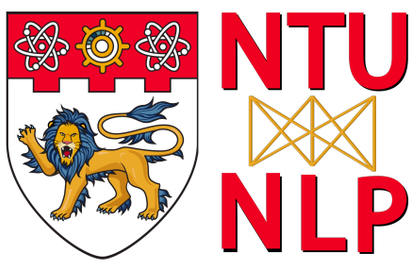
Deep Learning for Natural Language Processing

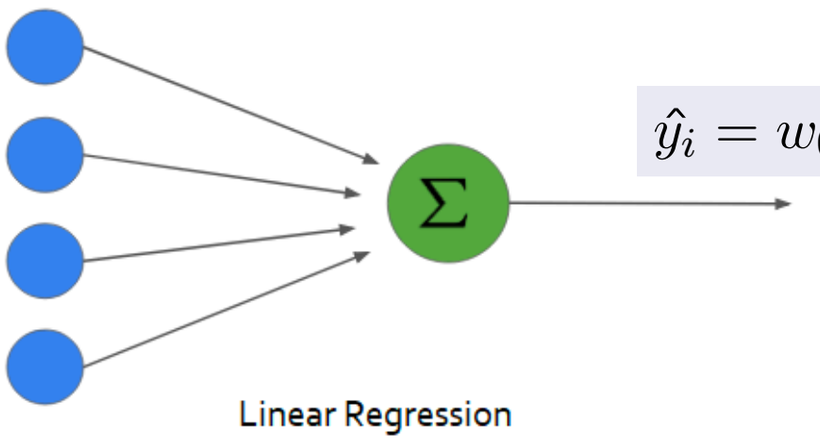
CE/CZ 4045

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

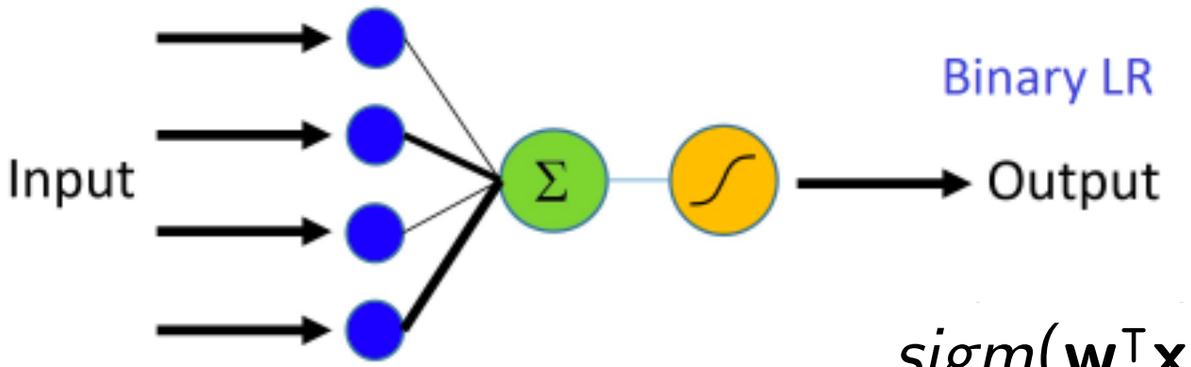
$$RSS(\mathbf{w}) \triangleq \sum_{i=1}^{N} (y_i - \mathbf{w}^\intercal \mathbf{x}_i)^2$$

Closed form solution

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

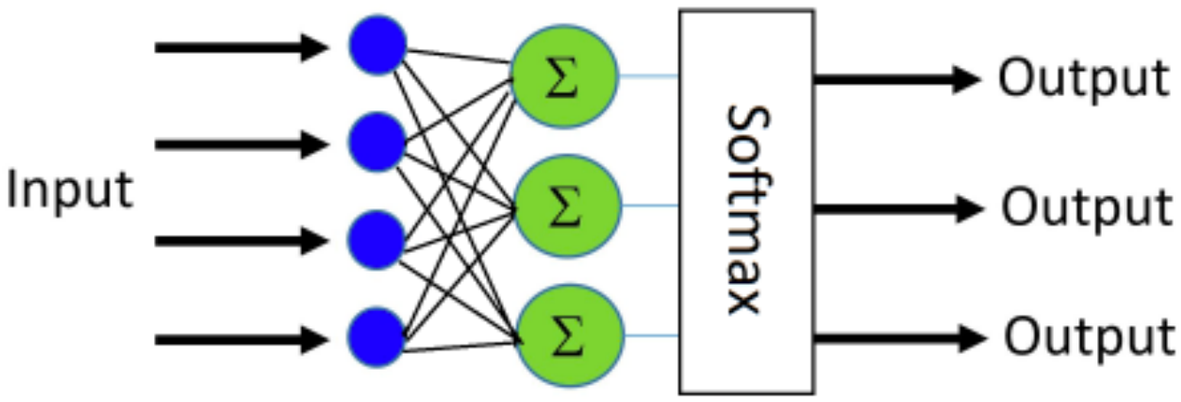
Linear Models (Recap)

NTU NLP



Binary LR

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



Multiclass LR

Softmax $$\frac{\exp(\mathbf{w}_c^\intercal \mathbf{x})}{\sum_{c'=1}^{C} \exp(\mathbf{w}_{c'}^\intercal \mathbf{x})}$$

SGD

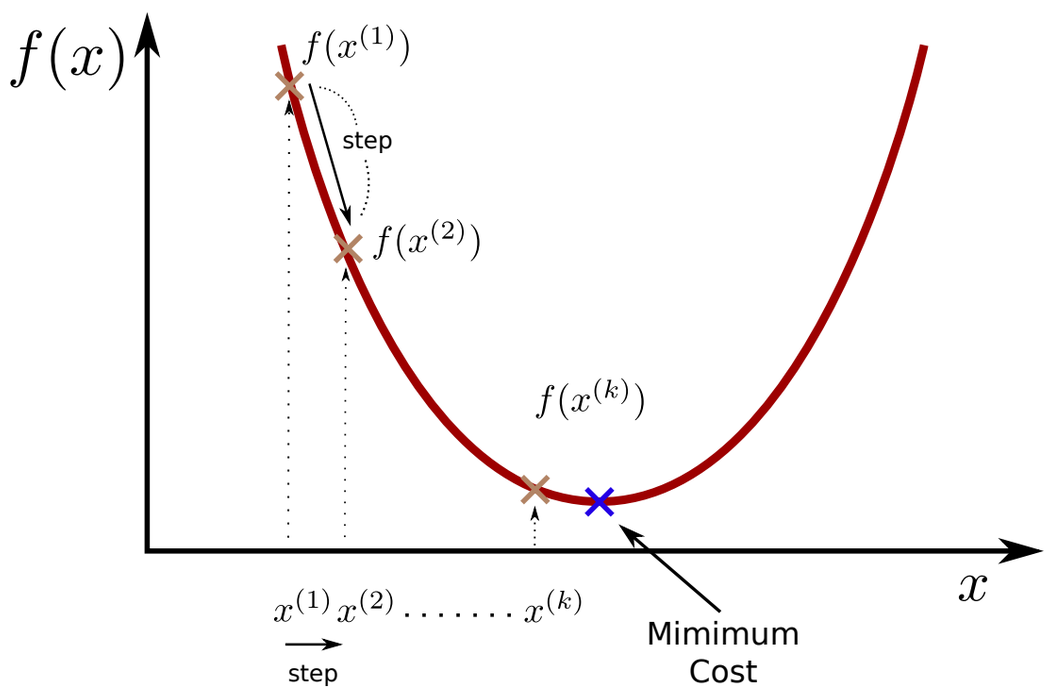


Figure: Illustration of gradient descent

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

Traditional Machine Learning

NTU NLP

- 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)

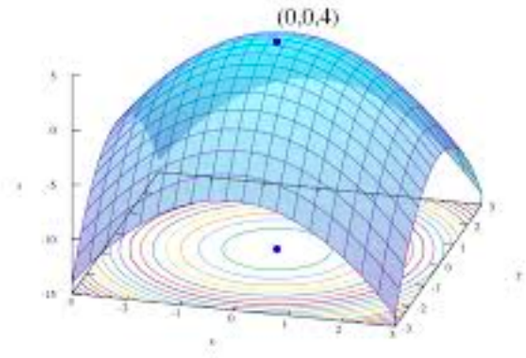
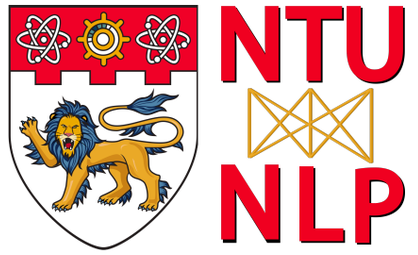


Figure 1: An example of NER application on an example text

Lecture Plan



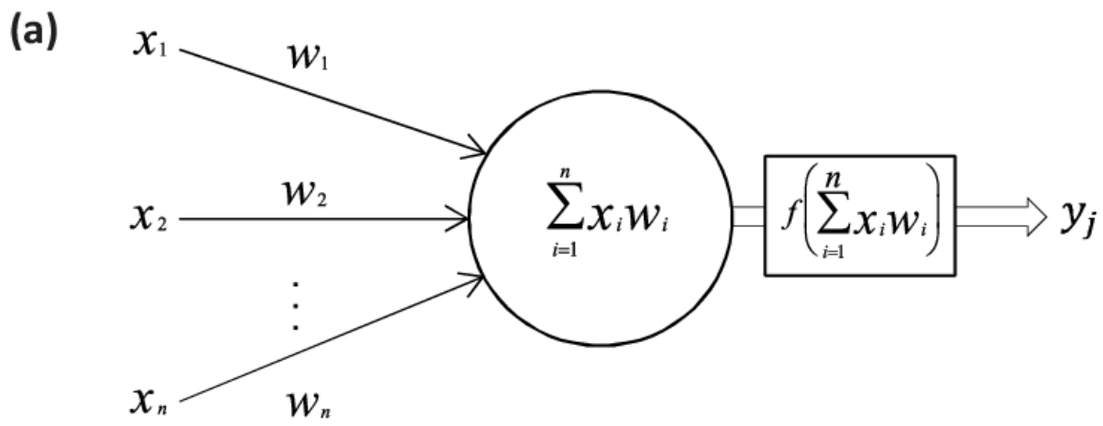
- From Logistic/Linear Regression to NN
  - Activation functions
- SGD with Backpropagation
- Regularisation (dropout, gradient clipping)

