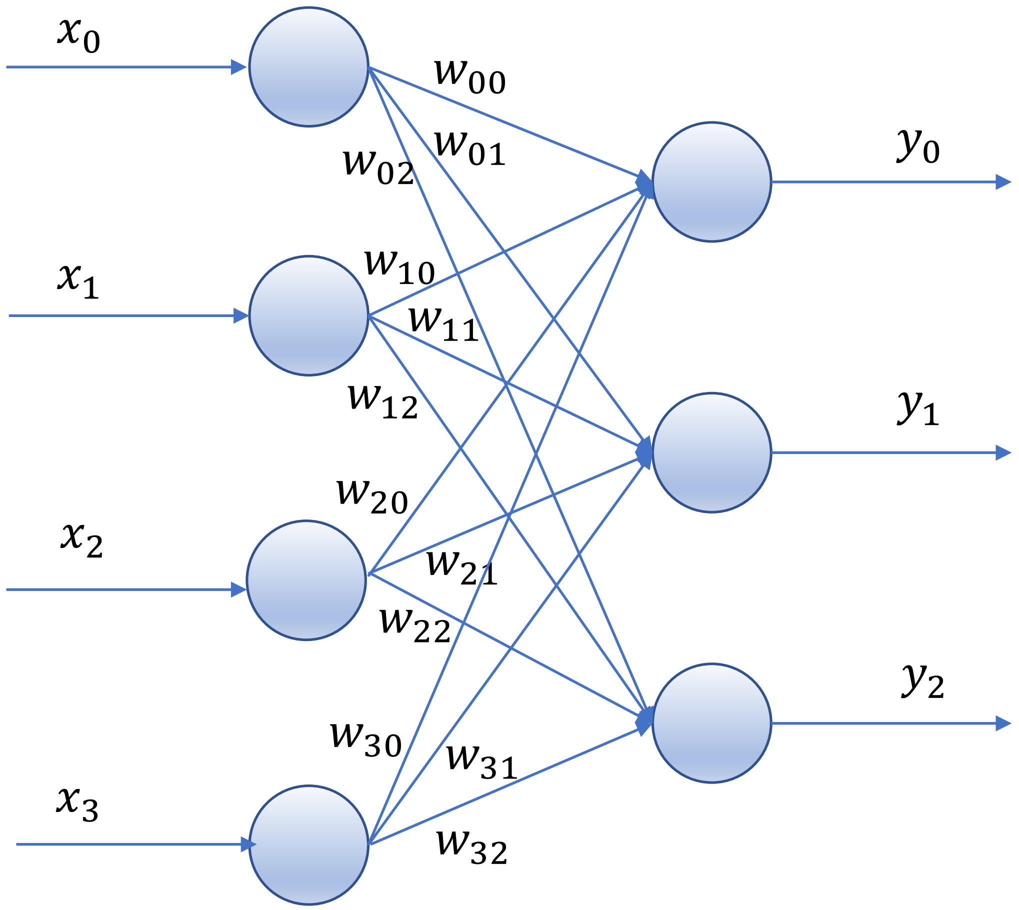
Neural Networks Basics

Dai Bui

Simple Neural Network



$$y = W^T x + b$$

Simple Neural Network

- Consider the simple case: we have sample data $\{x, y\}$ where x is input to a real world object and y is the measured output of the object.
- Let $f(y; W)$ be some objective function that we want to minimize
- We want to train the neural network, e.g., find W matrix, so that the predicted output $\hat{y}$ is close to measured y.

