to 0.9
  - η: learning rate; a good default value is 0.001

Source: Ruder, 2018

Adam

- Adaptive Moment Estimation (Adam) [Kingma and Ba, 2015] also stores **running average of past squared gradients** v_t like Adadelta and RMSprop.

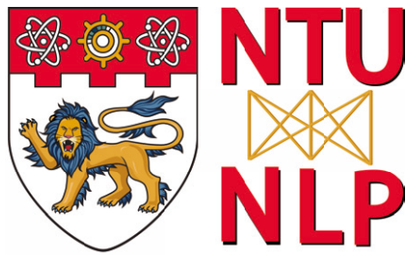
- Like Momentum, stores **running average of past gradients** m_t.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2
\end{aligned}
\tag{13}
$$

- m_t: first moment (mean) of gradients
- v_t: second moment (uncentered variance) of gradients
- β_1, β_2: decay rates

Source: Ruder, 2018

Adam



- m_t and v_t are initialized as 0-vectors. For this reason, they are biased towards 0.

- Compute bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{14}$$

- Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{15}$$

Source: Ruder, 2018

Summary

Method	Update equation
	$g_t = \nabla_{\theta_t} J(\theta_t)$
SGD	$\Delta\theta_t = -\eta \cdot g_t$
	$\theta_t = \theta_t + \Delta\theta_t$
Momentum	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta g_t$
NAG	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
Adagrad	$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$
Adadelta	$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$
RMSprop	$\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$
Adam	

- Adaptive learning rate methods (Adagrad, Adadelta, RMSprop, Adam) are **particularly useful for sparse features**.
- Adagrad, Adadelta, RMSprop, and Adam work well in similar circumstances.
- [Kingma and Ba, 2015] show that bias-correction helps Adam **slightly outperform RMSprop**.

Table: Update equations for the gradient descent optimization algorithms.

Source: Ruder, 2018

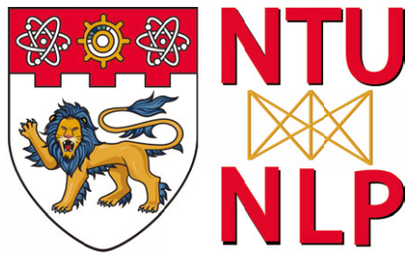
Which optimizer?

NTU NLP

- Adaptive learning rate methods (Adagrad, Adadelta, RMSprop, Adam) are **particularly useful for sparse features**.
- Adagrad, Adadelta, RMSprop, and Adam work well in similar circumstances.
- [Kingma and Ba, 2015] show that bias-correction helps Adam **slightly outperform RMSprop**.

Source: Ruder, 2018

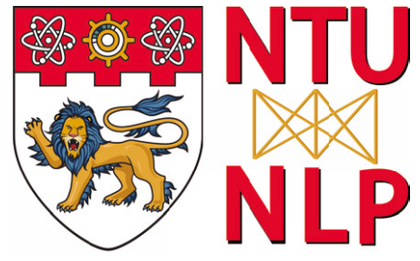
Additional strategies



1. Shuffling and Curriculum Learning [Bengio et al., 2009]
   - Shuffle training data after every epoch to **break biases**
   - Order training examples to **solve progressively harder problems**; infrequently used in practice
2. Batch normalization [Ioffe and Szegedy, 2015]
   - **Re-normalizes every mini-batch** to zero mean, unit variance
   - Must-use for computer vision
3. Early stopping
   - "*Early stopping (is) beautiful free lunch*" (Geoff Hinton)
4. Gradient noise [Neelakantan et al., 2015]
   - Add Gaussian noise to gradient
   - Makes model **more robust to poor initializations**

Source: Ruder, 2018

Tuned SGD vs. Adam



- Many recent papers use **SGD with learning rate annealing**.
- SGD with tuned learning rate and momentum is **competitive with Adam** [Zhang et al., 2017b].
- Adam **converges faster**, but **underperforms SGD** on some tasks, e.g. Machine Translation [Wu et al., 2016].
- Adam with **2 restarts and SGD-style annealing** converges faster and outperforms SGD [Denkowski and Neubig, 2017].
- **Increasing the batch size** may have the same effect as decaying the learning rate [Smith et al., 2017].

Source: Ruder, 2018

Learning to optimize

- Rather than manually defining an update rule, **learn it**.
- Update rules outperform existing optimizers and transfer across tasks.

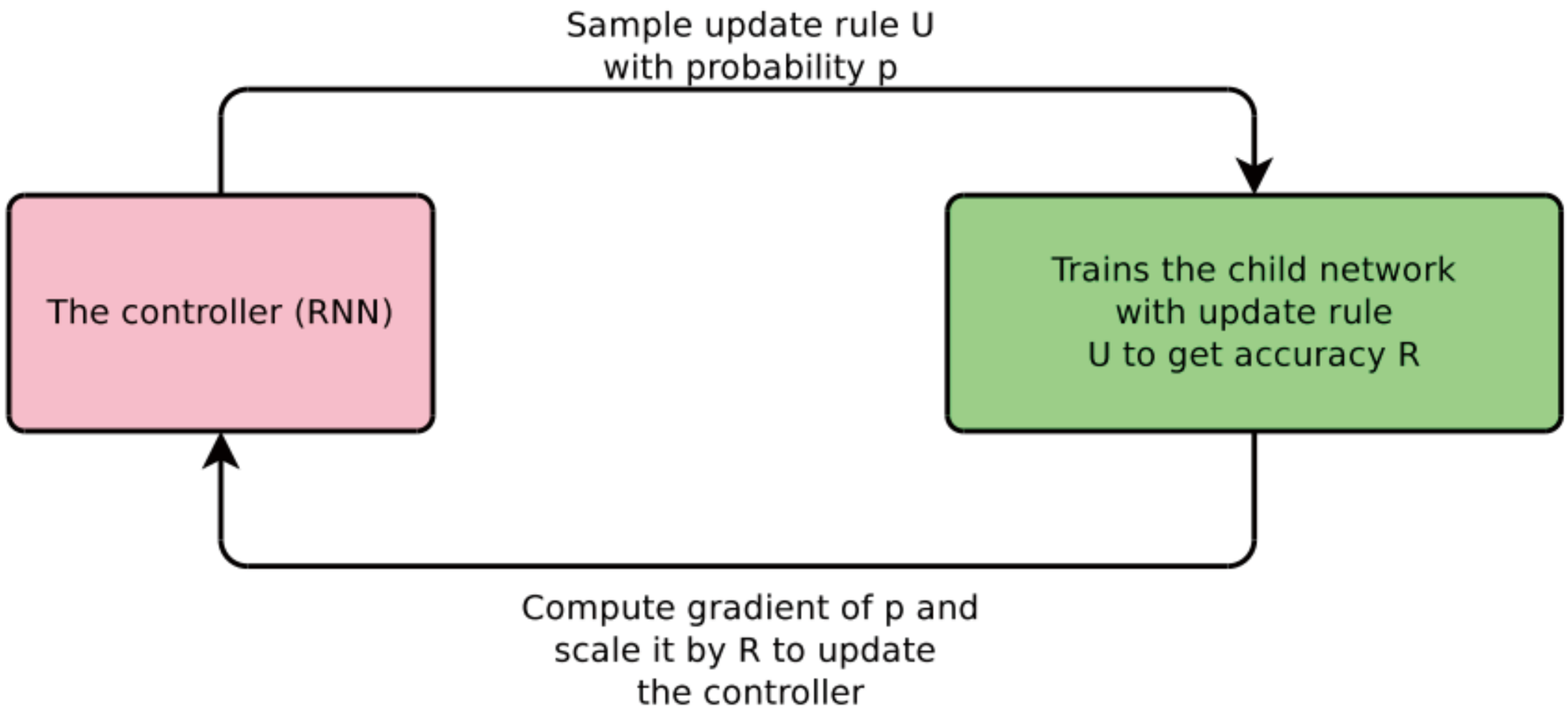
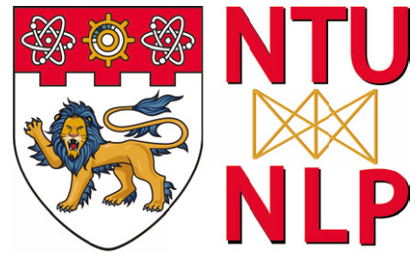