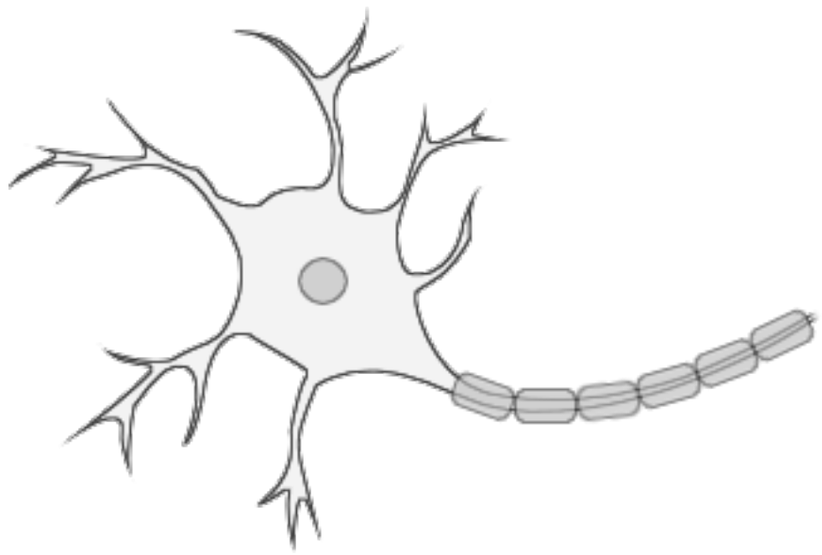
Neurons

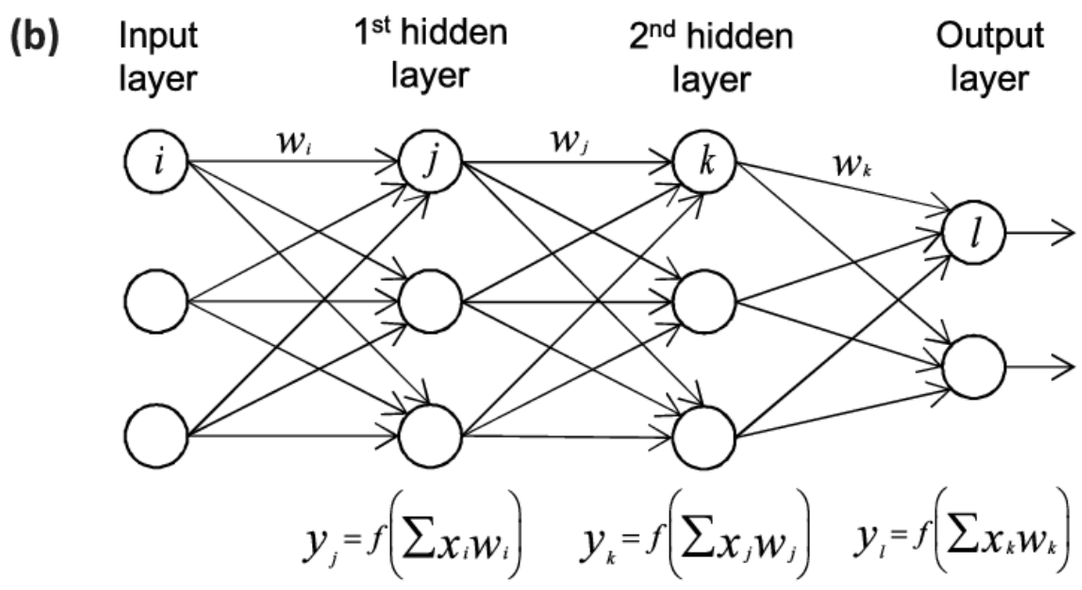
NTU NLP

if you understand how logistic regression works

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







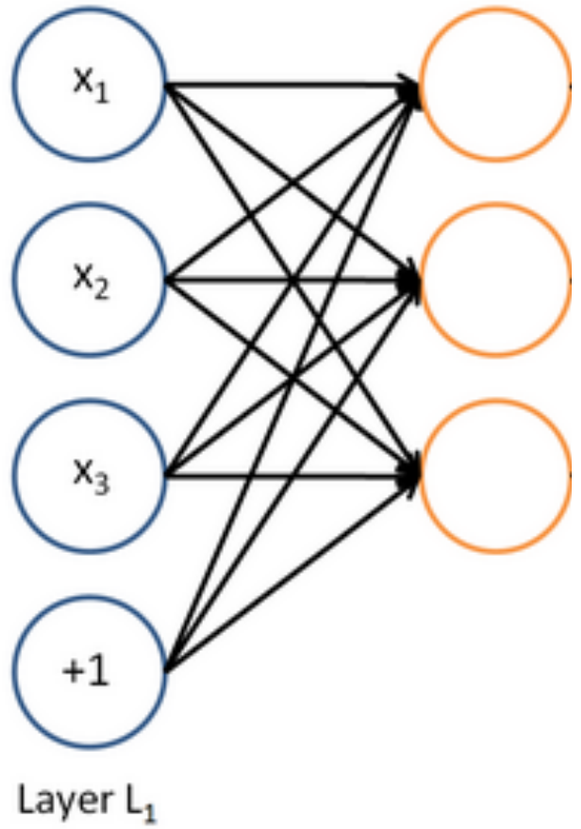
From LR to NN

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 …

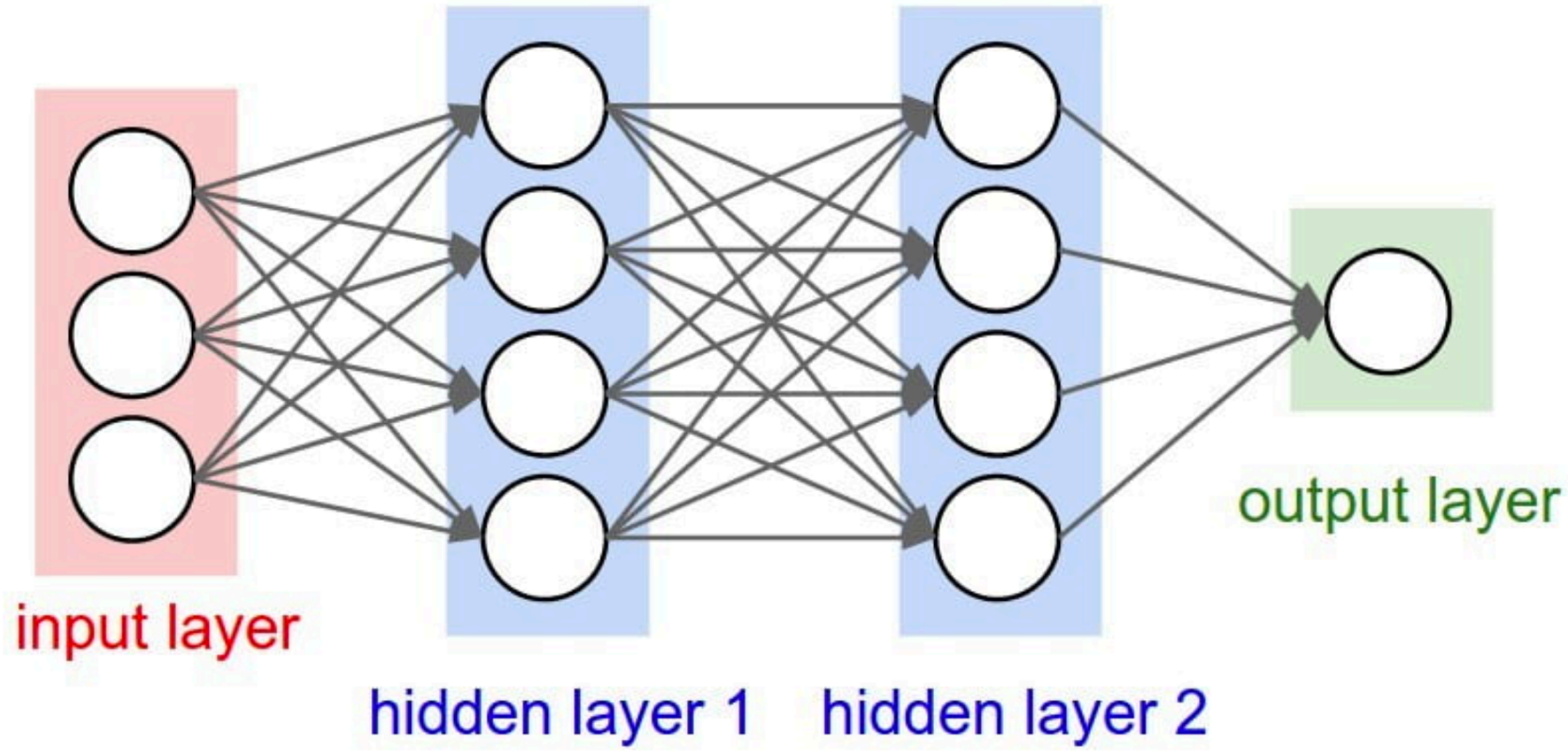
But these are not random variables



From LR to NN

NTU NLP

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

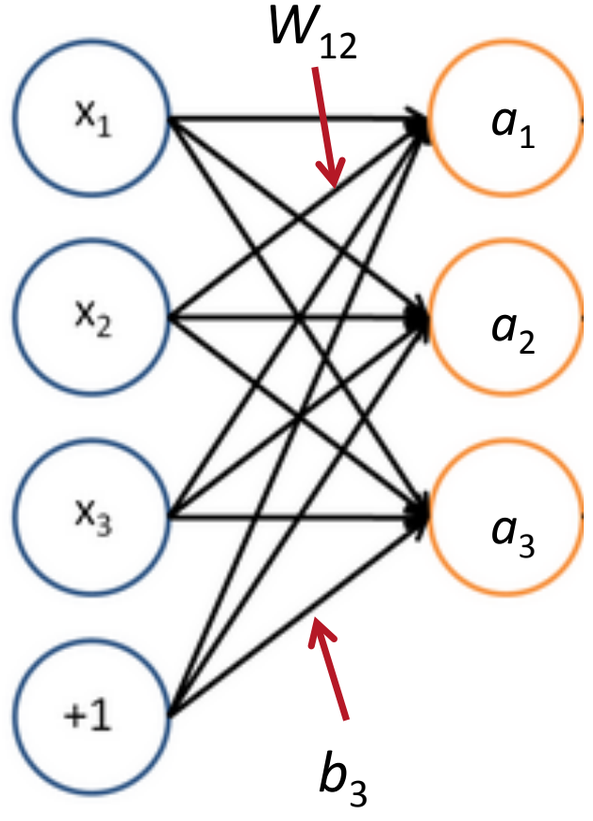


Output layer can be a classification layer or a regression layer

From LR to NN

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

.....



In Matrix Notation

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

Where f() is applied element-wise

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

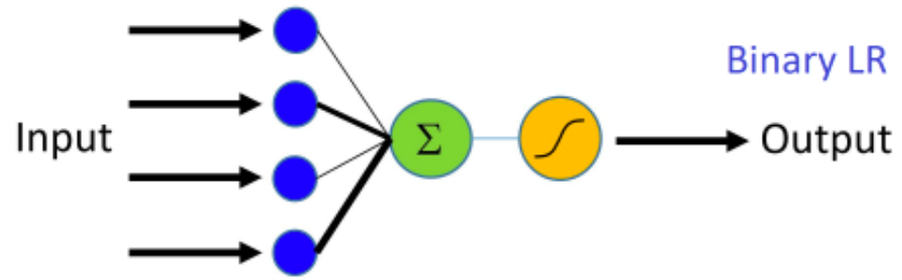
How Do We Train?

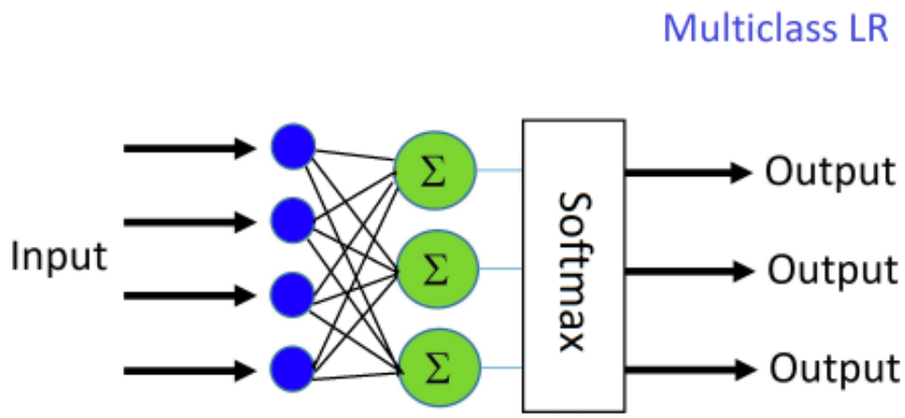
NTU NLP

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?