$$\|f(\hat{y}; W) - f(y; W)\|^2 \approx 0$$

- The idea is to adjust weight matrix W to reduce the error

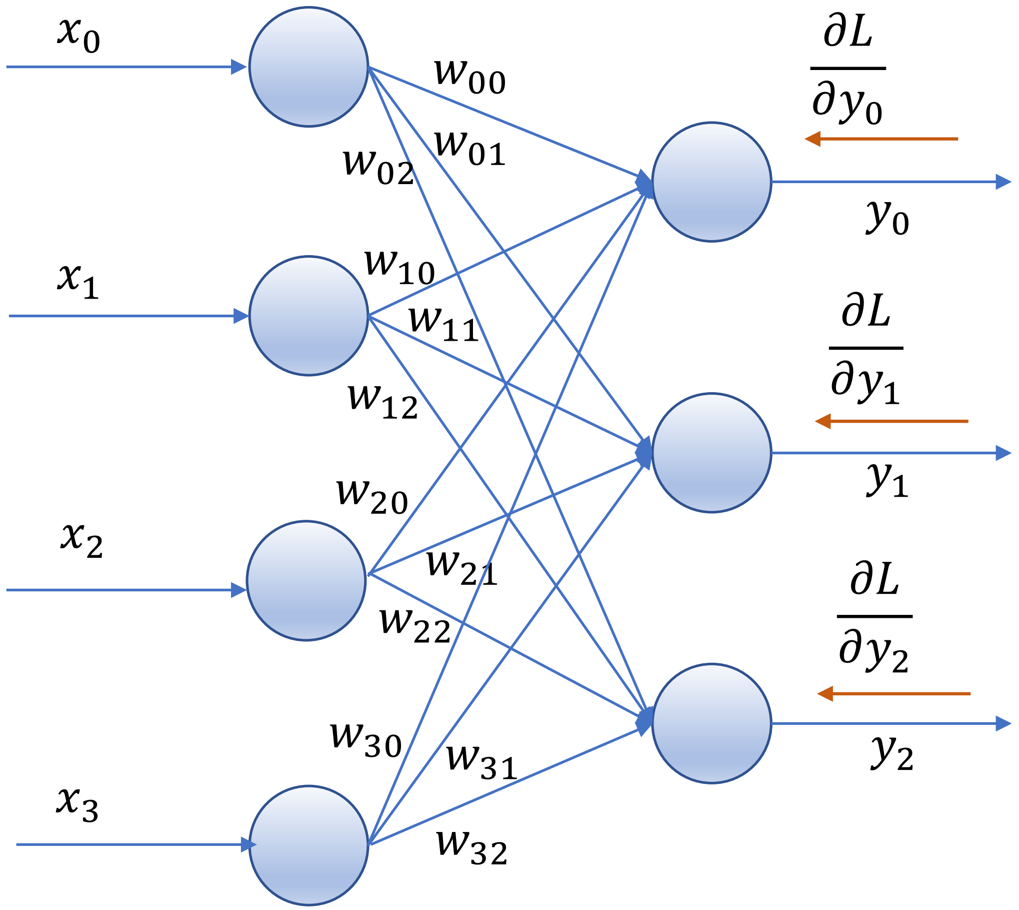
Neural Networks Training

- Neural networks training is composed of two phases
  - Forward propagation: $\hat{y} = W^T x + b$
  - Backward propagation
    - Compute error: $L(W) = \|f(\hat{y}; W) - f(y; W)\|^2$
    - Propagate the error e back to adjust W using the gradient descent method

$$W = W - \alpha \frac{\partial L}{\partial W}$$

Backpropagation

- Let assume that we can compute the gradient: $\frac{\partial f(y;W)}{\partial y}$
- Now we need to compute:
  - $\frac{\partial L}{\partial W}$: gradient is used to adjust W
  - $\frac{\partial L}{\partial x}$: gradient is propagated back to adjust weight matrices of previous layers in case we have multi-layer network

