Word2vec embeddings: CBOW and Skipgram

VL Embeddings

Uni Heidelberg

SS 2019

Skipgram – Intuition

- Window size: 2
- Center word at position t: Maus

	$P(w_{t-2} \mid w_t)$	$P(w_{t-1} \mid w_t)$		$P(w_{t+1} \mid w_t)$	$P(w_{t+2} \mid w_t)$		
Die	kleine	graue	Maus	frißt	den	leckeren	Käse
	w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}		

Skipgram – Intuition

- Window size: 2
- Center word at position t: frißt

		$P(w_{t-2}\|w_t)$	$P(w_{t-1}\|w_t)$		$P(w_{t+1}\|w_t)$	$P(w_{t+2}\|w_t)$	
Die	kleine	graue	Maus	frißt	den	leckeren	Käse
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

Skipgram – Intuition

- Window size: 2
- Center word at position t:

		$P(w_{t-2}\|w_t)$	$P(w_{t-1}\|w_t)$		$P(w_{t+1}\|w_t)$	$P(w_{t+2}\|w_t)$	
Die	kleine	graue	Maus	frißt	den	leckeren	Käse
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

Same probability distribution used for all context words

Skipgram – Objective function

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

Likelihood =
$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta) \quad (1)$$

What is θ?

Skipgram – Objective function

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

Likelihood $=$

$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta) \tag{1}$$

θ: vector representations of each word

Skipgram – Objective function

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

Likelihood =
$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta) \tag{1}$$

θ: vector representations of each word

Objective function (cost function, loss function): Maximise the probability of any context word given the current center word w_t

Skipgram – Objective function

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

Likelihood =

$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta) \quad (1)$$

θ: vector representations of each word

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

$$J(\theta) = -\frac{1}{T} log \ L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} log \ P(w_{t+j} | w_t; \theta) \quad (2)$$

Skipgram – Objective function

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

Likelihood = $$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta) \quad (1)$$

θ: vector representations of each word

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

$$J(\theta) = -\frac{1}{T} log\ L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} log\ P(w_{t+j} | w_t; \theta) \quad (2)$$

Minimising objective function ⇔ maximising predictive accuracy

Objective function – Motivation

- We want to model the probability distribution over mutually exclusive classes
  - measure the difference between predicted probabilities $\hat{y}$ and ground-truth probabilities y
  - during training: tune parameters so that this difference is minimised

Negative log-likelihood

Why is minimising the negative log likelihood equivalent to maximum likelihood estimation (MLE)?

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m} P(w_{t+j} | w_t; \theta)$$

$$MLE = argmax \; L(\theta, x)$$

Negative log-likelihood

Why is minimising the negative log likelihood equivalent to maximum likelihood estimation (MLE)?

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m} P(w_{t+j} | w_t; \theta)$$

$$MLE = argmax \ L(\theta, x)$$

- The log allows us to convert a product of factors into a summation of factors (nicer mathematical properties)
- $\arg\max_x(x)$ is equivalent to $\arg\min_x(-x)$

$$J(\theta) = -\frac{1}{T} log \ L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m} log \ P(w_{t+j} | w_t; \theta)$$

Negative log-likelihood

- We can interpret negative log-probability as information content or surprisal

  What is the log-likelihood of a model, given an event?

$\Rightarrow$ The negative of the surprisal of the event, given the model: A model is supported by an event to the extent that the event is unsurprising, given the model.

Cross entropy loss

Negative log likelihood is the same as cross entropy

Recap: Entropy

- If a discrete random variable X has the probability p(x), then the entropy of X is

$$H(X) = \sum_x p(x) log \frac{1}{p(x)} = - \sum_x p(x) log \; p(x)$$

$\Rightarrow$ expected number of bits needed to encode X if we use an optimal coding scheme

Cross entropy loss

Negative log likelihood is the same as cross entropy

Recap: Entropy

- If a discrete random variable X has the probability p(x), then the entropy of X is

$$H(X) = \sum_x p(x) log \frac{1}{p(x)} = - \sum_x p(x) log \ p(x)$$

$\Rightarrow$ expected number of bits needed to encode X if we use an optimal coding scheme

Cross entropy

$\Rightarrow$ number of bits needed to encode X if we use a suboptimal coding scheme q(x) instead of p(x)

$$H(p, q) = \sum_x p(x) log \frac{1}{q(x)} = - \sum_x p(x) log \ q(x)$$

Cross entropy loss and Kullback-Leibler divergence

Cross entropy is always larger than entropy (exception: if $p = q$)

Cross entropy loss and Kullback-Leibler divergence

Cross entropy is always larger than entropy (exception: if $p = q$)