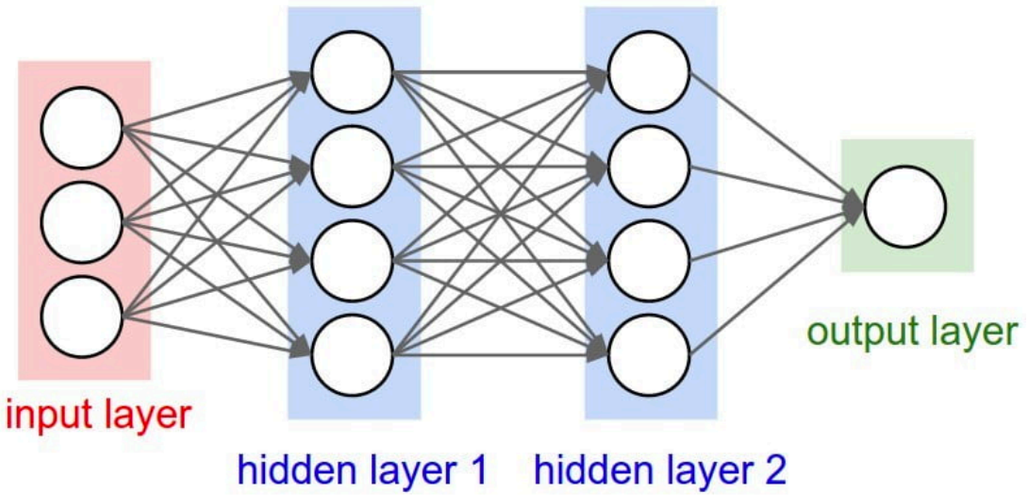
NTU NLP

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**
    $\mathbf{g} = \nabla f(\theta, \mathbf{z}_i)$
    $\theta \leftarrow \mathit{proj}_{\Theta}(\theta - \eta \mathbf{g})$
    Update η
  end for
until *converged*

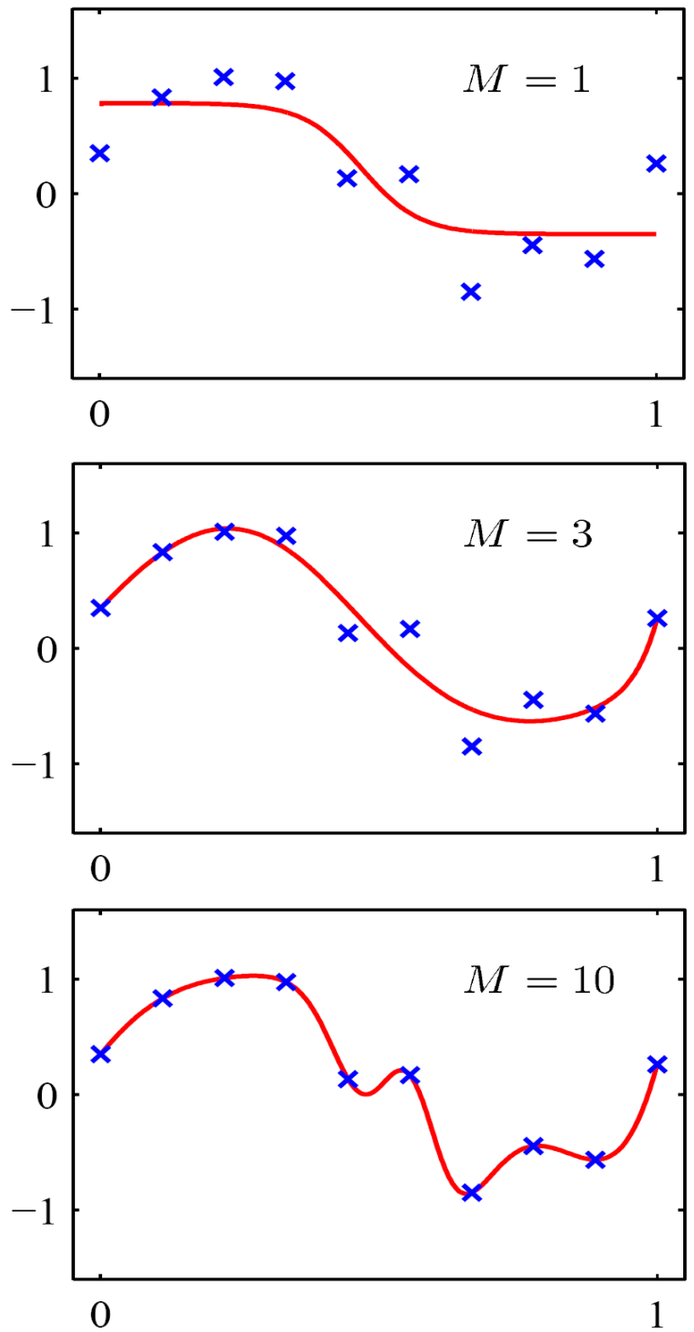
Why Non-Linearity?

- For logistic regression: map to probabilities

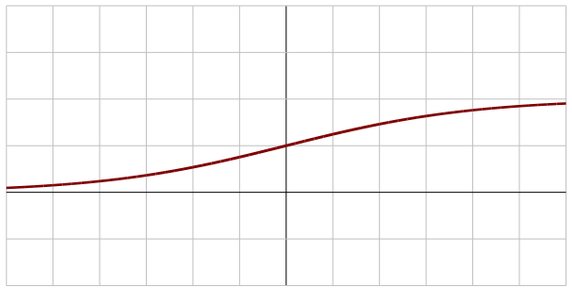
- For NN: function approximation, e.g., regression or classification

  - 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

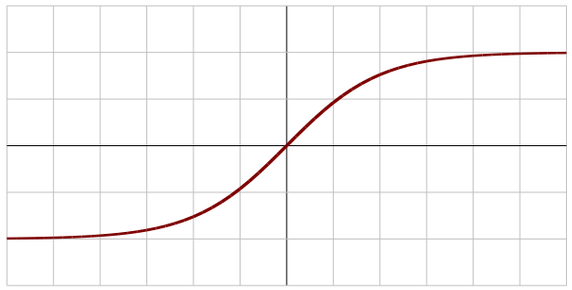
$sigm(\mathbf{w}^T\mathbf{x})$



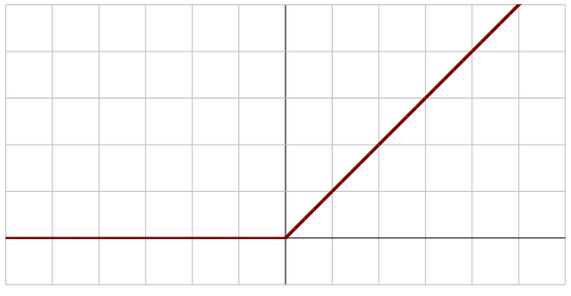
Activation Functions



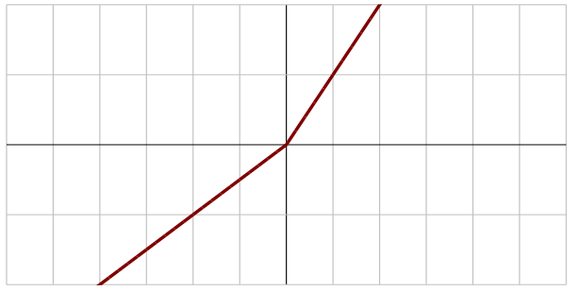
(a) Sigmoid



(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):

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

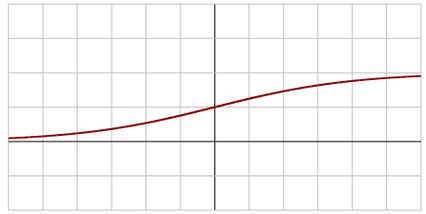
- Leaky ReLU:

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

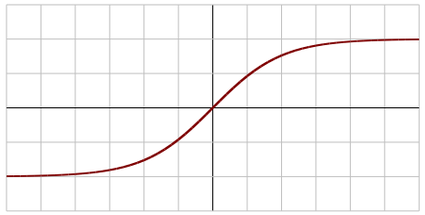
, where α is small constant

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

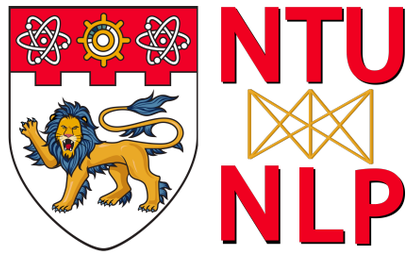


(b) tanh

(c) ReLU

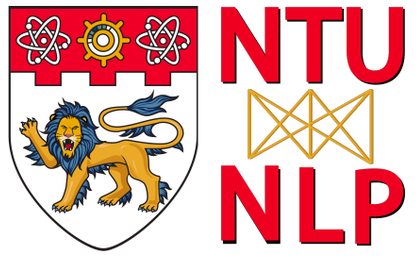
(d) Leaky ReLU

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



- From Logistic/Linear Regression to NN
  - Activation functions
- SGD with Backpropagation
- Regularisation (dropout, gradient clipping)

Gradient Descent (Recall)