Figure: Neural Optimizer Search [Bello et al., 2017]

Source: Ruder, 2018

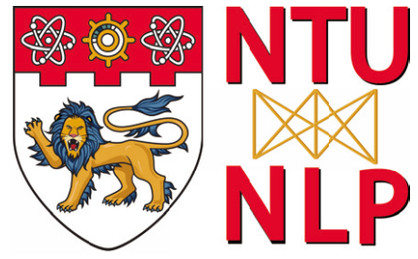
Optimisation and Generalization



- **Optimization** is closely tied to **generalization**.
- The number of possible local minima **grows exponentially with the number of parameters** [Kawaguchi, 2016].
- Different local minima **generalize to different extents**.
- Recent insights in understanding generalization:
  - Neural networks can **completely memorize random inputs** [Zhang et al., 2017a].
  - Sharp minima found by batch gradient descent have **high generalization error** [Keskar et al., 2017].
  - Local minima that generalize well can be **made arbitrarily sharp** [Dinh et al., 2017].
- Several submissions at ICLR 2018 on understanding generalization.

See https://ruder.io/optimizing-gradient-descent/

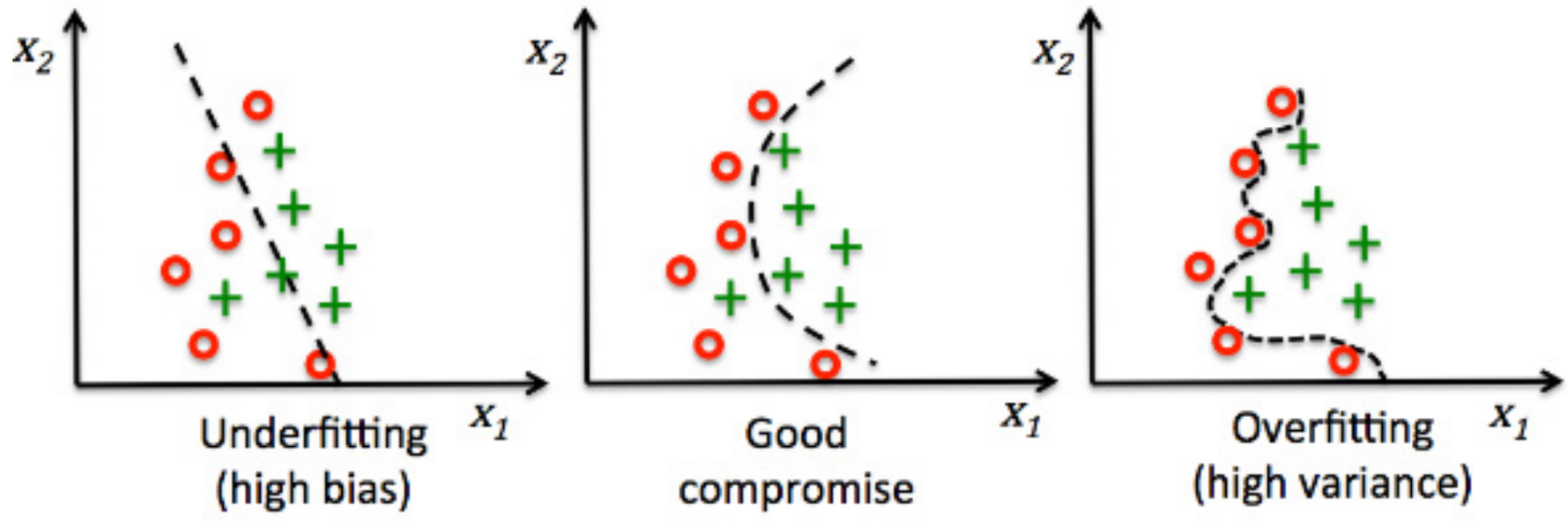
Lecture Plan



- Why Deep Learning for NLP
- From Logistic/Linear Regression to NN
  - Activation functions
  - SGD with Backpropagation
  - Adaptive SGD (adagrad, adam, RMSProp)
- Regularisation & Initialization
- Introduction to word vectors

Regularisation

DNNs tend to overfit to the training data → poor generalisation performance.

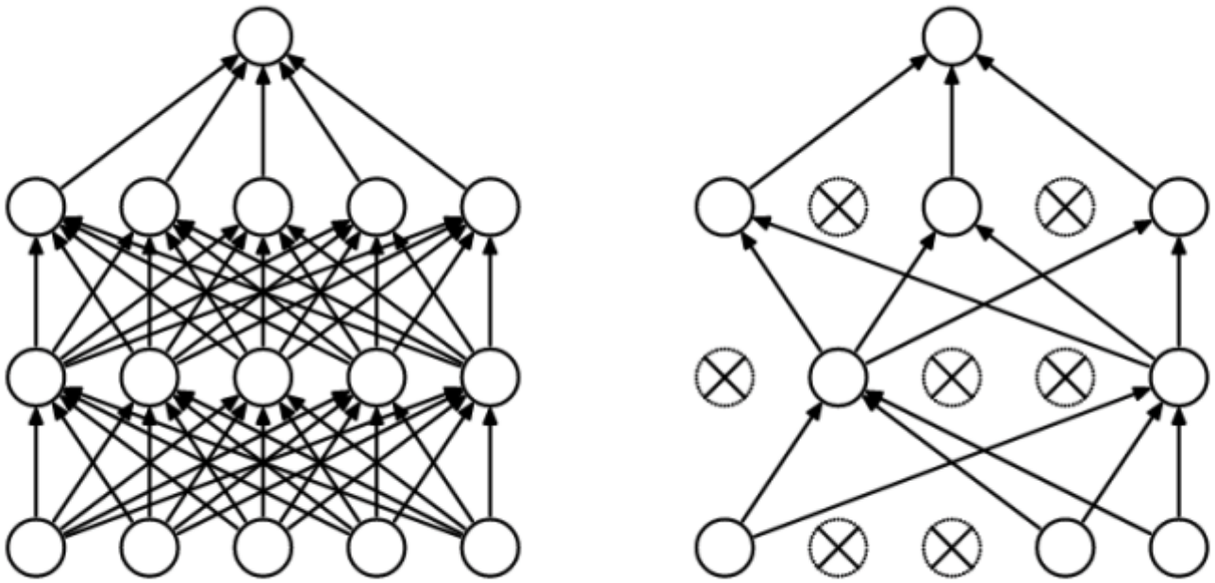


Generalization error increases due to overfitting.