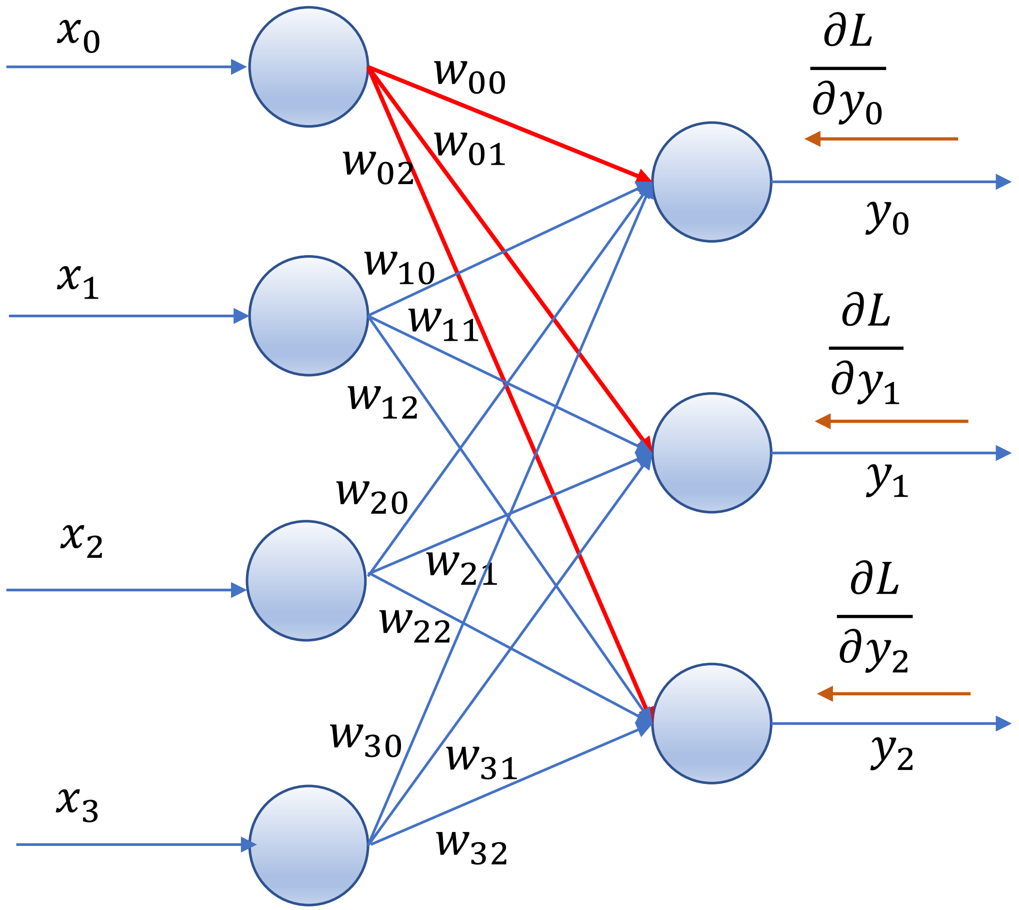


Backpropagation Intuition

- We have the following relation: if $y = f(x)$ then $x \to x + \Delta x \Rightarrow y \to\approx y + \frac{\partial y}{\partial x}\Delta x$
- Let us consider a change Δx_0, this change will lead to:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} \to \begin{bmatrix} y_0 + w_{00}\Delta x_0 \\ y_1 + w_{01}\Delta x_0 \\ y_2 + w_{02}\Delta x_0 \end{bmatrix} = y + \Delta x_0 \begin{bmatrix} w_{00} \\ w_{01} \\ w_{02} \end{bmatrix}$$

$$\Rightarrow f(y) \to f(y) + \begin{bmatrix} \frac{\partial L}{\partial y} \end{bmatrix}^T \Delta x_0 \begin{bmatrix} w_{00} \\ w_{01} \\ w_{02} \end{bmatrix}$$



Backpropagation Intuition

- In short, a change Δx_0 will lead to an approximate change of $\left[\frac{\partial L}{\partial y}\right]^T \Delta x_0 \begin{bmatrix} w_{00} \\ w_{01} \\ w_{02} \end{bmatrix}$ in the objective function

$$\frac{\partial L}{\partial x_0} = \lim_{\Delta x_0 \to 0} \frac{\left(L(y) + \left[\frac{\partial L}{\partial y}\right]^T \Delta x_0 \begin{bmatrix} w_{00} \\ w_{01} \\ w_{02} \end{bmatrix}\right) - L(y)}{\Delta x_0} = \left[\frac{\partial L}{\partial y}\right]^T \begin{bmatrix} w_{00} \\ w_{01} \\ w_{02} \end{bmatrix} = \frac{\partial L}{\partial y_0} w_{00} + \frac{\partial L}{\partial y_1} w_{01} + \frac{\partial L}{\partial y_2} w_{02}$$