Kullback-Leibler (KL) divergence: difference between

cross entropy and entropy

Cross entropy loss and Kullback-Leibler divergence

Cross entropy is always larger than entropy (exception: if $p = q$)

Kullback-Leibler (KL) divergence: difference between

cross entropy and entropy

$$KL(p||q) = \sum_x p(x) log \frac{1}{q(x)} - \sum_x p(x) log \frac{1}{p(x)} = \sum_x p(x) log \frac{p(x)}{q(x)}$$

$\Rightarrow$ number of *extra* bits needed when using $q(x)$ instead of $p(x)$ (also known as the relative entropy of p with respect to q)

Cross entropy loss and Kullback-Leibler divergence

Cross entropy is always larger than entropy (exception: if $p = q$)

Kullback-Leibler (KL) divergence: difference between

cross entropy and entropy

$$KL(p||q) = \sum_x p(x) log \frac{1}{q(x)} - \sum_x p(x) log \frac{1}{p(x)} = \sum_x p(x) log \frac{p(x)}{q(x)}$$

$\Rightarrow$ number of *extra* bits needed when using $q(x)$ instead of $p(x)$ (also known as the relative entropy of p with respect to q)

Cross entropy:

$$H(p,q) = -\sum_{x \in X} p(x)\ log\ q(x) = H(p) + KL(p||q)$$

Cross entropy loss and Kullback-Leibler divergence

Cross entropy is always larger than entropy (exception: if $p = q$)

Kullback-Leibler (KL) divergence: difference between

cross entropy and entropy

$$KL(p||q) = \sum_x p(x) log \frac{1}{q(x)} - \sum_x p(x) log \frac{1}{p(x)} = \sum_x p(x) log \frac{p(x)}{q(x)}$$

$\Rightarrow$ number of *extra* bits needed when using $q(x)$ instead of $p(x)$ (also known as the relative entropy of p with respect to q)

Cross entropy:

$$H(p,q) = -\sum_{x \in X} p(x)\ log\ q(x) = H(p) + KL(p||q)$$

Minimising $H(p,q) \rightarrow$ minimising the KL divergence from q to p

Cross-entropy loss (or logistic loss)

- Use cross entropy to measure the difference between two distributions p and q
- Use total cross entropy over all training examples as the loss

$$L_{cross-entropy}(p, q) = -\sum_i p_i log(q_i)$$

$$= -log(q_t) \qquad \text{for hard classification}$$

where q_t is the correct class

Cross-entropy loss (or logistic loss)

- Use cross entropy to measure the difference between two distributions p and q
- Use total cross entropy over all training examples as the loss

$$L_{cross-entropy}(p,q) = -\sum_i p_i log(q_i)$$

$$= -log(q_t)$$

for hard classification
where q_t is the correct class

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

Negative log-likelihood = cross entropy

Skipgram – Objective function

We want to minimise the objective function:

Cross-entropy loss

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\leq j\leq m\\ j\neq 0}} log\ P(w_{t+j}|w_t;\theta) \tag{2}$$

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

Skipgram – Objective function

We want to minimise the objective function:

Cross-entropy loss

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\leq j\leq m \\ j\neq 0}} log\ P(w_{t+j}|w_t;\theta) \tag{2}$$

- Question: How to calculate $P(w_{t+j}|w_t;\theta)$?
- Answer: We will use two vectors per word w:
  - v_w when w is a center word
  - u_w when w is a context word
- Then for a center word c and a context word o:

$$P(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w\in V} exp(u_w^T v_c)} \tag{3}$$

Skipgram – Objective function

We want to minimise the objective function:

Cross-entropy loss

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\leq j\leq m \\ j\neq 0}} log\ P(w_{t+j}|w_t;\theta) \quad (2)$$

- <u>Question:</u> How to calculate $P(w_{t+j}|w_t;\theta)$?
- <u>Answer:</u> We will use two vectors per word w:
  - v_w when w is a center word
  - u_w when w is a context word
- Then for a center word c and a context word o:

$$P(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w\in V} exp(u_w^T v_c)} \quad (3)$$

Take dot products between the two word vectors, put them in Softmax

Recap: Dot products

- Measure of similarity (well, kind of...)
- Bigger if u and v are more similar
  (if vectors point in the same direction)

$$u^\top v = u \cdot v = \sum_{i=1}^{n} u_i v_i \tag{4}$$

- Iterating over $w = 1 \dots W : u_w^\top v$

$\Rightarrow$ work out how similar each word is to v

$$P(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w=1}^{V} exp(u_w^T v_c)} \tag{5}$$

Softmax function

Standard mapping from $\mathbb{R}^V$ to a probability distribution

Exponentiate to make positive

$$p_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

Normalise to get probability

- Softmax function maps arbitrary values x_i to a probability distribution p_i
  - max because amplifies probability of largest x_i
  - soft because still assigns some probability to smaller x_i

This gives us a probability estimate $p(w_{t-1}|w_t)$

Difference Sigmoid Function – Softmax

Sigmoid Function

- binary classification in logistic regression
- sum of probabilities not necessarily 1
- activation function

Softmax Function

- multi-classification in logistic regression
- sum of probabilities will be 1

Why two representations for each word?