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

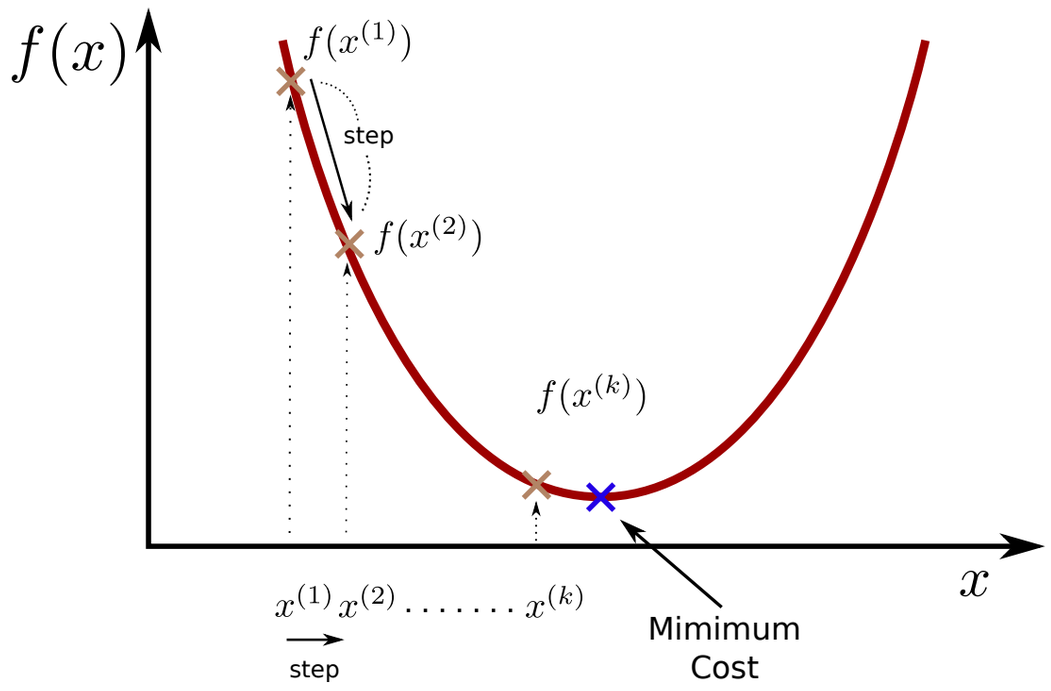


Figure: Illustration of gradient descent

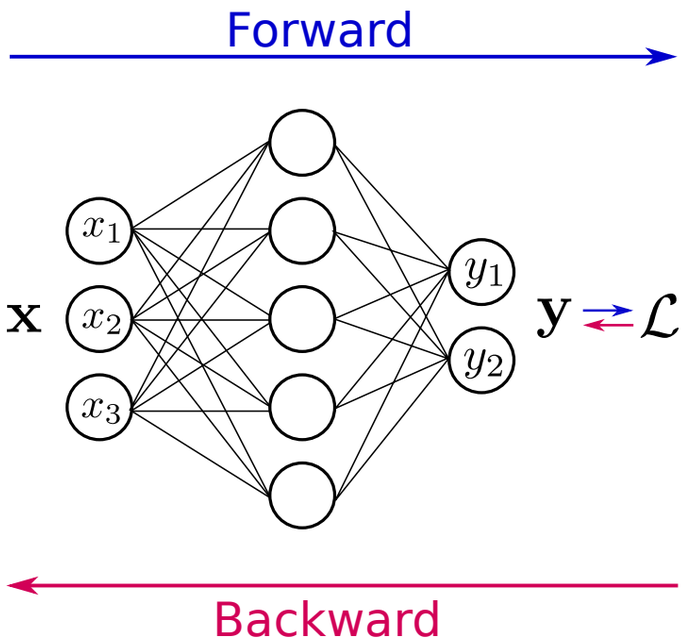
Forward- & Back-propagation

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

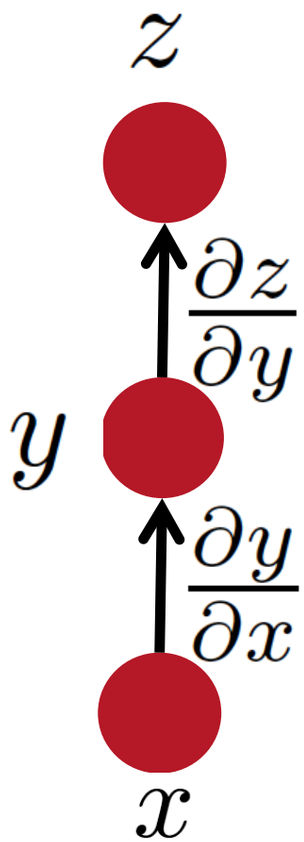
Chain Rule of Derivatives

NTU NLP

Compute gradient of example-wise loss wrt parameters

- Simply apply the chain rule of derivative

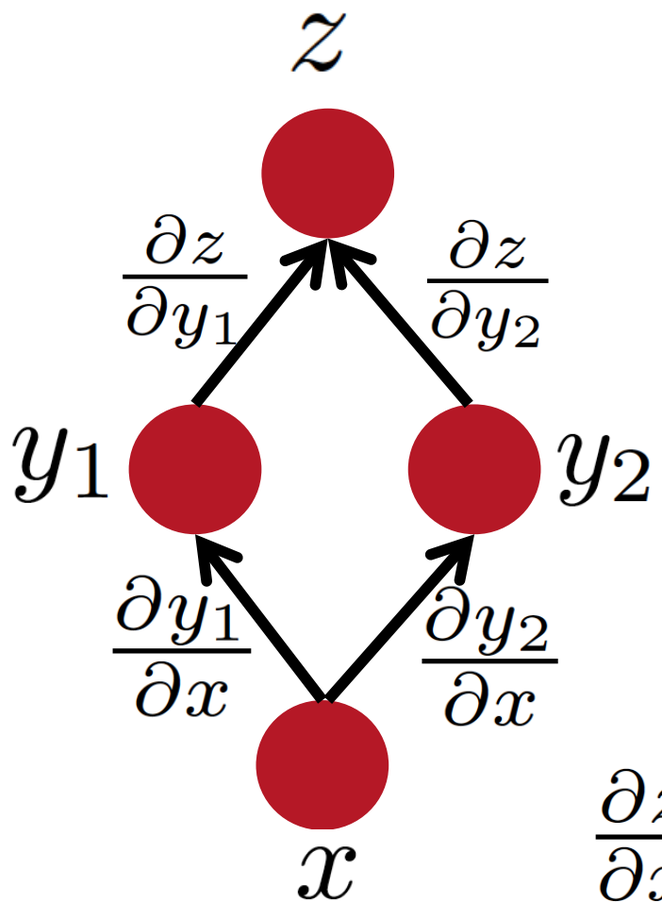
$$z = f(y) \quad y = g(x) \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$



Chain Rule of Derivatives

NTU NLP

- Multiple-path chain rule

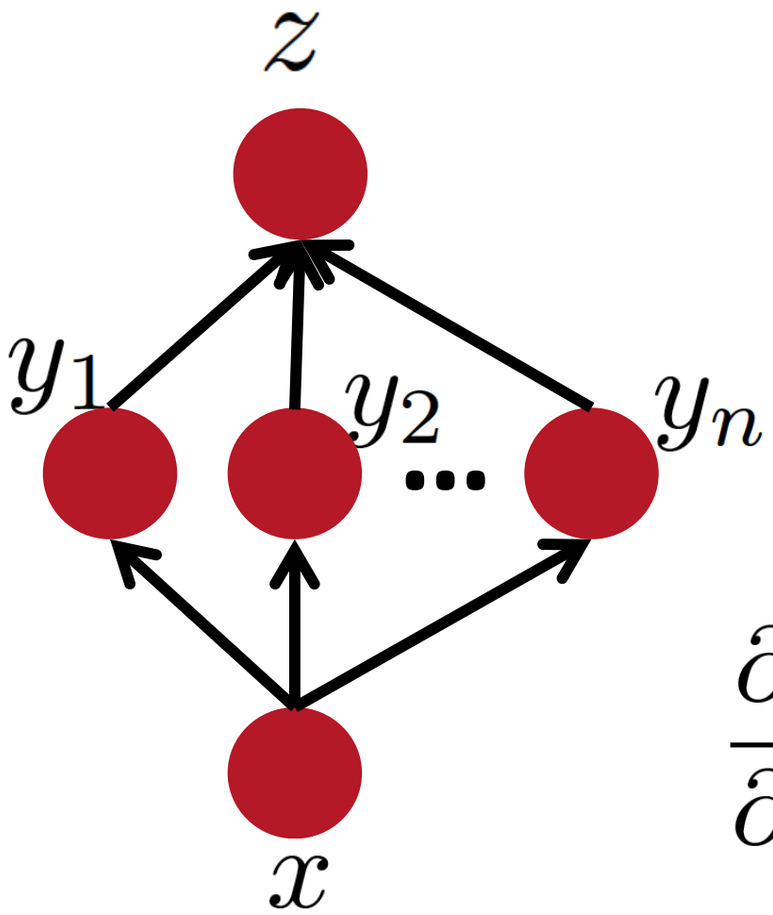


$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Chain Rule of Derivatives

NTU NLP

- Multiple-path chain rule (general)



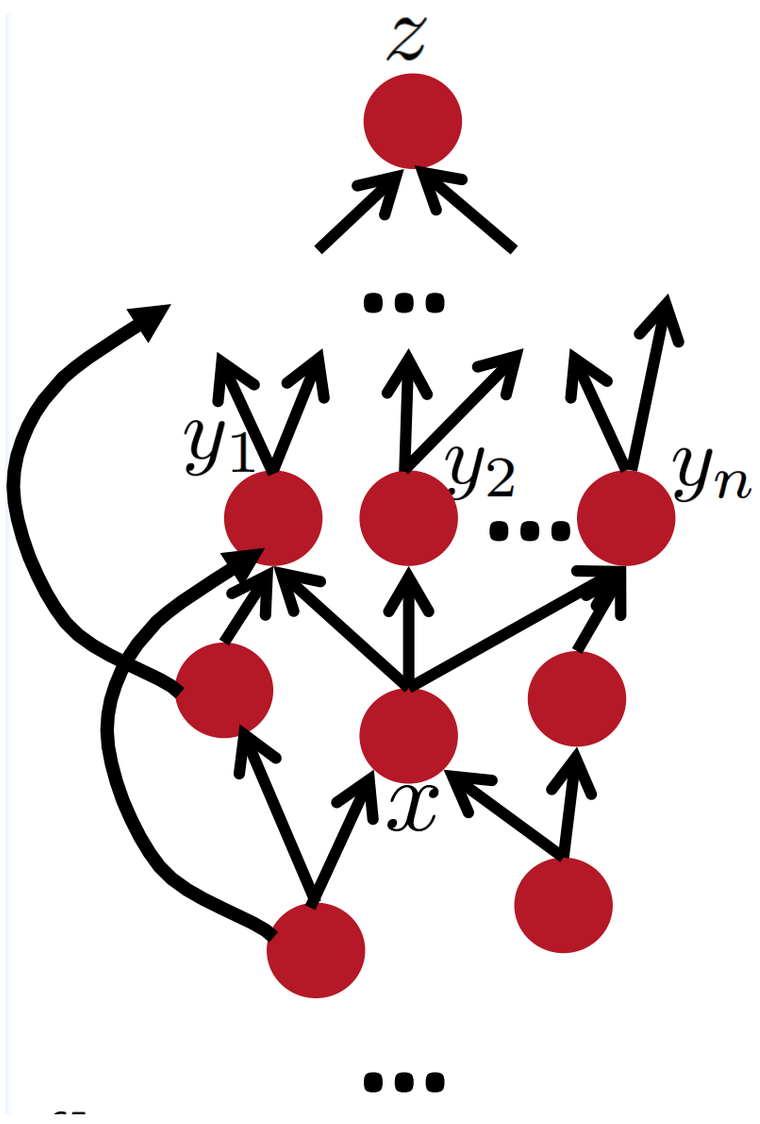
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

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

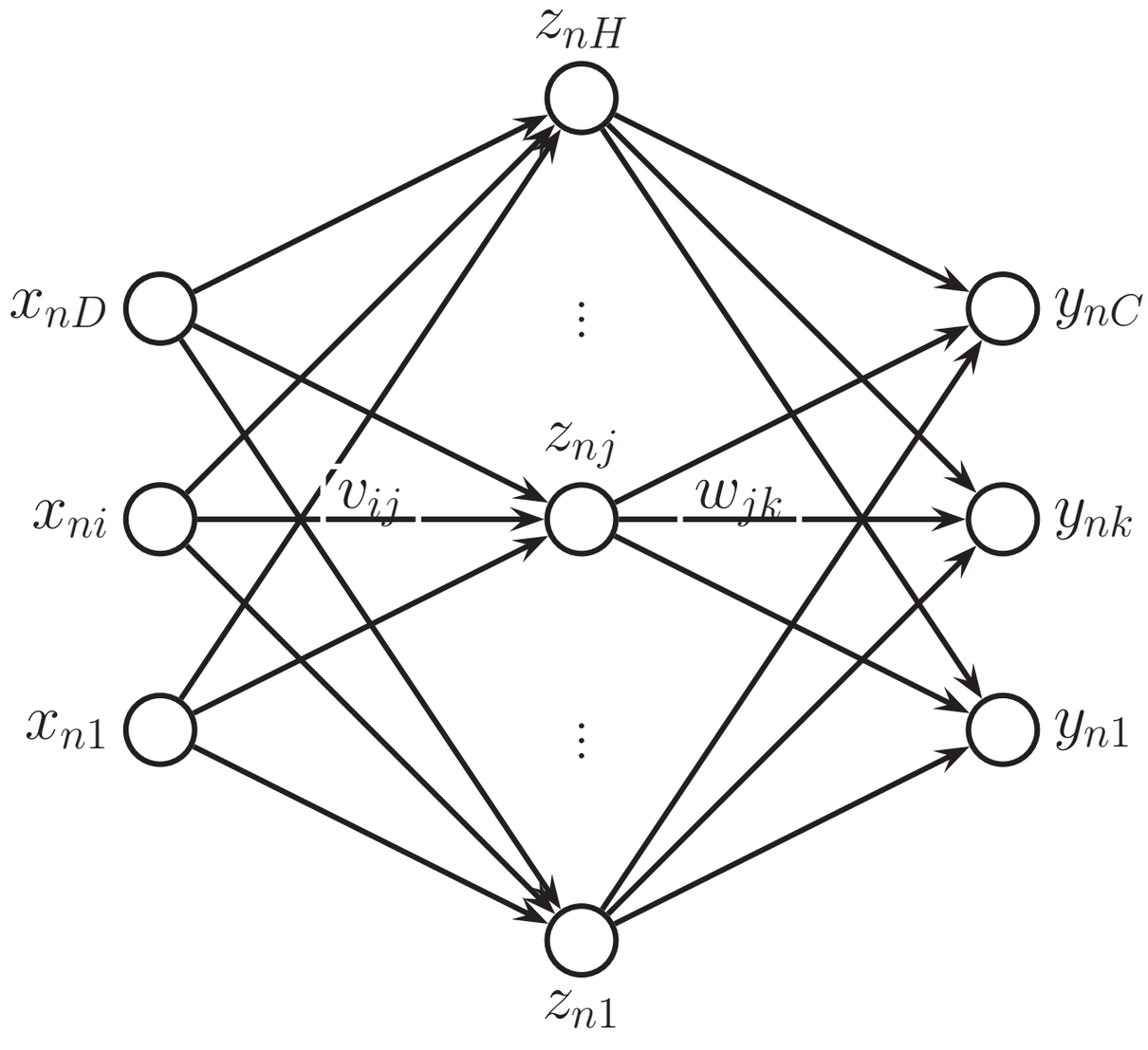
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Source: Socher & Manning