We need to regularise the network

Dropout

- Simple, but powerful regulariser
- Randomly turnoff some nodes during training.
- Use all activations during inference

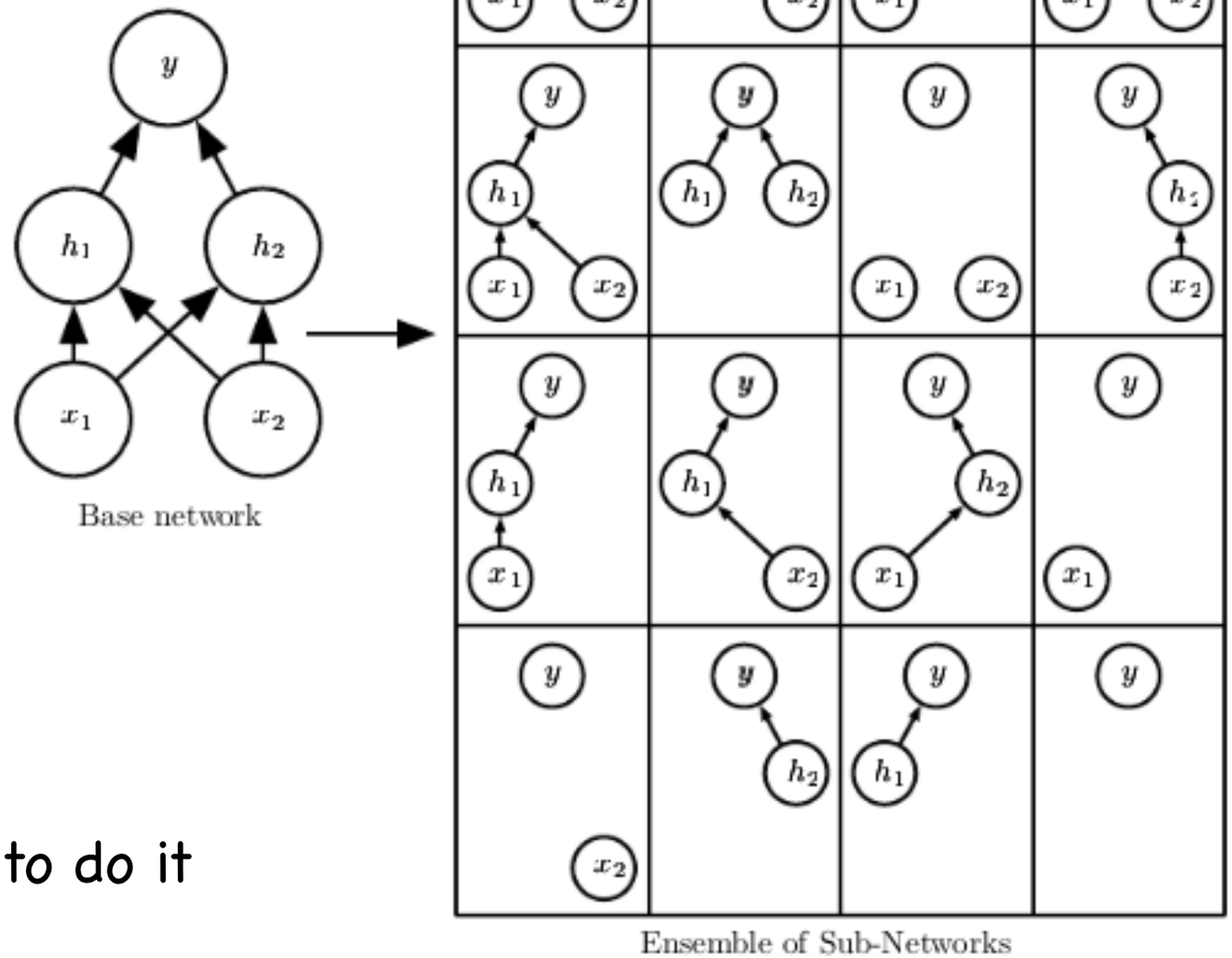


(a) Standard Neural Net (b) After applying dropout.

- Dropout has the effect of making the training noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.
- Dropout encourages the network to learn a sparse representation
- It breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust

Dropout

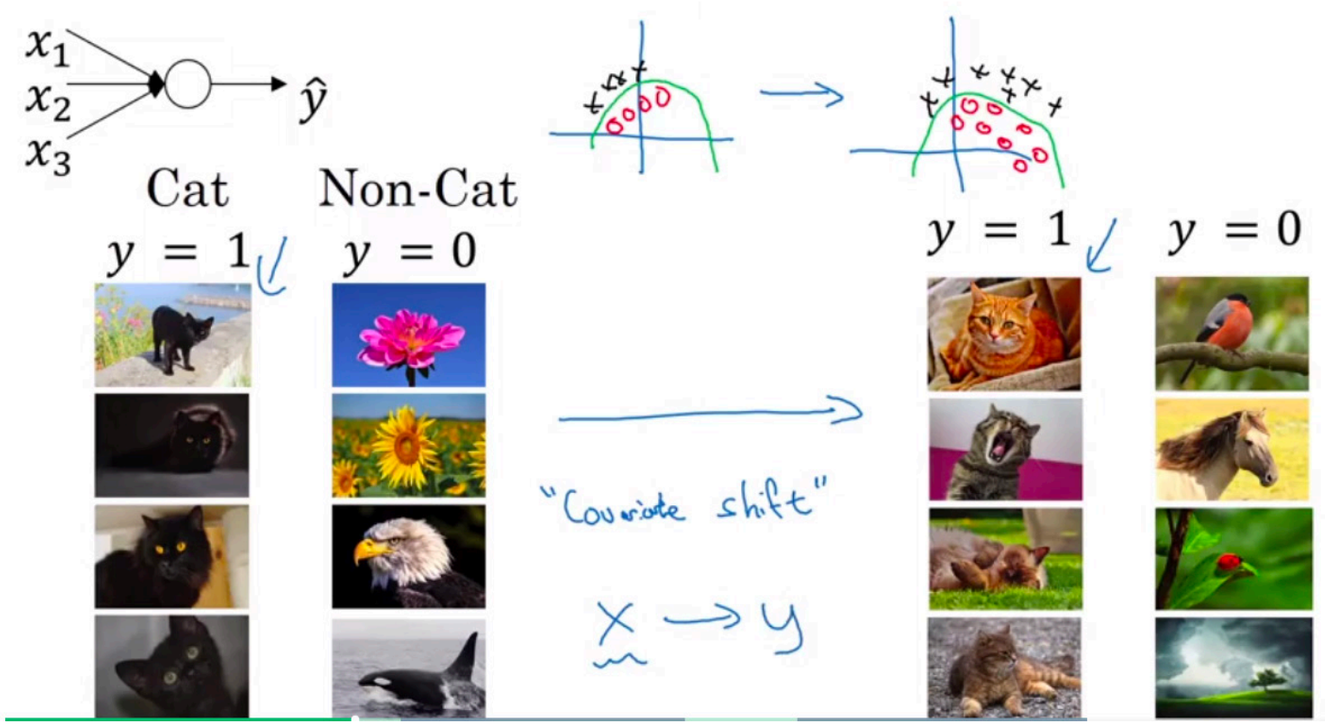
- Has ensemble effect



- Practical exercise on how to do it

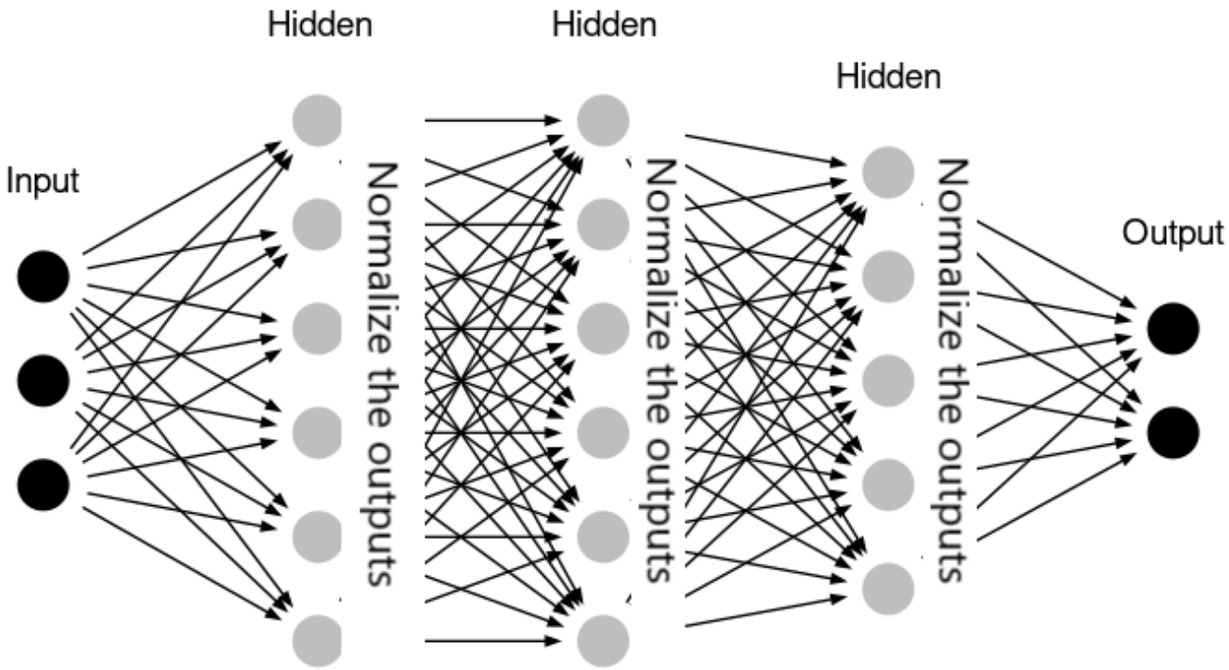
Batch Normalisation

- Internal Covariate Shift
- Normalising activations
- Has a little regularization effect



Source: Deeplearning.ai

Batch Normalisation



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

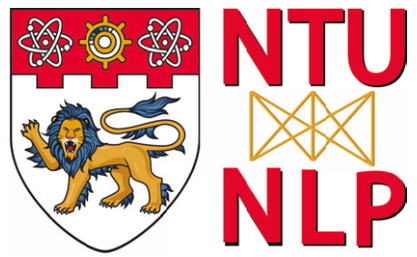
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Source: Ioffe & Szegedy

Why Exploding Gradient is a Problem



- If the gradient becomes too big, then the SGD update step becomes too big:

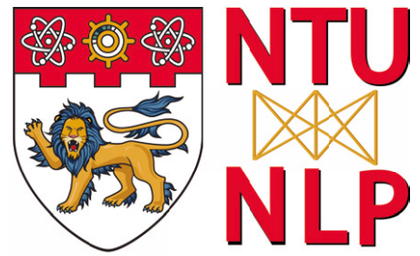
$$\theta^{new} = \theta^{old} - \underbrace{\overbrace{\alpha}^{\text{learning rate}} \nabla_\theta J(\theta)}_{\text{gradient}}$$

- This can cause bad updates: we take too large a step and reach a bad parameter configuration (with large loss)

- In the worst case, this will result in Inf or NaN in your network (then you have to restart training from an earlier checkpoint)

Source: Stanford 224n

Solution for Exploding Gradient



Gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

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

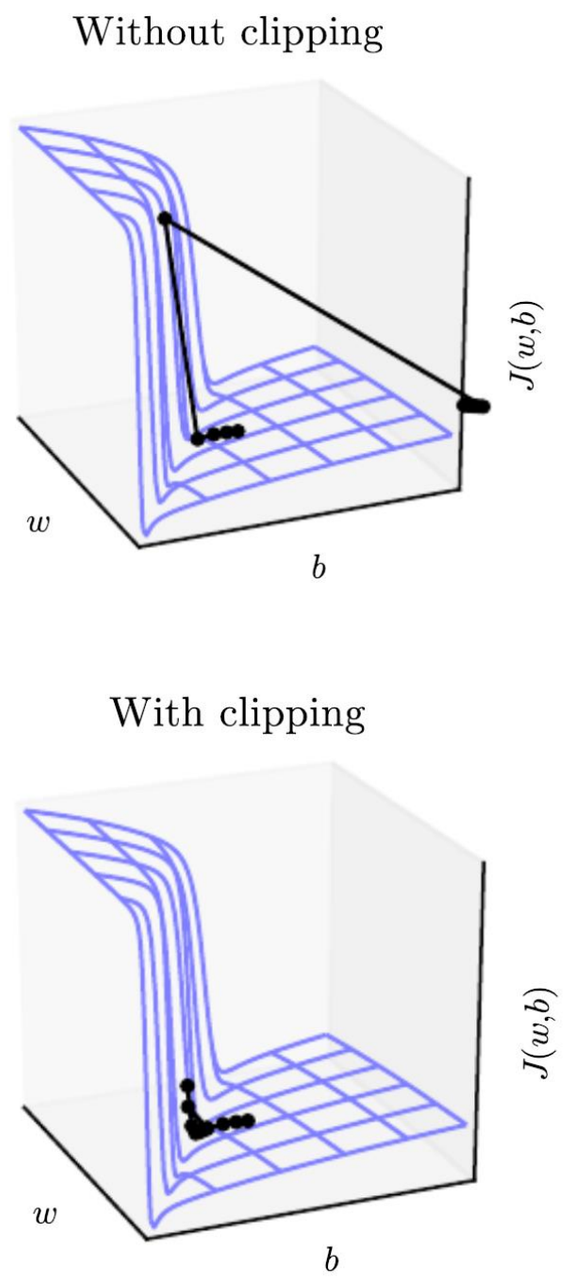
- Intuition: take a step in the same direction, but a smaller step

""On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf

Source: Stanford 224n

Solution for Exploding Gradient

- This shows the loss surface of a simple RNN (hidden state is a scalar not a vector)
- The "cliff" is dangerous because it has steep gradient
- In the Fig. at the top, gradient descent takes two very big steps due to steep gradient, resulting in climbing the cliff then shooting off to the right (both bad updates)
- At the bottom, gradient clipping reduces the size of those steps, so effect is less drastic