- We create two representations for each word in the corpus:
  1. w as a context word
  2. w as a center word
- Easier to compute $\rightarrow$ we can optimise vectors separately
- Also works better in practice...

Skipgram – Predict the label

Dot product compares similarity of o and c
Larger dot product = larger probability

$$p(o|c) = \frac{exp(u_o^\top v_c)}{\sum_{w \in V} exp(u_w^\top v_c)} \tag{6}$$

After taking exponent,
normalise over entire vocab

Skipgram – Predict the label

Dot product compares similarity of o and c
Larger dot product = larger probability

$$p(o|c) = \frac{exp(u_o^\top v_c)}{\sum_{w \in V} exp(u_w^\top v_c)} \tag{6}$$

After taking exponent,
normalise over entire vocab

- For training the model, compute for all words in the corpus:

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

Skipgram – Training the model

- Recall: θ represents all model parameters, in one long vector
- For d-dimensional vectors and V-many words:

$$\theta = \begin{bmatrix} v_{aas} \\ v_{amaranth} \\ \vdots \\ v_{zoo} \\ u_{aas} \\ u_{ameise} \\ \vdots \\ u_{zoo} \end{bmatrix} \in \mathbb{R}^{2dV} \tag{7}$$

- Remember: every word has two vectors $\Rightarrow$ 2d
- We now optimise the parameters θ

Skipgram – Training the model

Generative model: predict the context for a given center word

- We have an objective function:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m} log\ P(w_{t+j}|w_t)$$

- We want to minimise the negative log-likelihood (maximise the probability we predict)
- Probability distribution: $p(o|c) = \frac{exp(u_o^\top v_c)}{\sum_{w\in V} exp(u_w^\top v_c)}$
- How do we know how to change the parameters (i.e. the word vectors)?

Skipgram – Training the model

Generative model: predict the context for a given center word

- We have an objective function:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m} log\ P(w_{t+j}|w_t)$$

- We want to minimise the negative log-likelihood (maximise the probability we predict)
- Probability distribution: $p(o|c) = \frac{exp(u_o^\top v_c)}{\sum_{w\in V} exp(u_w^\top v_c)}$

- How do we know how to change the parameters (i.e. the word vectors)? $\rightarrow$ Use the gradient

Minimising the objective function

We want to optimise (maximise or minimise) our objective function

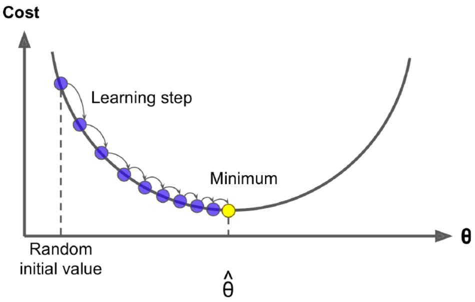
- How do we know how to change the parameters?

Use the gradient

- Gradient $\nabla J(\theta)$ of a function gives direction of steepest ascent
- Gradient Descent is an algorithm to minimise $J(\theta)$

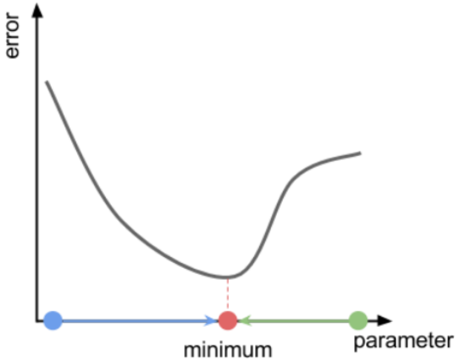
Gradient Descent – Intuition

- **Idea:**
  - for a current value of θ, calculate gradient of $J(\theta)$
  - then take a small step in the direction of the negative gradient
  - repeat



Gradient Descent – Intuition

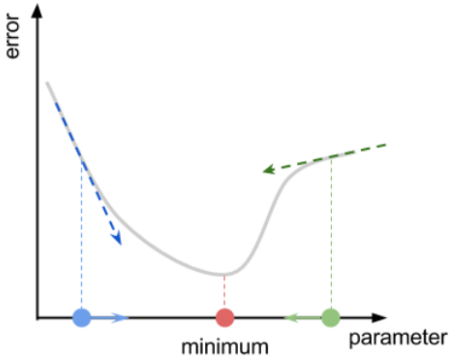
- Find local minimum for a given cost function
    - at each step, GD tells us in which direction to move to lower the cost



- No guarantee that we find the best global solution!