Back-propagation Derivation

- Single Layer NN

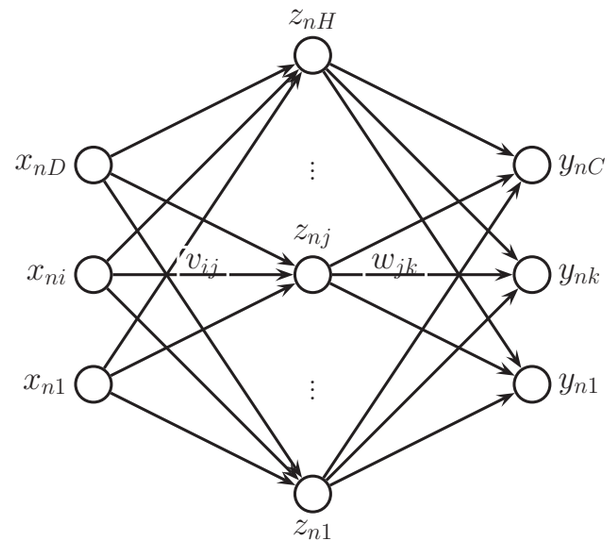
$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n$$



Back-propagation Derivation

- Gradient of the loss at the last layer

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



$$J(\boldsymbol{\theta}) = -\sum_n \sum_k y_{nk} \log \hat{y}_{nk}(\boldsymbol{\theta})$$ where $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W})$

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

$$\nabla_{\mathbf{w}_k} J_n = \frac{\partial J_n}{\partial b_{nk}} \nabla_{\mathbf{w}_k} b_{nk}$$ Chain rule

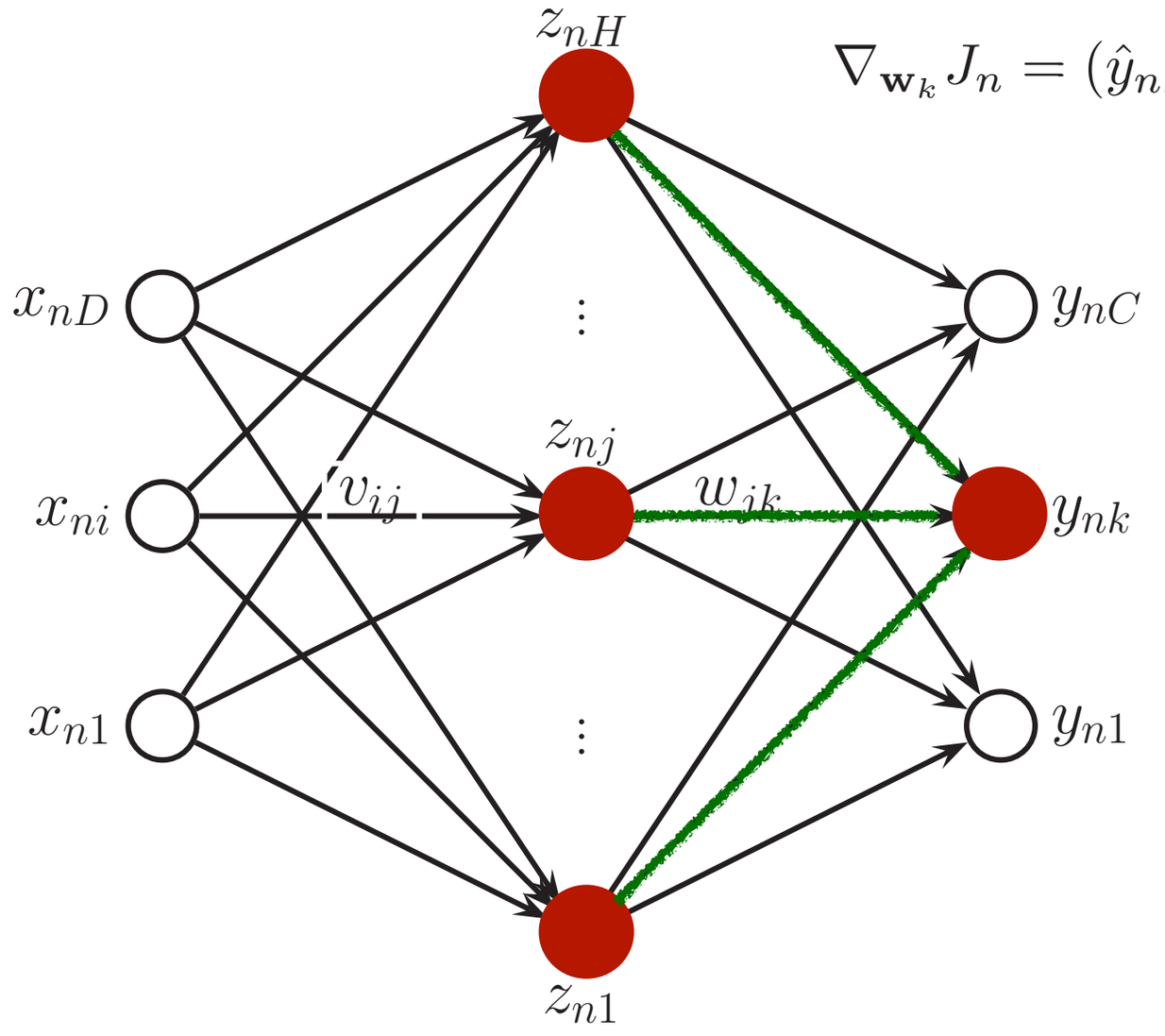
For any GLM: $$\frac{\partial J_n}{\partial b_{nk}} \triangleq \delta_{nk}^w = (\hat{y}_{nk} - y_{nk})$$ Assignment 1 for Softmax

$$\nabla_{\mathbf{w}_k} b_{nk} = \mathbf{z}_n \qquad \nabla_{\mathbf{w}_k} J_n = (\hat{y}_{nk} - y_{nk})\, \mathbf{z}_n$$

Back-propagation Derivation

- Gradient of the loss at the last layer

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

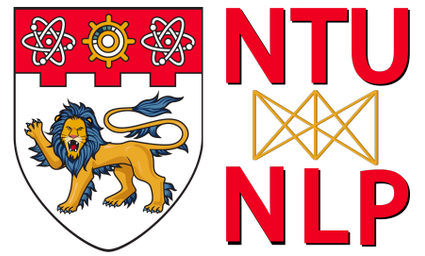


$$\nabla_{\mathbf{w}_k} J_n = (\hat{y}_{nk} - y_{nk})\, \mathbf{z}_n$$

$(\hat{y}_{nk} - y_{nk})$

Error signal

Back-propagation Derivation



- Gradient of the loss at the first layer

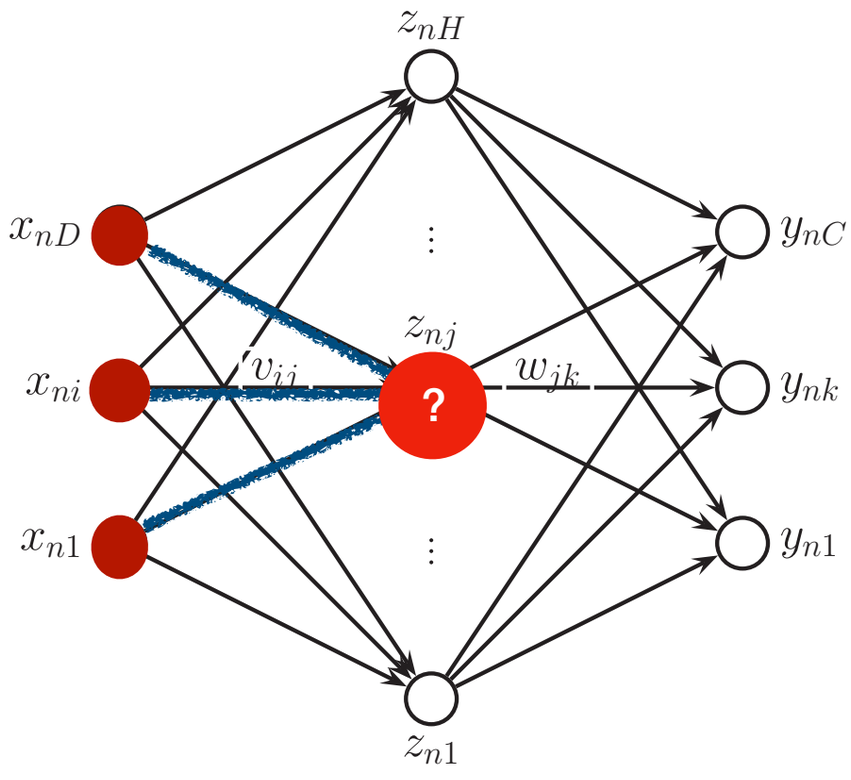
$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n$$

Gradient wrt V

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

Since

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



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

Back-propagation Derivation

NTU NLP

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

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

$$\delta_{nj}^{v} = \frac{\partial J_n}{\partial a_{nj}} = \sum_{k=1}^{K} \frac{\partial J_n}{\partial b_{nk}} \frac{\partial b_{nk}}{\partial a_{nj}}$$

We know: $\frac{\partial J_n}{\partial b_{nk}} \triangleq \delta_{nk}^{w} = (\hat{y}_{nk} - y_{nk})$