Source: Stanford 224n

Initialisation

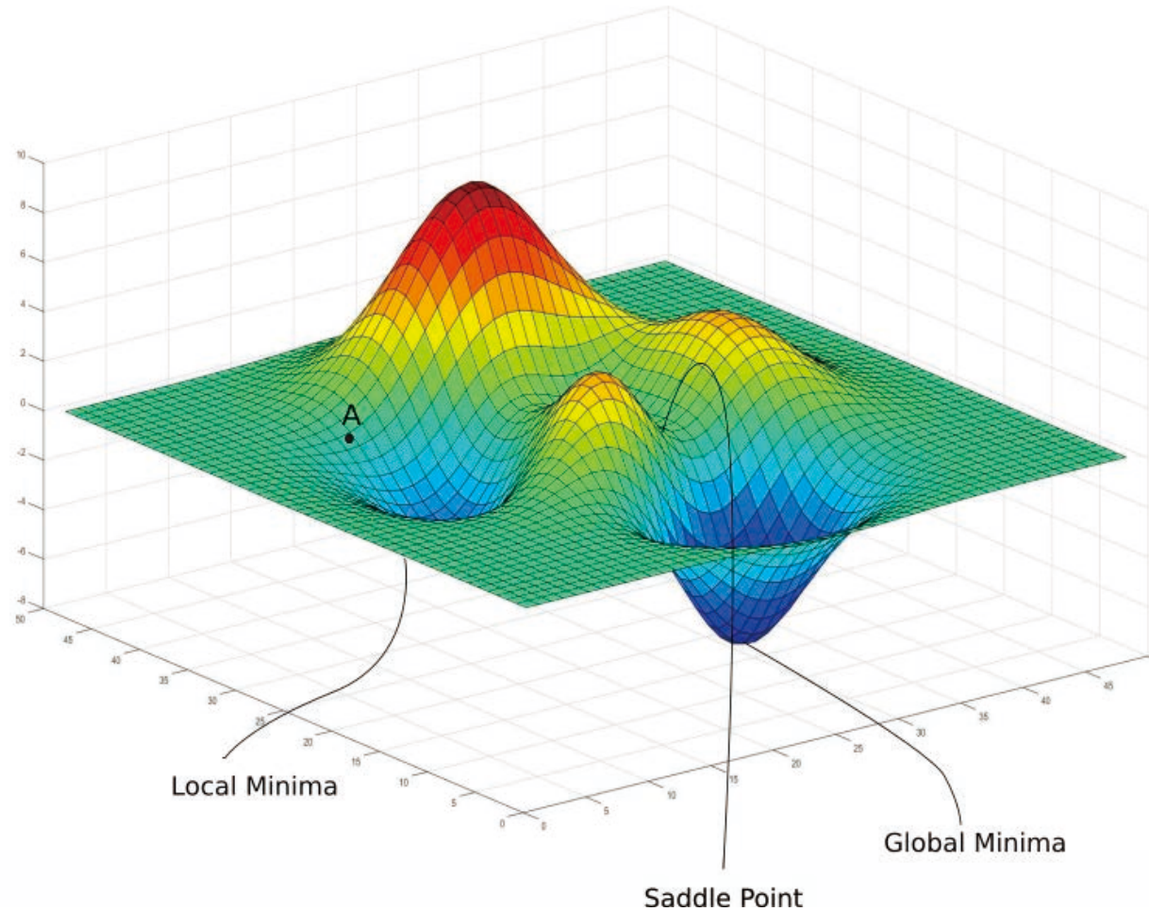
NN objectives are not convex — many local minima

Stochastic gradient descent for NN has no convergence guarantee

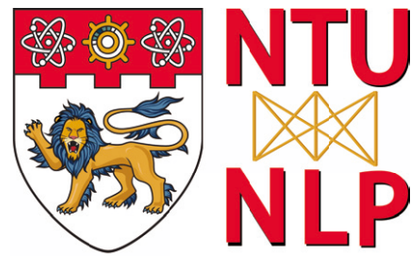
Sensitive to the values of the initial parameters.

- Good initialization speeds up network training speed

Is important to initialize all weights to small random values.



Initialisation



LeCun Initialization

- $Y = WX + B$
- $y = w_1x_1 + w_2x_2 + ... + w_nx_n + b$
- $Var(y) = Var(w_1x_1 + w_2x_2 + ... + w_nx_n + b)$
- $Var(y) = nVar(w_i)Var(x_i)$

We want the variance of the input to be equal to output variance.

- $Var(w_i) = \frac{1}{n}$

Initialisation

Xavier & He Initialisation

We want the variance of the input to be equal to output variance.

- $Var(w_i) = \frac{1}{n}$

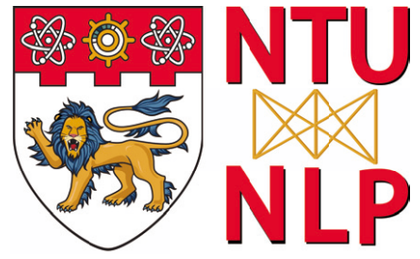
Sigmoid or Hyperbolic tangent

$$n = \frac{n_{in} + n_{out}}{2}$$

ReLU, or LeakyReLU

$$n = \frac{2}{n_{in}}$$

Lecture Plan



- Why Deep Learning for NLP
- From Logistic/Linear Regression to NN
  - Activation functions
- SGD with Backpropagation
- Adaptive SGD (adagrad, adam, RMSProp)
- Regularisation (dropout, gradient clipping)
- Introduction to word vectors

Word Meaning

- The idea that is represented by a word, phrase, etc.
- How do we represent it in a computer

Common solution: Use e.g. WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** ("is a" relationships).

e.g. synonym sets containing "good":