Gradient Descent – Intuition

- How do we know the direction?
- Best guess: move in the direction of the slope (gradient) of the cost function



- Arrows: gradient of the cost function at different points

Gradient Descent – Intuition

- Gradient of a function
  - vector that points in the direction of the steepest ascent
- Gradient is deeply connected to its derivative
- Derivative f' of a function
  - a single number that indicates how fast the function is rising when moving in the direction of its gradient
  - $f'(p)$: value of f' at point p
    - $f'(p) > 0 \Rightarrow f$ is going up
    - $f'(p) < 0 \Rightarrow f$ is going down
    - $f'(p) = 0 \Rightarrow f$ is flat

Gradient-Based Optimisation

Given some function $y = f(x)$ with $x, y \in \mathbb{R}$

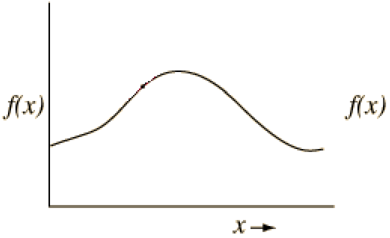
- We want to optimise (maximise or minimise) it by updating x

$$min_{x \in \mathbb{R}} f(x)$$

Gradient-Based Optimisation

Given some function $y = f(x)$ with $x, y \in \mathbb{R}$

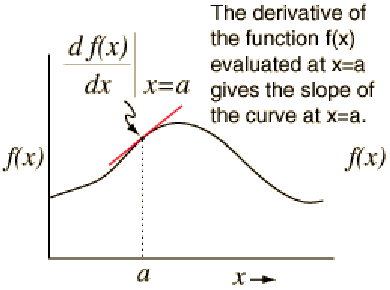
- $min_{x \in \mathbb{R}} f(x)$



Gradient-Based Optimisation

Given some function $y = f(x)$ with $x, y \in \mathbb{R}$

- $min_{x \in \mathbb{R}} f(x)$



Gradient-Based Optimisation

Given some function $y = f(x)$ with $x, y \in \mathbb{R}$

- the derivative $f'(x)$ of this function is $\frac{dy}{dx}$
- gives the slope of $f(x)$ at point x

$\Rightarrow$ tells us how to change x
to make a small improvement in y:

$x_i = x_{i-1} - \alpha f'(x_i)$ α = step size or learning rate

Gradient-Based Optimisation

Given some function $y = f(x)$ with $x, y \in \mathbb{R}$

- the derivative $f'(x)$ of this function is $\frac{dy}{dx}$
- gives the slope of $f(x)$ at point x

$\Rightarrow$ tells us how to change x
to make a small improvement in y:

$x_i = x_{i-1} - \alpha f'(x_i)$ α = step size or learning rate

- Gradient Descent: reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative

Gradient-Based Optimisation

Given some function $y = f(x)$ with $x, y \in \mathbb{R}$

- the derivative $f'(x)$ of this function is $\frac{dy}{dx}$
- gives the slope of $f(x)$ at point x

$\Rightarrow$ tells us how to change x
to make a small improvement in y:

$x_i = x_{i-1} - \alpha f'(x_i)$ α = step size or learning rate

- Gradient Descent: reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative

What if we have functions with multiple inputs?

Gradient Descent with multiple inputs

- We can use partial derivatives $\frac{\partial}{\partial x_i} f(x)$
  - measures how f changes as only x_i increases at point x
- Gradient of f:
  - gives direction of steepest ascent $\nabla_x f(x)$
  - vector containing all partial derivatives for $f(x)$
- Element i of the gradient ∇ is the partial derivative of f with respect to x_i

Gradient Descent with multiple inputs

- We can use partial derivatives $\frac{\partial}{\partial x_i} f(x)$
  - measures how f changes as only x_i increases at point x
- Gradient of f:
  - gives direction of steepest ascent $\nabla_x f(x)$
  - vector containing all partial derivatives for $f(x)$
- Element i of the gradient ∇ is the partial derivative of f with respect to x_i

Which direction should we step to decrease the function?