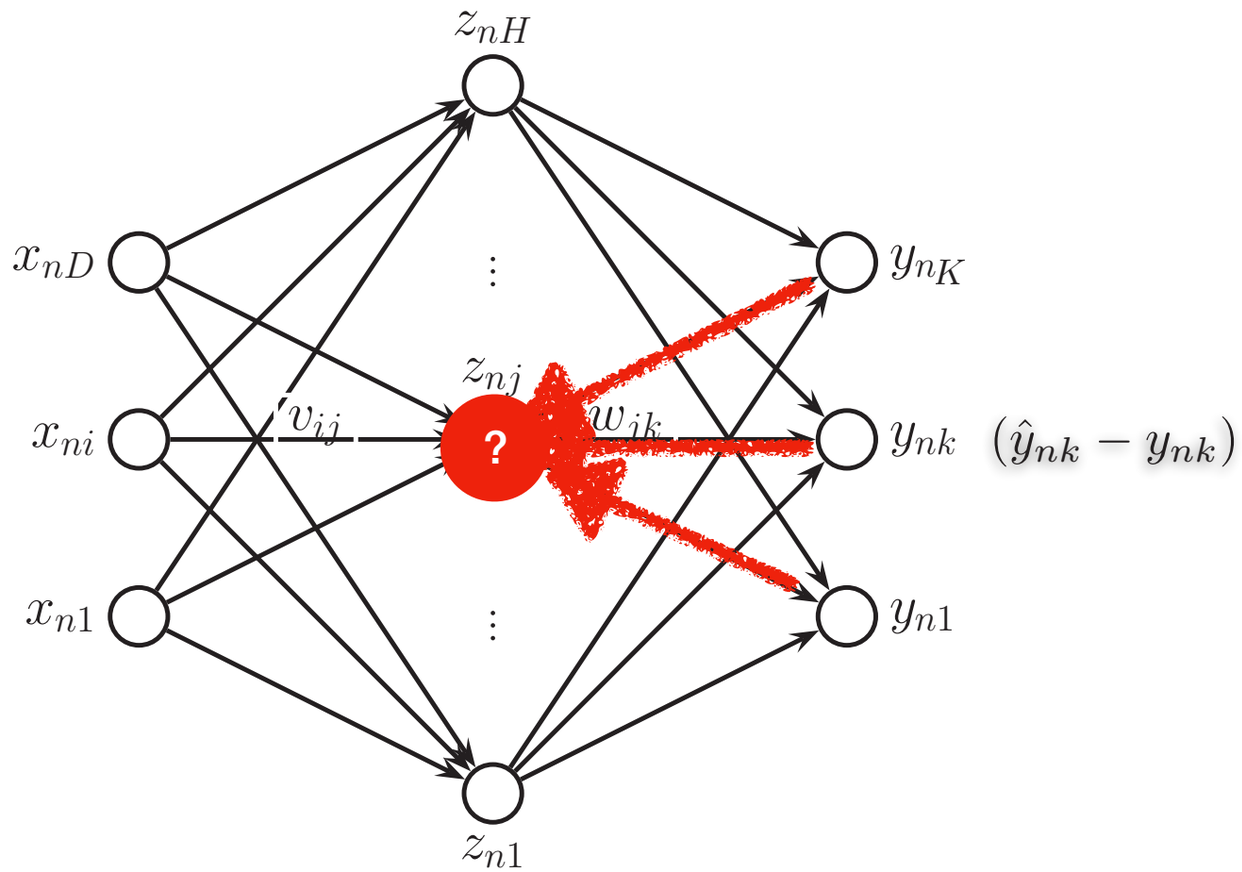
Multiple-path chain rule

$$b_{nk} = \sum_j w_{kj} g(a_{nj}) \; = w_{kj} g'(a_{nj})$$

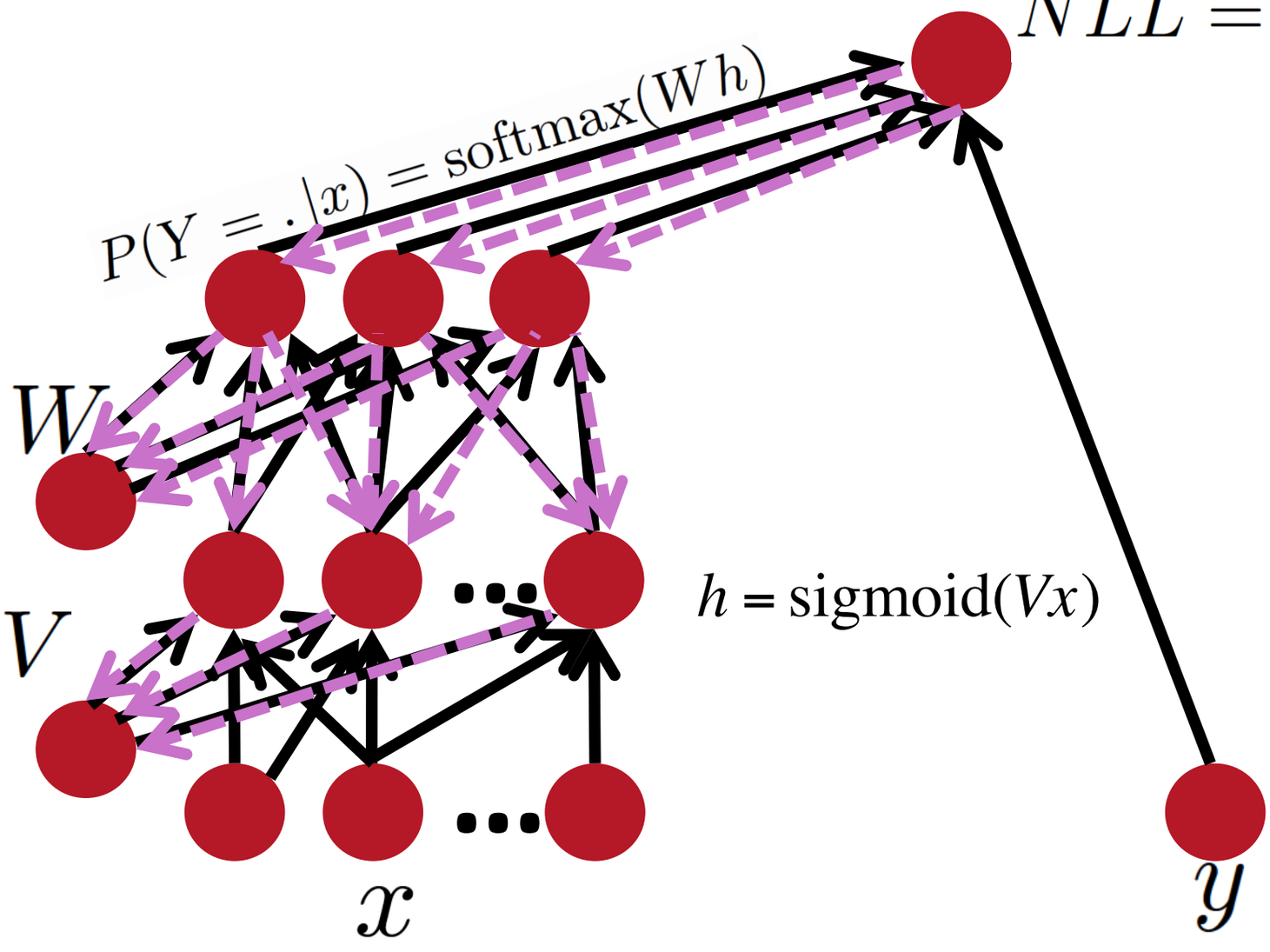
Finally,

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

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_n [\boldsymbol{\delta}_n^{v} \mathbf{x}_n, \; \boldsymbol{\delta}_n^{w} \mathbf{z}_n]$$



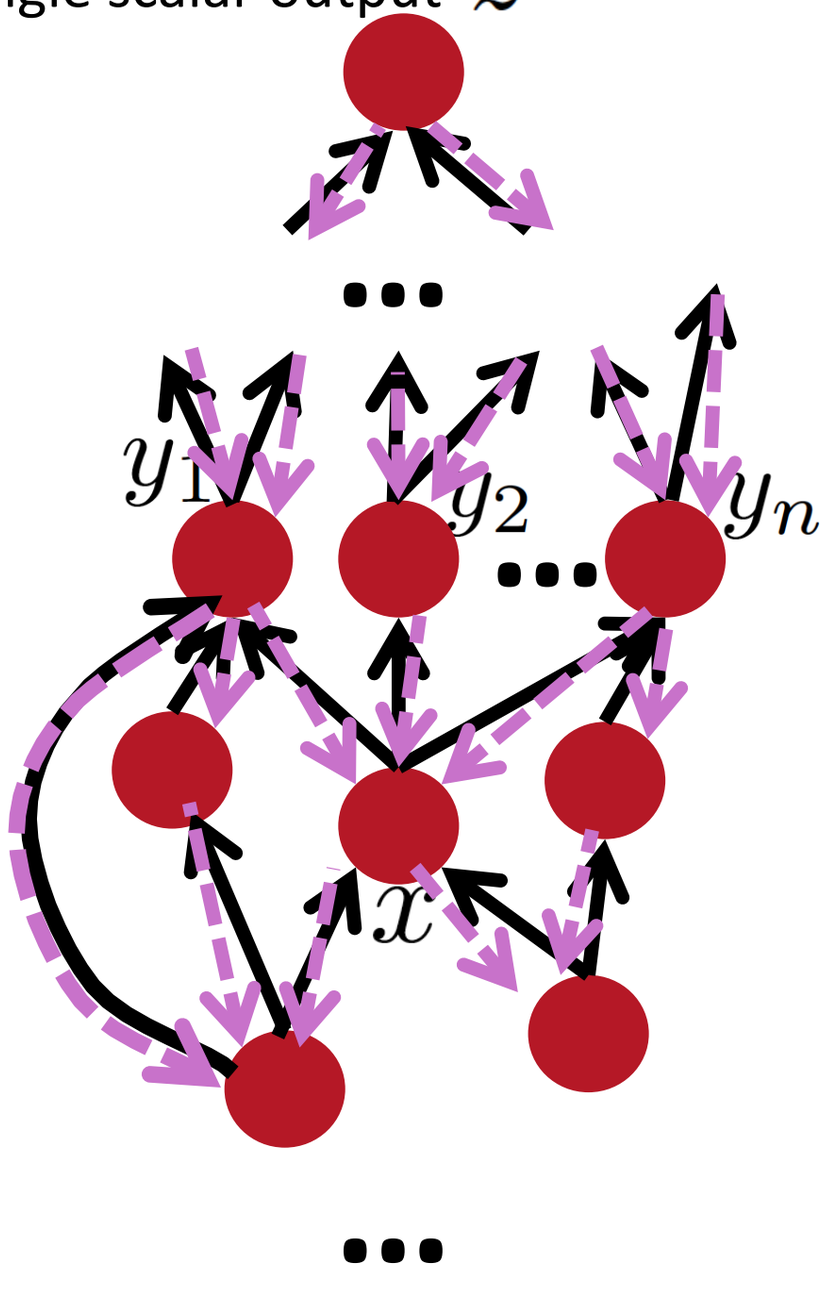
Back-propagation in an MLP



Source: Socher & Manning

Back-propagation in a General Computational Graph

Single scalar output z



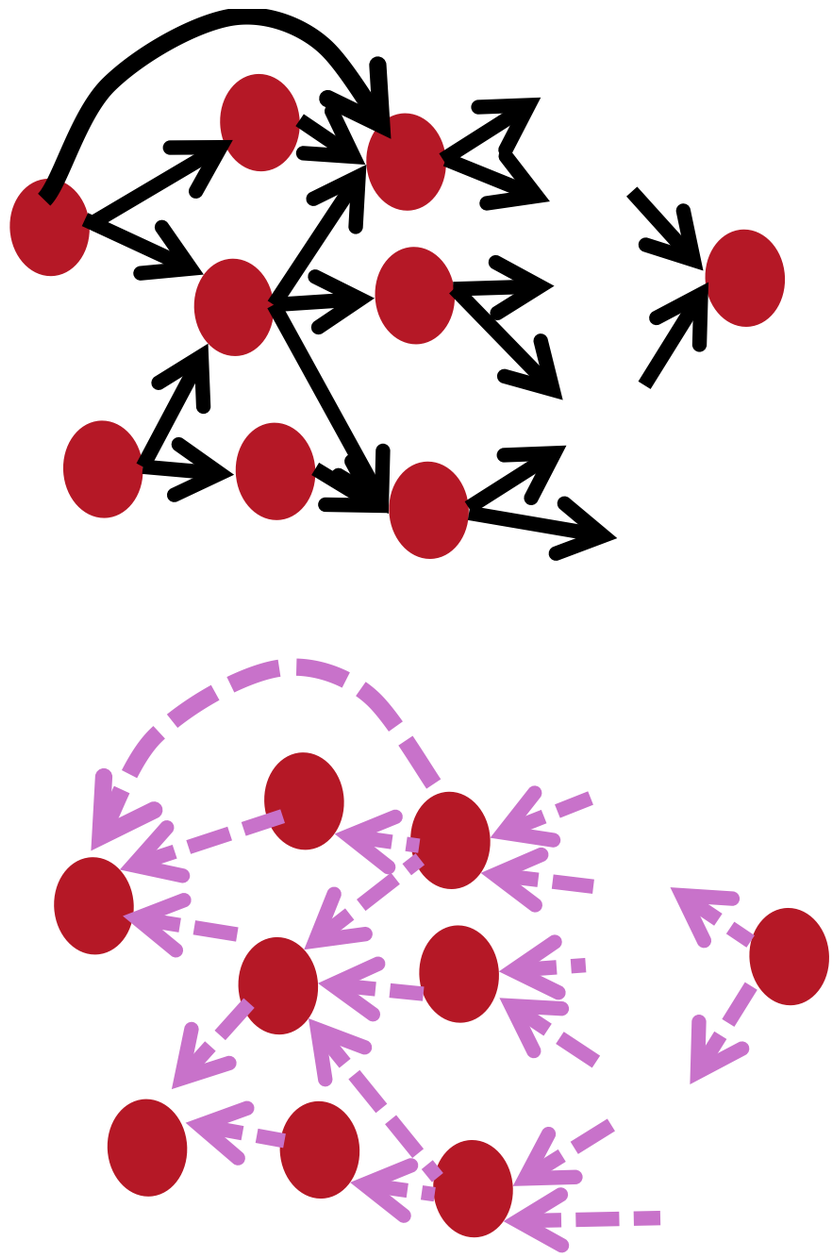
1. Fprop: visit nodes in topo-sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \ldots y_n\}$ = 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