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
            ", ".join([l.name() for l in synset.lemmas()])))
```

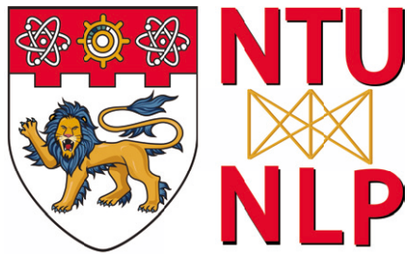
```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
…
adverb: well, good
adverb: thoroughly, soundly, good
```

e.g. hypernyms of "panda":

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

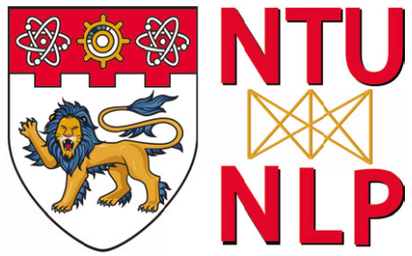
Problems with WordNet



- Great as a lexical resource but missing nuance
  - e.g. "proficient" is listed as a synonym for "good". This is only correct in some contexts.
- Missing new meanings of words, e.g., wicked, badass, nifty, wizard, genius, ninja, bombest
- Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt

Source: Stanford 224N

Problems with Discrete Representation



- Hard to compute accurate word similarity

- In traditional NLP, we regard words as discrete symbols: hotel, conference, motel

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

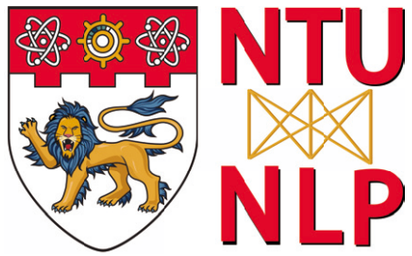
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