Backpropagation Mathematically

- By the chain rule, we know
- $\frac{\partial L}{\partial x_0} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x_0} = \begin{bmatrix} \frac{\partial L}{\partial y_0} \\ \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{bmatrix}^T \begin{bmatrix} w_{00} \\ w_{01} \\ w_{02} \end{bmatrix} = \frac{\partial L}{\partial y_0} w_{00} + \frac{\partial L}{\partial y_1} w_{01} + \frac{\partial L}{\partial y_2} w_{02}$

Backpropagation Input Gradient

$$\boxed{\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^T}$$

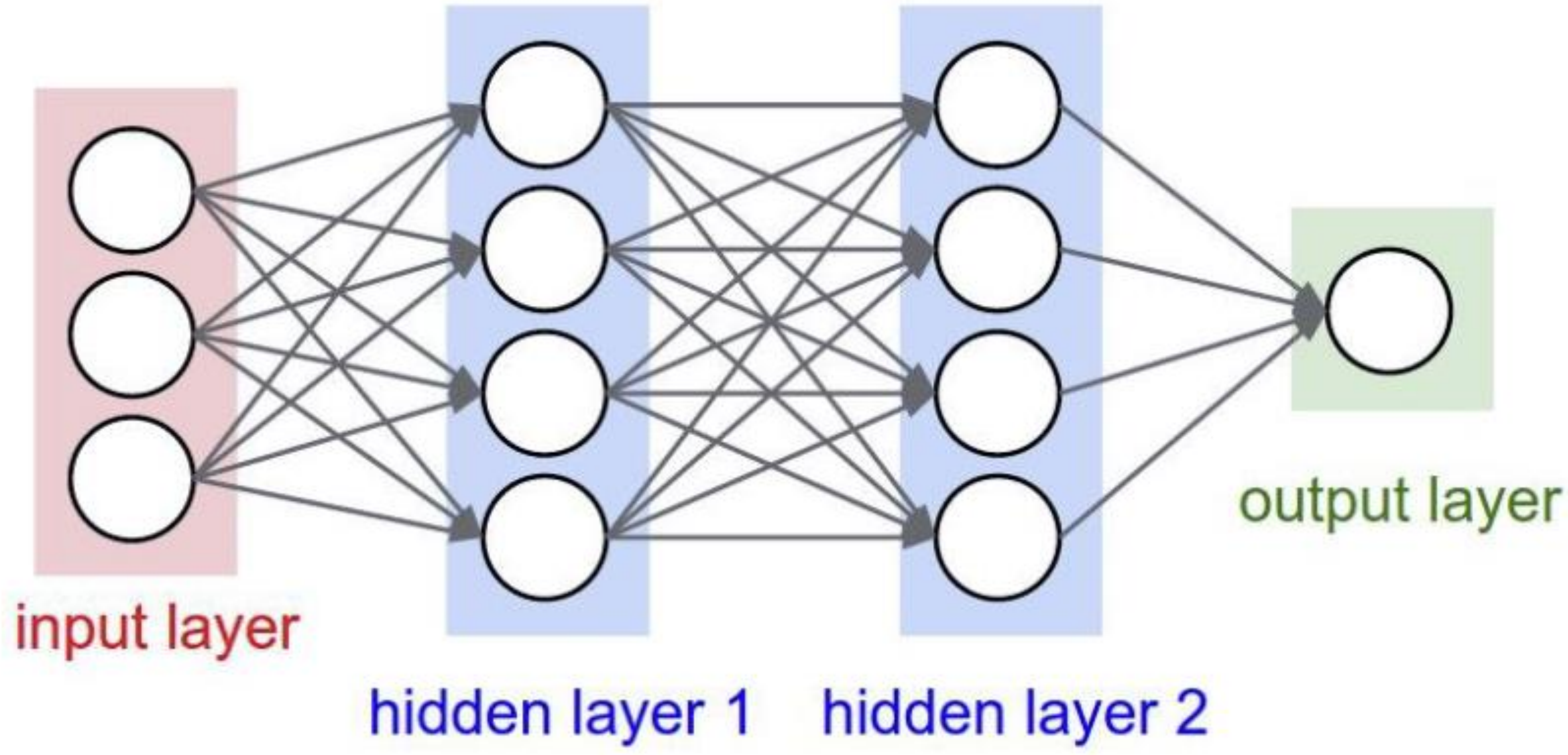
Backpropagation Weight Gradient

$$\boxed{\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial y}}$$

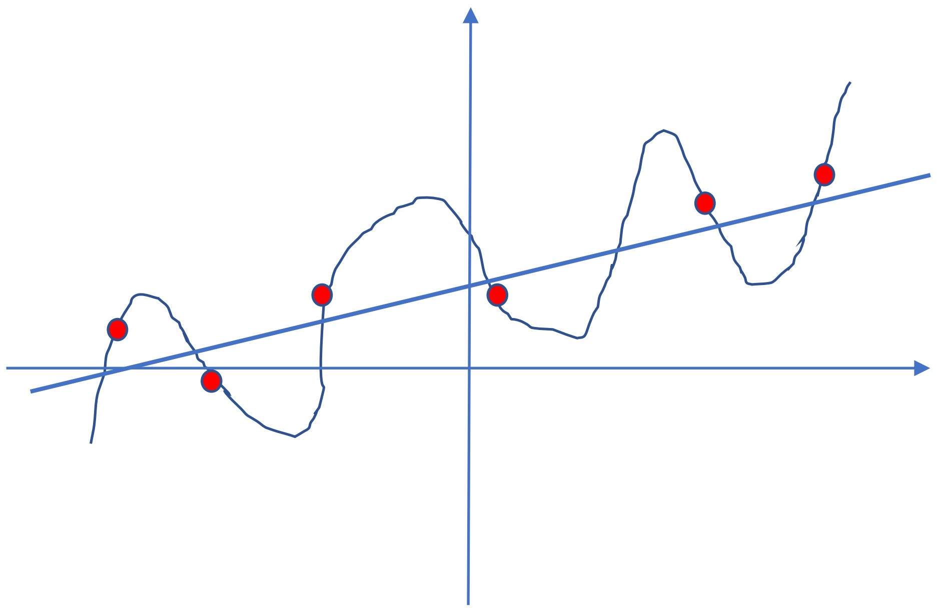
Gradient Descent

$$\boxed{W = W - \alpha \frac{\partial L}{\partial W}}$$

Multilayer Neural Networks



Regularization



- In case we use too many layers as well as too wide hidden layers, it is very easy to fit the data when training but prediction is often wrong. This is the overfitting problem
- To mitigate the problem, we add a regularization factor to reduce the number of weights used.

$$L(W) = f(x; W) + \lambda R(W)$$

$$R(W) = \sum_{i,j} W_{ij}^2$$

Vectorizing Across Multiple Samples

- When we feed one sample at a time into a network of l hidden layers, the computation is composed of multiple vector-matrix multiplication

$$y = ((xW_1 + b_1) \ldots W_l) + b_l$$

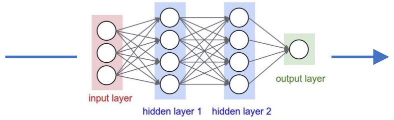
- In general, it is much faster if we can combine multiple vector-matrix multiplications into one single matrix-matrix multiplications

$$r_0 = v_0 M, r_1 = v_1 M, \ldots, r_n = v_n M \Rightarrow \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_l \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_l \end{bmatrix} M$$

- We want to combine input multiple samples together similarly when training to speed up computation
- We call this group of samples batch.

Softmax Classifier

- Suppose that we have the following output after the layers



cat	3.2		0.13		1
car	5.1	$\xrightarrow{\text{softmax}}$	0.87	How do we compare?	0
frog	-1.7		0.00		0

- How do we interpret those raw score numbers to determine the prediction outcome?