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
```

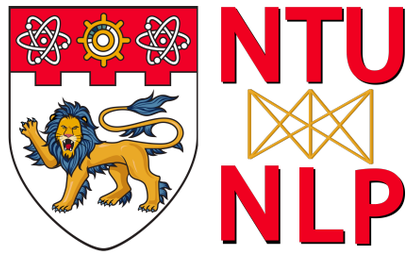
Source: Ruder, 2018

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.

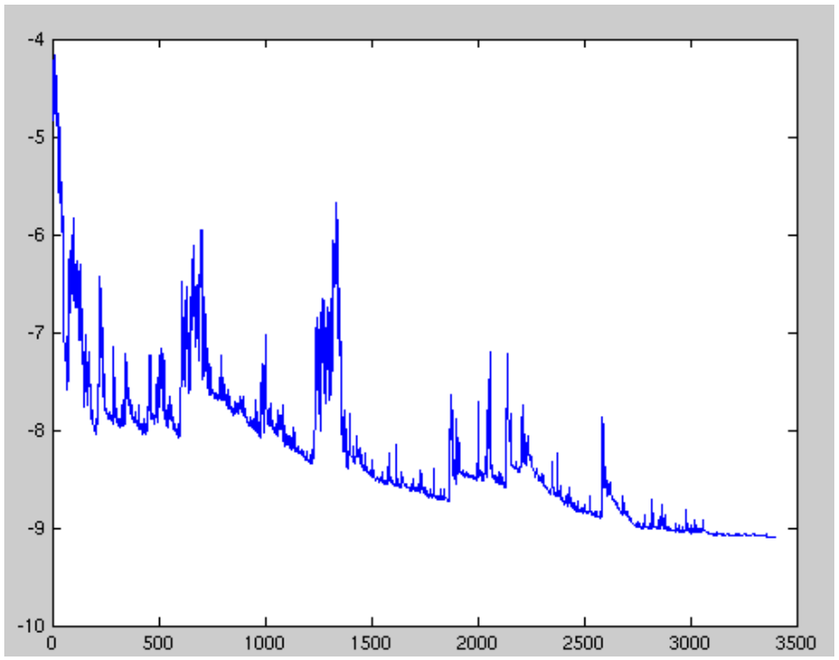


Figure: SGD fluctuation (Source: Wikipedia)

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

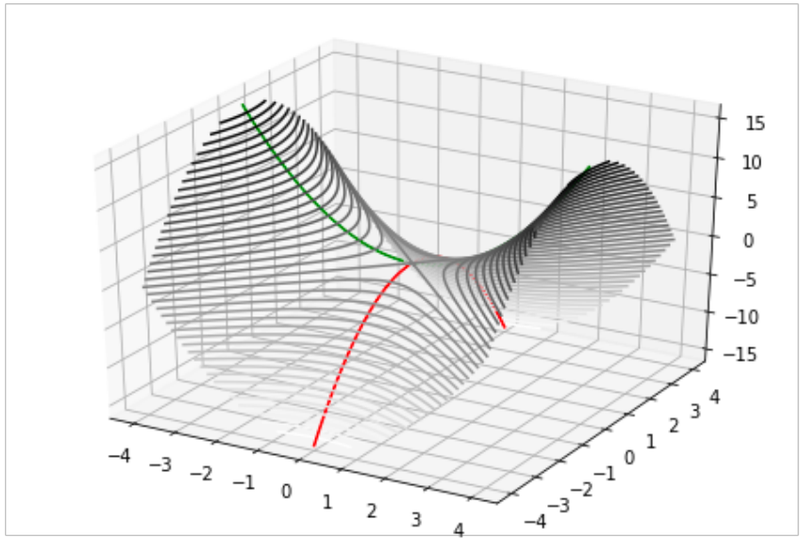
Method	Accuracy	Update Speed	Memory Usage	Online Learning
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

Table: Comparison of trade-offs of gradient descent variants

Source: Ruder, 2018

Challenges

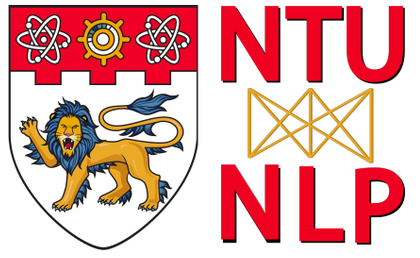
- Choosing a **learning rate**.
- Defining an **annealing schedule**.
- Updating features to **different extent**.
- **Avoiding suboptimal minima.**



Saddle Point

Source: Ruder, 2018

SGD Variants

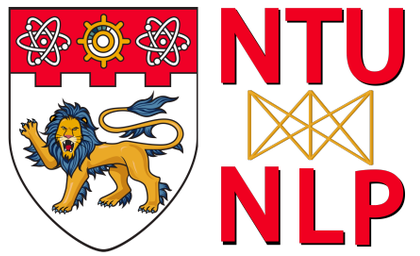


1. Momentum
2. Nesterov accelerated gradient
3. Adagrad
4. Adadelta
5. RMSprop
6. Adam
7. Adam extensions

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

Source: Ruder, 2018

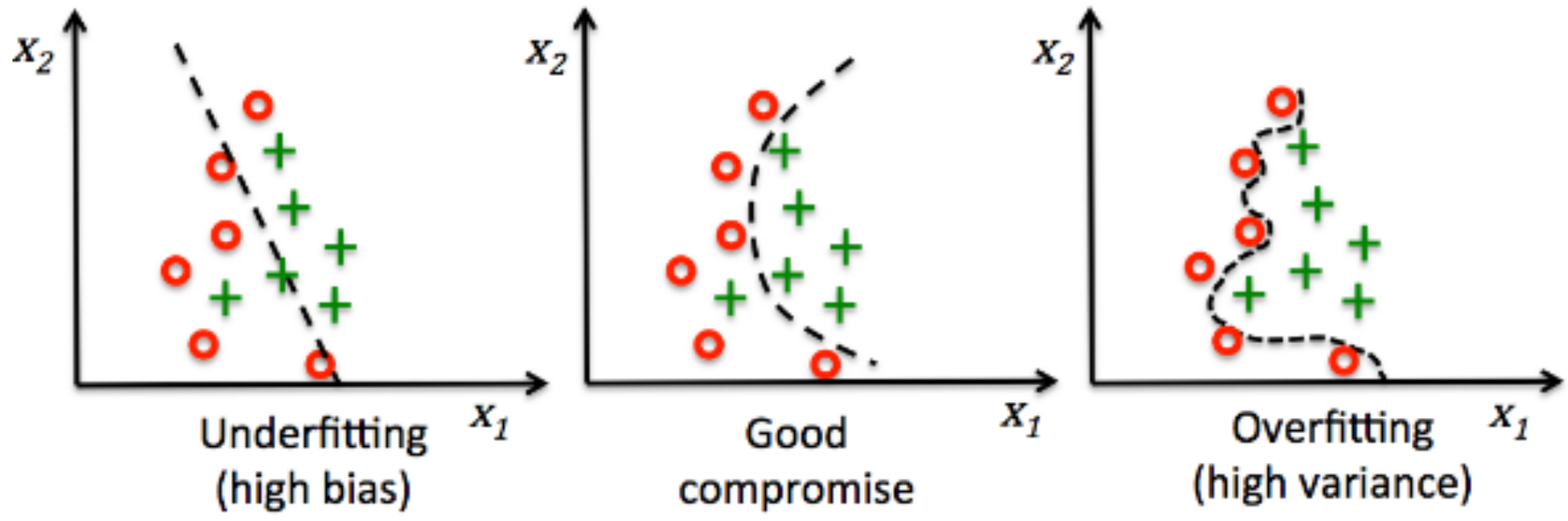
Lecture Plan



- From Logistic/Linear Regression to NN
  - Activation functions
  - SGD with Backpropagation
- Regularisation (dropout, gradient clipping)

Regularisation

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

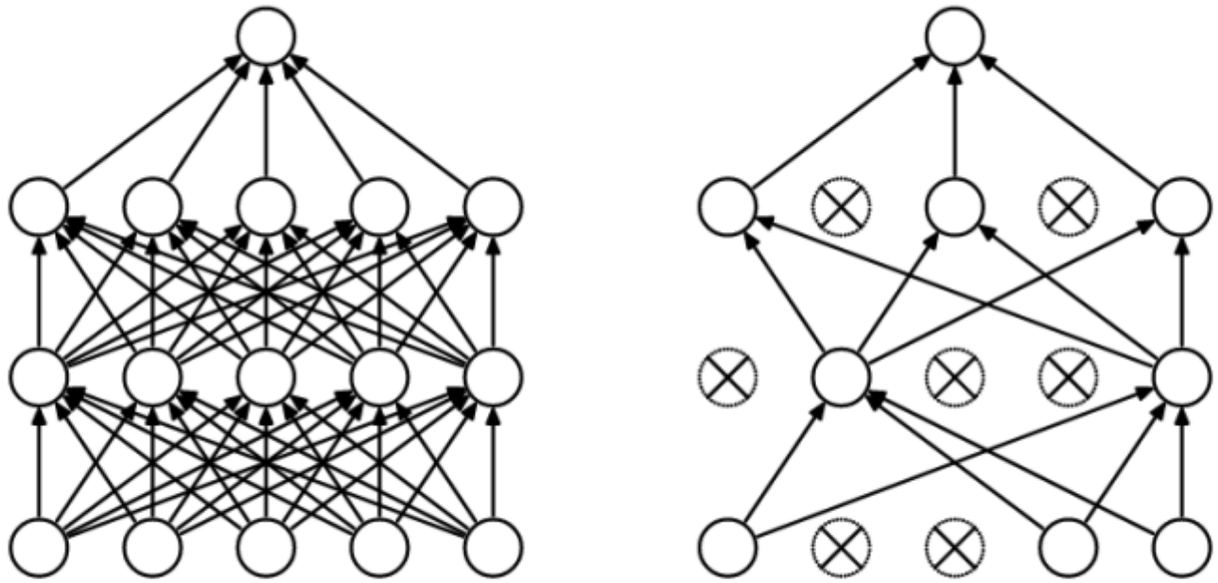


Generalization error increases due to overfitting.