These two vectors are orthogonal.

- Could try to rely on WordNet's list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.

Source: Stanford 224N

Representing words by their context



- **Instead: learn to encode similarity in the vectors themselves**

- Distributional semantics: **A word's meaning is given by the words that frequently appear close-by**
  - "You shall know a word by the company it keeps"

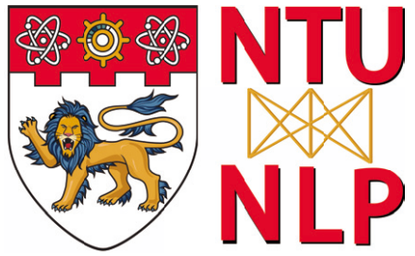
(J. R. Firth 1957: 11)

- When a word w appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).

- Use the many contexts of w to build up a representation of w

Source: Stanford 224N

Representing words by their context



- When a word w appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).

- Use the many contexts of w to build up a representation of w

…government debt problems turning into	***banking***	*crises as happened in 2009…*
…saying that Europe needs unified	***banking***	*regulation to replace the hodgepodge…*
…India has just given its	***banking***	*system a shot in the arm…*

These context words will represent ***banking***

Word Vectors

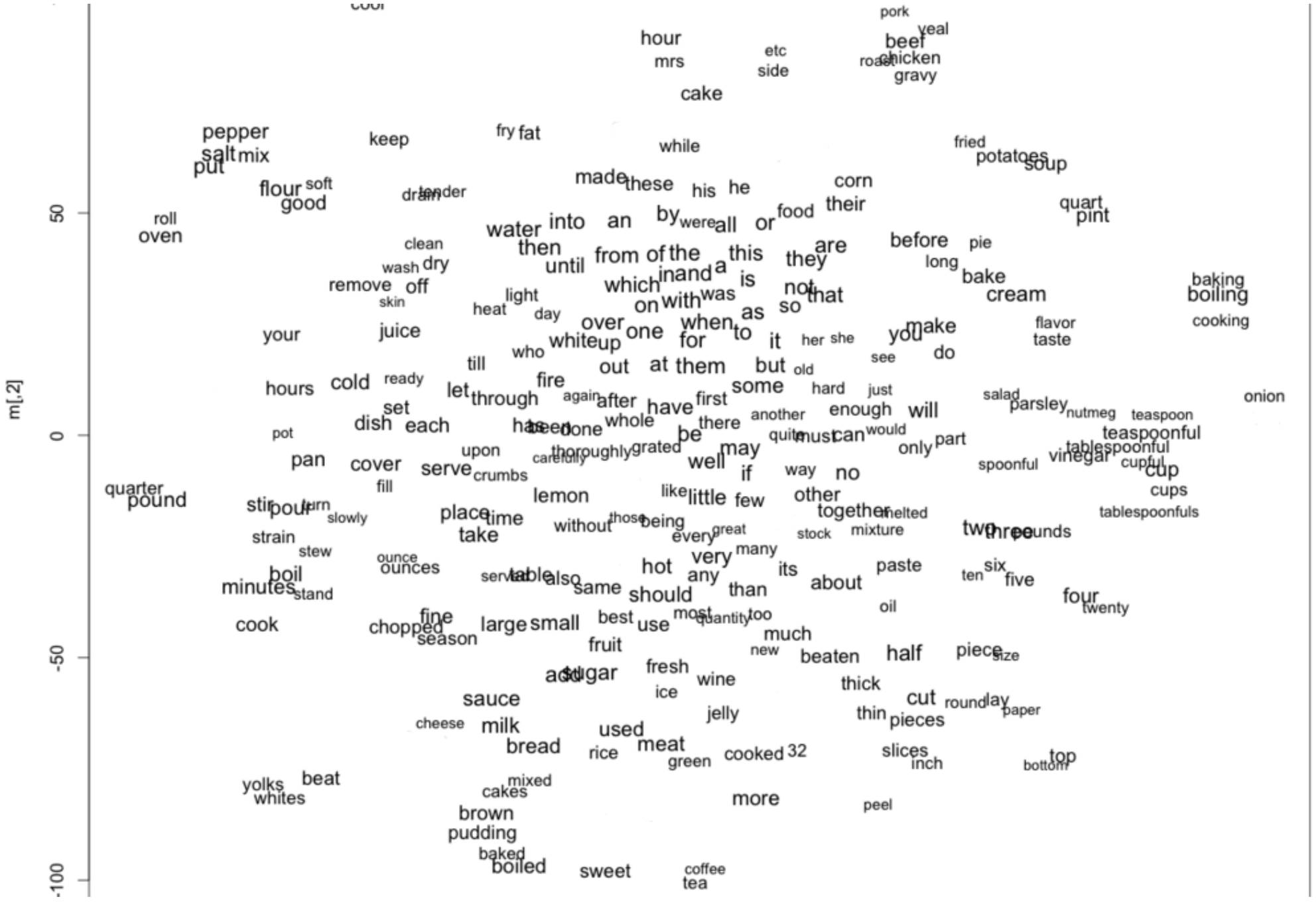
We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts

$$\textit{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

word vectors are sometimes called word embeddings or word representations.
They are a distributed representation.

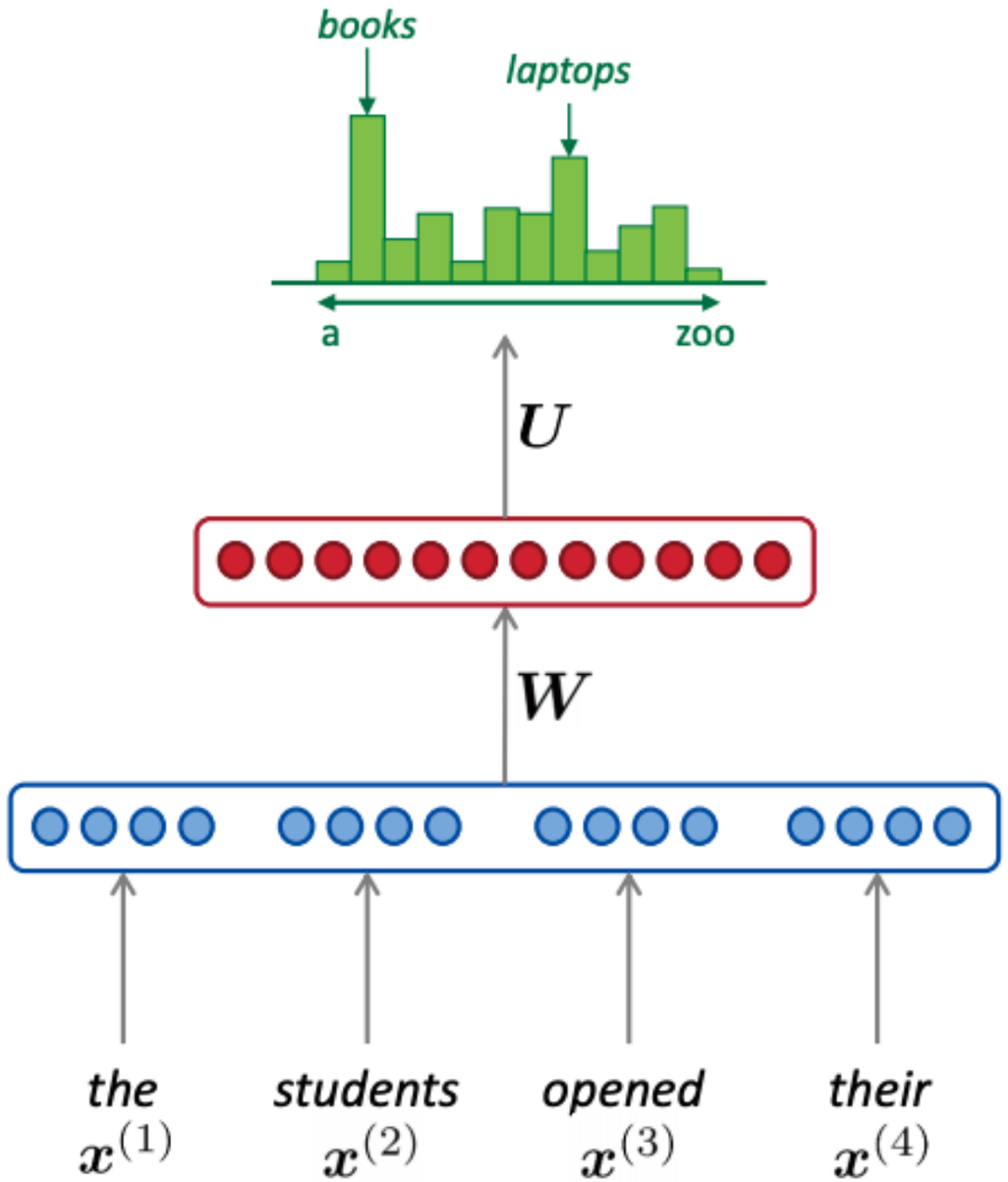
Source: Stanford 224N

Word Vectors



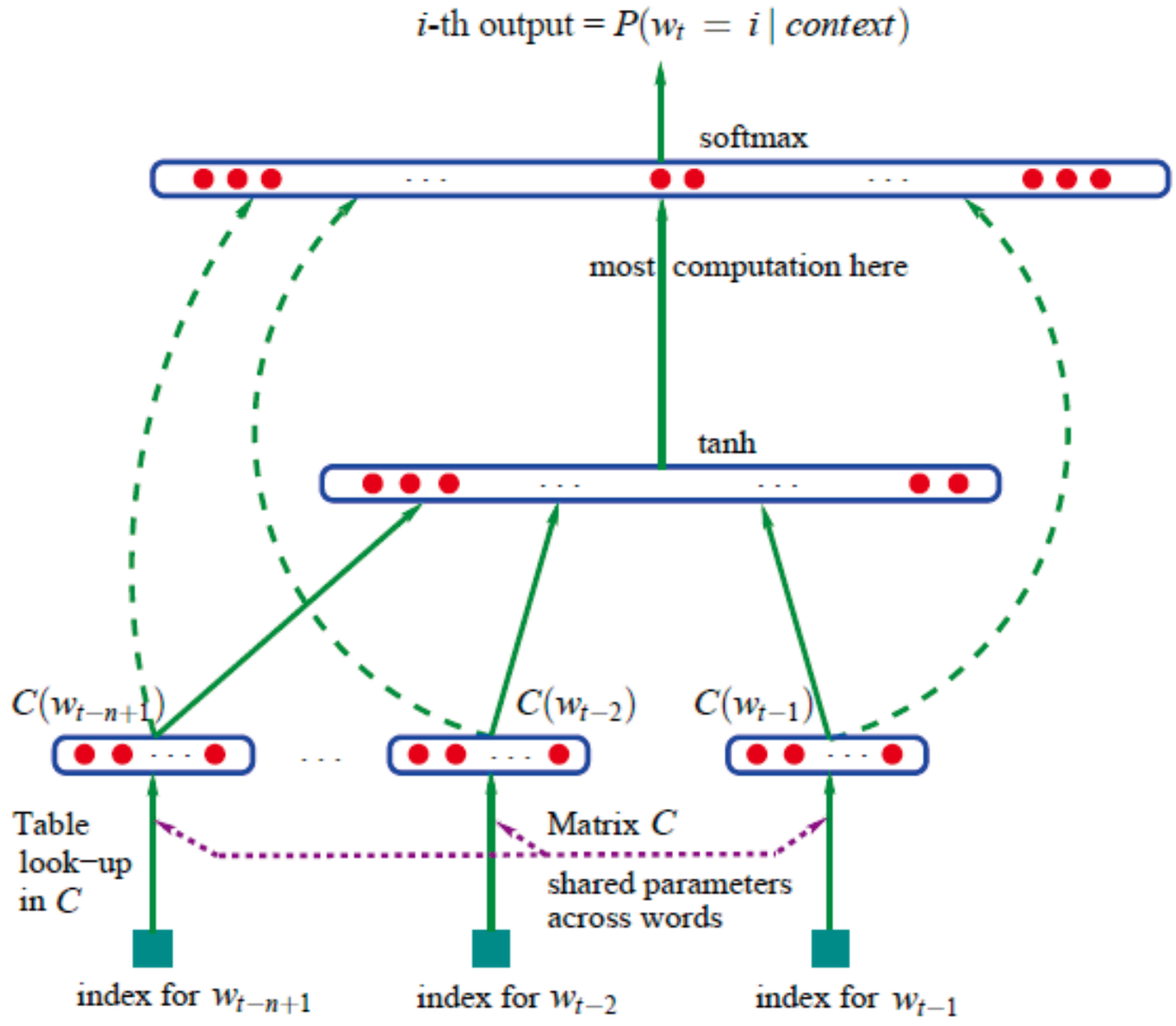
Language Model

A language model takes a list of words (history), and attempts to predict the word that follows them



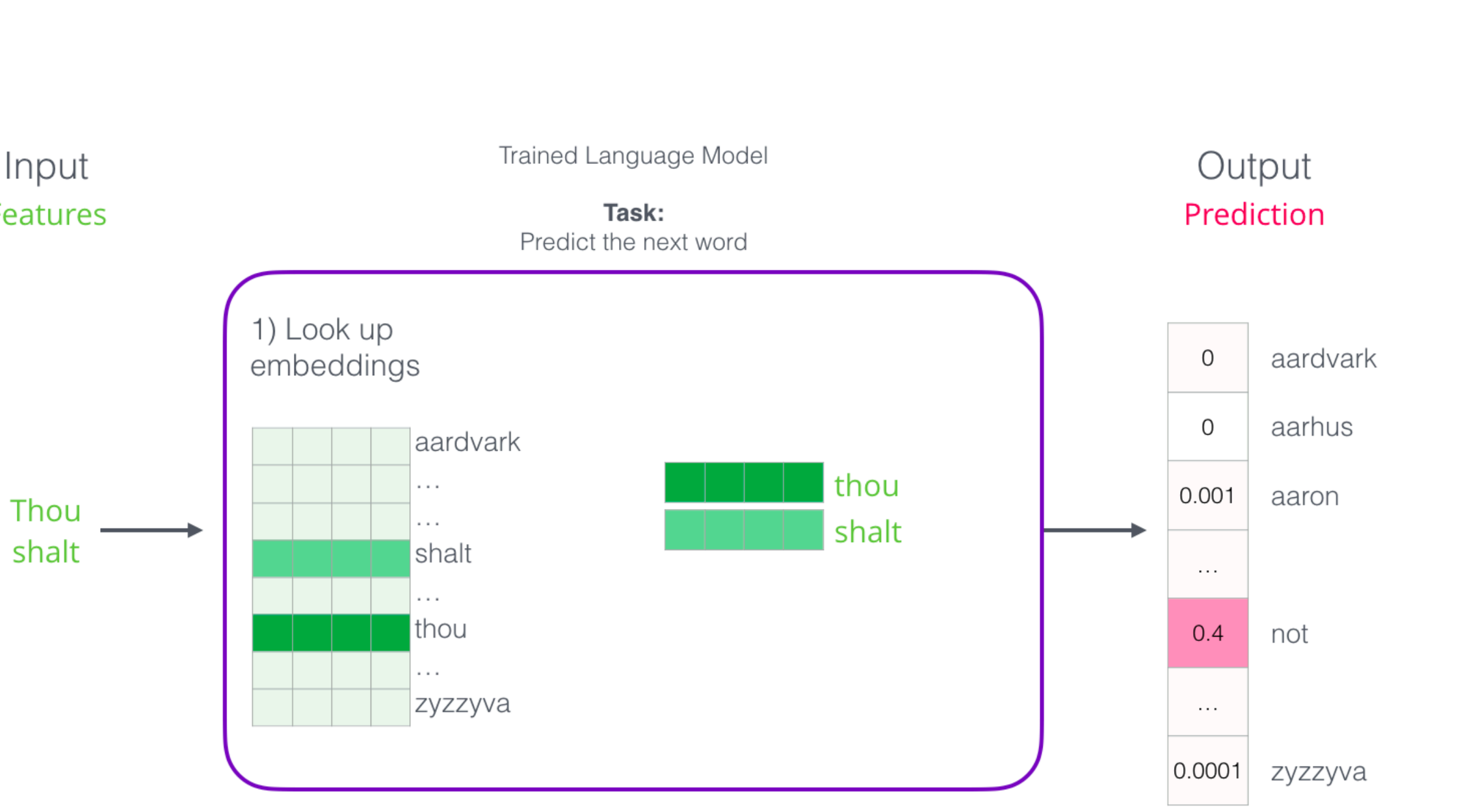
Language Model

A language model takes a list of words (history), and attempts to predict the word that follows them



Source: Bengio, 2003

Language Model

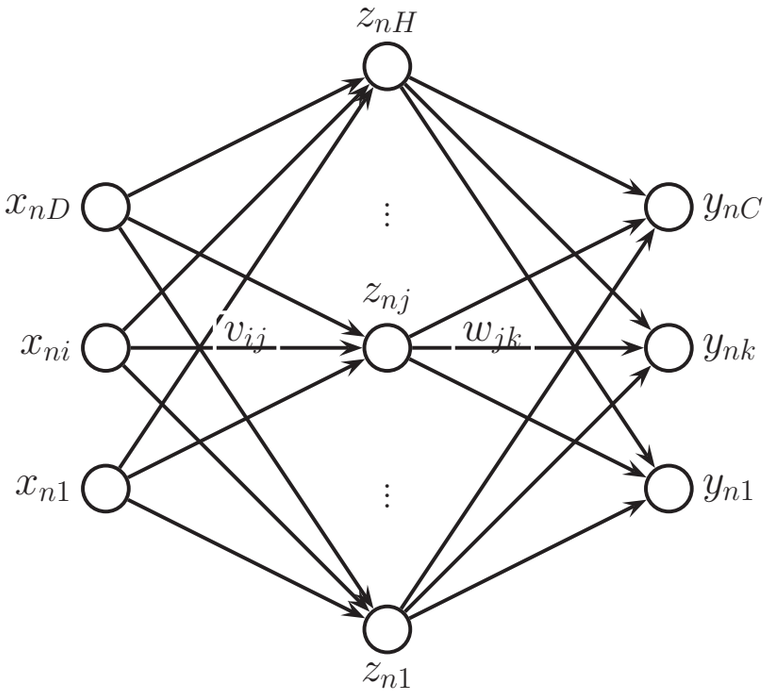


Source:http://jalammar.github.io/

Summary

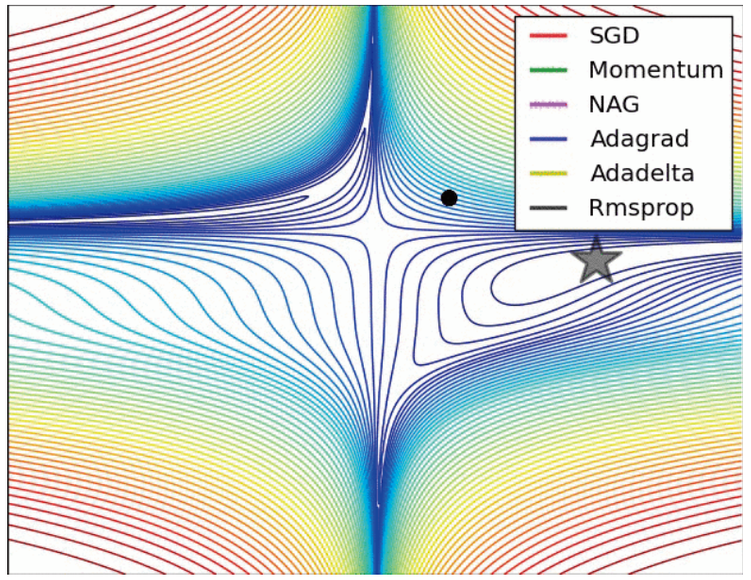
NTU NLP

MLP

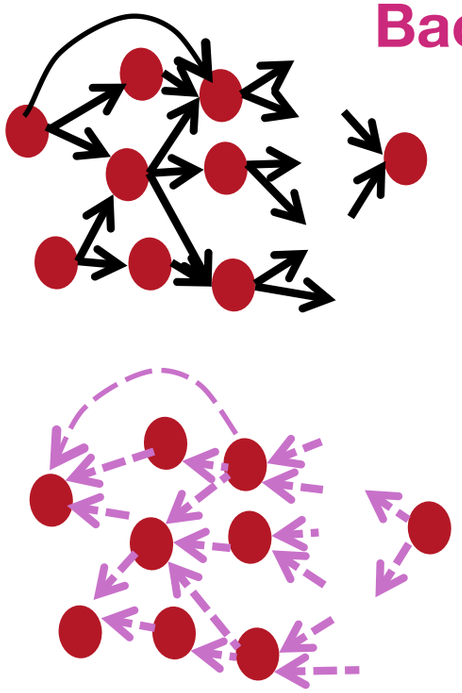


Adaptive SGD

$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$$



BackProp



Word Vectors and LM