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

- Softmax value is the normalized probability assigned to the correct label y_i given input x_i

Cross Entropy

- Information theory: The cross-entropy between a true distribution p and an estimated distribution q is defined as:

$$H(p,q) = -\sum_x p(x)\log q(x)$$

- Because the true distribution $p = [0, \dots, 1, \dots, 0]$ where 1 is at the y_i-th position and $q(y_i) = P(Y = y_i | X = x_i) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}$, therefore the cross entropy between the "predicted" and the "true" distributions becomes

$$\log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} = -s_{y_i} + \log \sum_j e^{s_j}$$

Cross Entropy Loss

- From information theory:
$$H(p, q) = H(p) + D_{KL}(p||q)$$
- $H(p)$ is entropy of true distribution p, which is 0
- $D_{KL}(p||q)$ is Kullback–Leibler divergence, which is a measure how p is different from q
- As we want q to become similar to p, we need to minimize the cross entropy, in other words, we want to minimize the loss function
$$L_i = -s_{y_i} + \log \sum_j e^{s_j}$$

Cross Entropy Loss

- Notice that
$$L_i = -\log P(Y = y_i | X = x_i)$$
- Minimizing cross entropy loss function is the same as maximizing the probability of the correct class $P(Y = y_i | X = x_i)$

Gradient of Cross Entropy Loss

- Given input samples m input samples $\{x^j, y^j\}$
- For each sample, we have the loss function, where C is the number classes, or possible outcomes

$$L(W) = -\sum_{i=1}^{C} y_i^j \log\left(\frac{e^{s_i}}{\sum_{k=1}^{C} e^{s_k}}\right)$$

$$= -\sum_{i=1}^{C} y_i^j \log\left(\frac{e^{W_i^T x^j}}{\sum_{k=1}^{C} e^{W_k^T x^j}}\right)$$

$$= -\sum_{i=1}^{C} y_i^j \log\left(e^{W_i^T x^j}\right) - y_i^j \log\left(\sum_{k=1}^{C} e^{W_k^T x^j}\right)$$

$$= -\sum_{i=1}^{C} y_i^j W_i^T x^j + \log\left(\sum_{k=1}^{C} e^{W_k^T x^j}\right)$$

##notice that $\sum_i^C y_i^j = 1$

Gradient of Cross Entropy Loss

$$\frac{\partial L(W)}{\partial W_i} = -y_i^j x^j + \frac{e^{W_i^T x^j}}{\sum_{k=1}^{C} e^{W_k^T x^j}}$$
$$= -y_i^j x^j + p_i^j x^j$$
$$= (p_i^j - y_i^j) x^j$$

As a result

$$\frac{\partial L(W)}{\partial W} = \begin{bmatrix} (p_0^j - y_0^j)(x^j)^T \\ (p_1^j - y_1^j)(x^j)^T \\ \vdots \\ (p_n^j - y_n^j)(x^j)^T \end{bmatrix} = \begin{bmatrix} (p_0^j - y_0^j) \\ (p_1^j - y_1^j) \\ \vdots \\ (p_n^j - y_n^j) \end{bmatrix} (x^j)^T$$

Batch Gradient of Cross Entropy Loss

- Because we often compute loss function for batch, given

$$X = \begin{bmatrix} (x^1)^T \\ (x^2)^T \\ \vdots \\ (x^m)^T \end{bmatrix}, Y = \begin{bmatrix} (y^1)^T \\ (y^2)^T \\ \vdots \\ (y^m)^T \end{bmatrix}, P = \begin{bmatrix} (p^1)^T \\ (p^2)^T \\ \vdots \\ (p^m)^T \end{bmatrix}$$

then we have

$$\frac{\partial L(W)}{\partial W} = \frac{1}{m} X^T (P - Y)$$

What about the Gradient of Regularization?

$$\frac{\partial L(W)}{\partial W} = \frac{1}{m} X^T (P - Y) + 2\lambda W$$

Stable Softmax

- What happens if s_j in $\frac{e^{s_k}}{\sum_j e^{s_j}}$ are big?
  - The exponential function can be numerically overflown
- Notice that: $\frac{e^{s_k}}{\sum_j e^{s_j}} = \frac{e^{-c}\, e^{s_k}}{e^{-c}\, \sum_j e^{s_j}} = \frac{e^{s_k - c}}{\sum_j e^{s_j - c}}$
- We can basically subtract a constant from each score without changing the softmax probility
- To avoid the case that some s^j is too big, we do the following

$$s_j = s_j - \max_k s_k$$