We need to regularise the network

Dropout

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

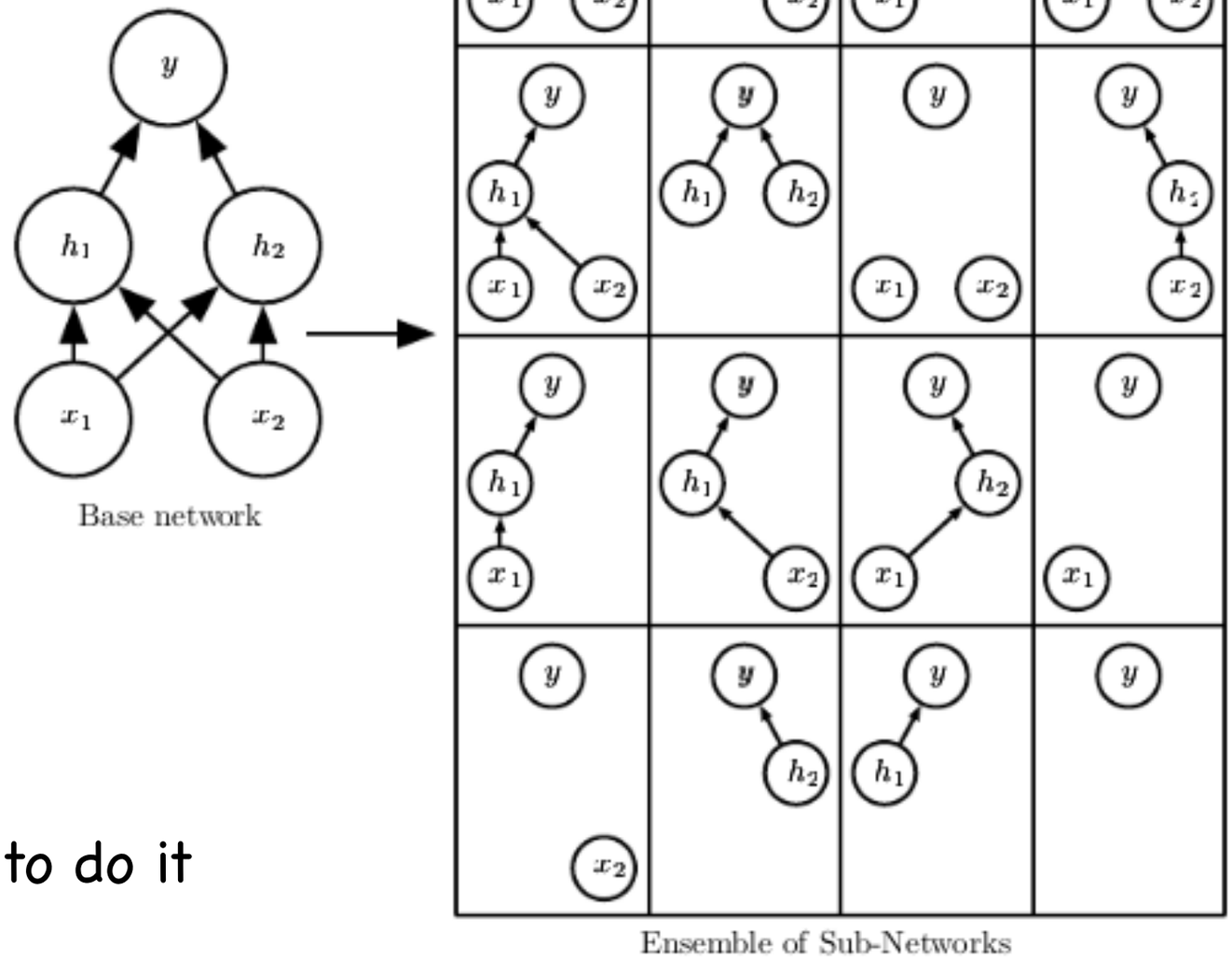


(a) Standard Neural Net (b) After applying dropout.

- 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

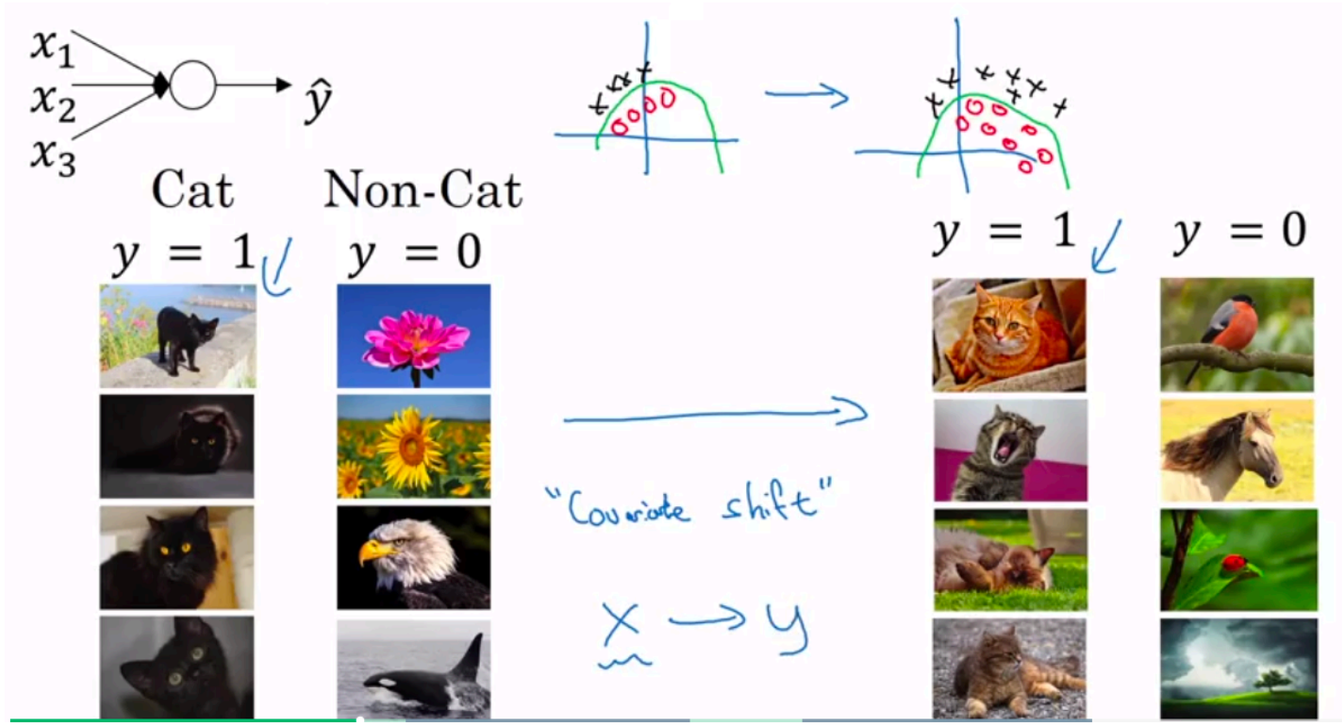
- Has ensemble effect



- Practical exercise on how to do it

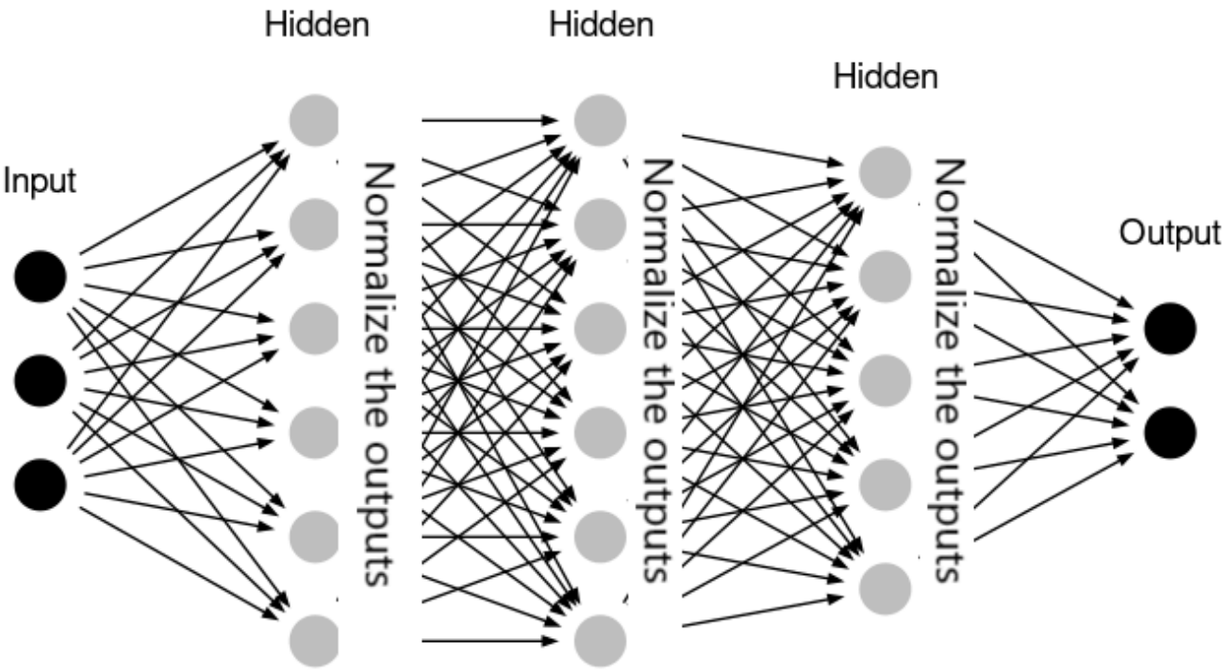
Batch Normalisation

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

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

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

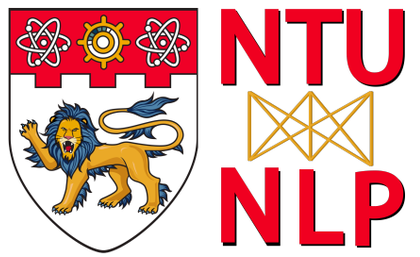
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

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:

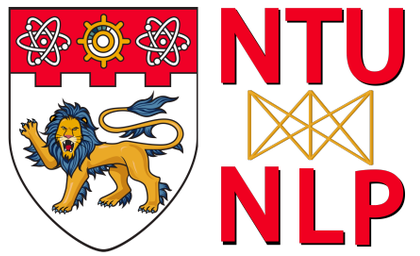
$$\theta^{new} = \theta^{old} - \underbrace{\overbrace{\alpha}^{\text{learning rate}} \nabla_\theta J(\theta)}_{\text{gradient}}$$

- 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

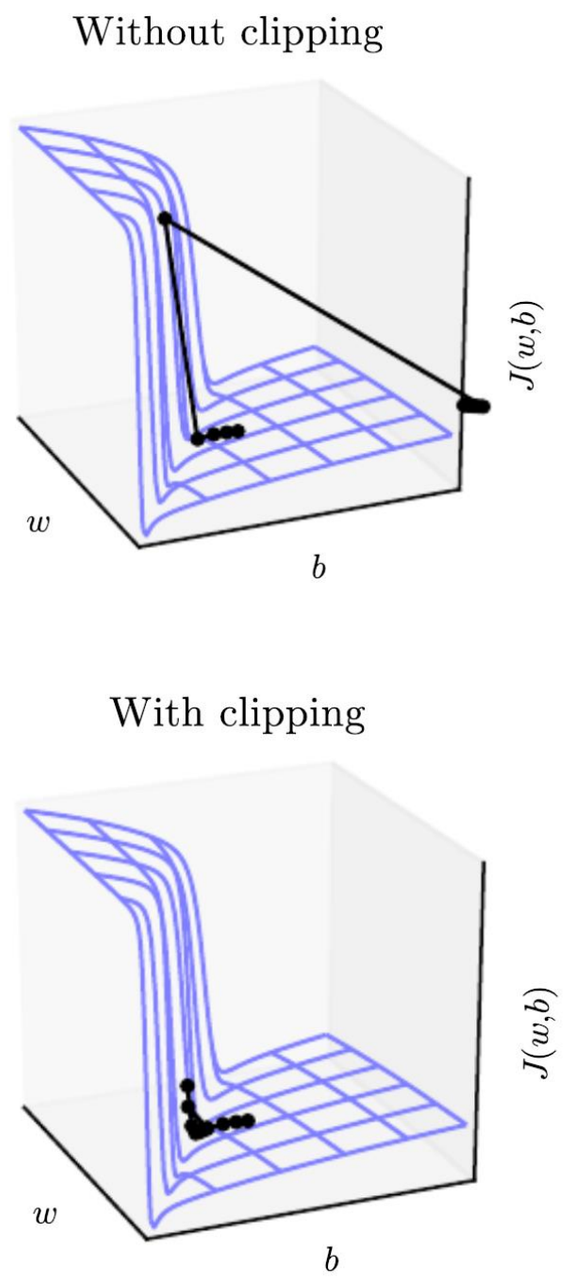
- Intuition: take a step in the same direction, but a smaller step

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

Source: Stanford 224n

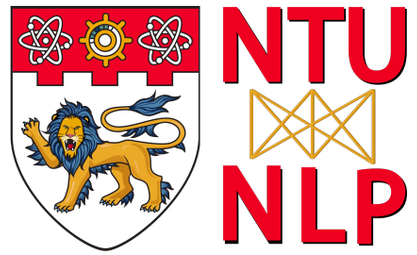
Solution for Exploding Gradient

- 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

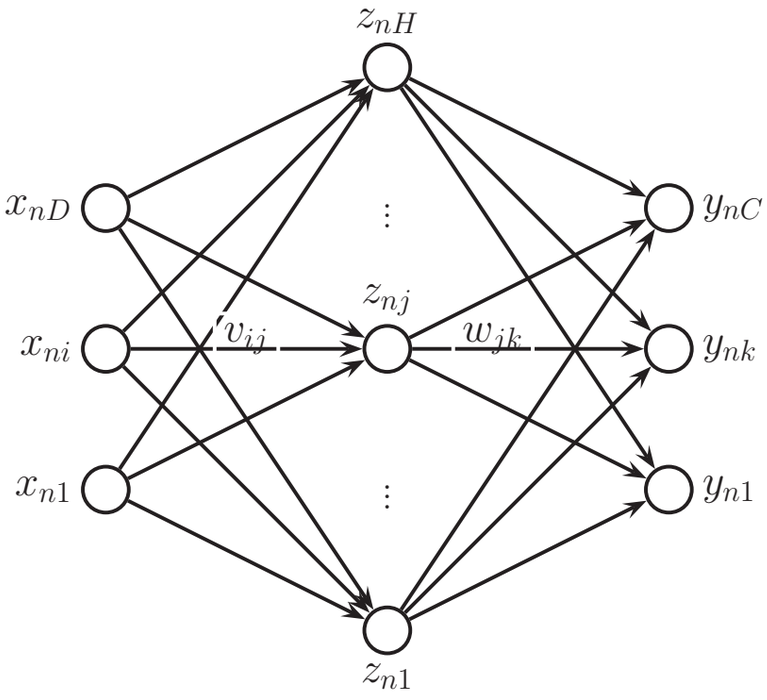


Source: Stanford 224n

Summary



MLP



BackProp