Gradient Descent with multiple inputs

- We can use partial derivatives $\frac{\partial}{\partial x_i} f(x)$
  - measures how f changes as only x_i increases at point x

- Gradient of f:
  - gives direction of steepest ascent $\nabla_x f(x)$
  - vector containing all partial derivatives for $f(x)$
- Element i of the gradient ∇ is the partial derivative of f with respect to x_i

Which direction should we step to decrease the function?

- Gradient descent algorithm:
  - compute $\nabla_x f(x)$
  - take small step in $-\nabla_x f(x)$ direction
  - repeat

Gradient Descent with multiple inputs

- We can use partial derivatives $\frac{\partial}{\partial x_i} f(x)$
  - measures how f changes as only x_i increases at point x
- Gradient of f:
  - gives direction of steepest ascent $\nabla_x f(x)$
  - vector containing all partial derivatives for $f(x)$
- Element i of the gradient ∇ is the partial derivative of f with respect to x_i

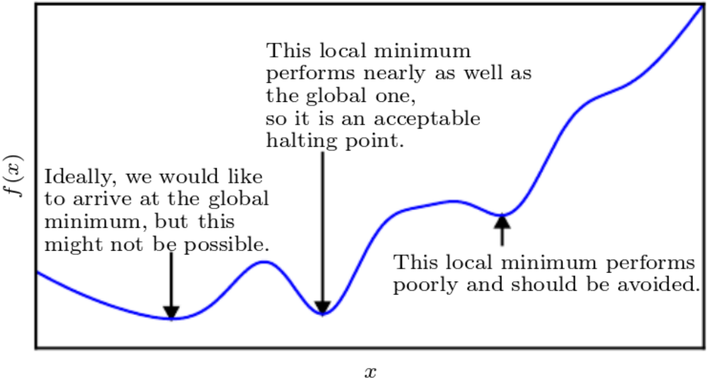
Which direction should we step to decrease the function?

- Gradient descent algorithm:
  - compute $\nabla_x f(x)$
  - take small step in $-\nabla_x f(x)$ direction
  - repeat

- Minimise f by applying small updates to x: $x' = x - \alpha \nabla_x f(x)$

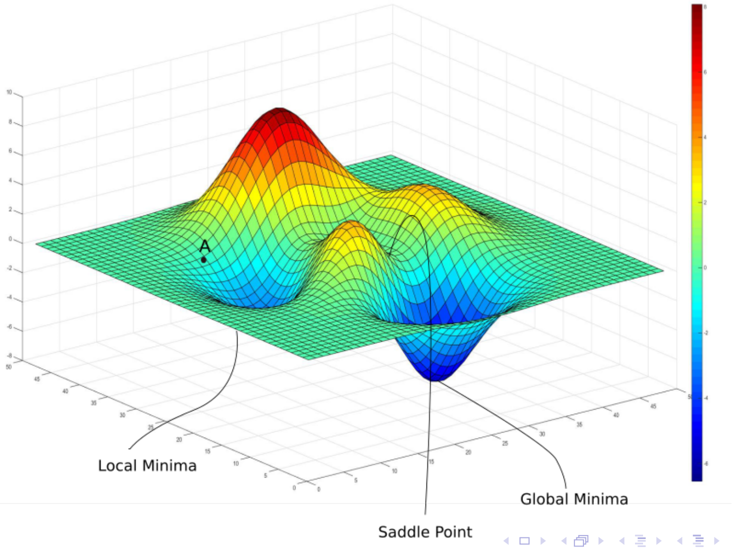
Gradient Descent with multiple inputs

Critical points in 2D (one input value):



Gradient Descent with multiple inputs

Critical points in 3D:



Gradient Descent with multiple inputs

- Update equation (in matrix notation):

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$$

α = step size or learning rate

- Update equation (for a single parameter):

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

Gradient Descent with multiple inputs

- Problem: $J(\theta)$ is a function of **all** windows in the corpus (extremely large!)
  - So $\nabla_\theta J(\theta)$ is very expensive to compute

  $\Rightarrow$ Takes too long for a single update!
- Solution: Stochastic Gradient Descent
  - Repeatedly sample windows and update after each one

Stochastic Gradient Descent (SGD)

Goal: find parameters θ that reduce cost function $J(\theta)$

Algorithm 1 Pseudocode for SGD

Input:
– function $f(x;\theta)$
– training set of inputs $x_1, \ldots, x_n$ and gold outputs $y_1, \ldots, y_n$
– loss function J
while stopping criteria not met **do**
    Sample a training example x_i, y_i
    Compute the loss $J(f(x_i;\theta), y_i)$
    $\nabla \leftarrow$ gradients of $J(f(x_i;\theta), y_i)$ w.r.t. θ
    Update $\theta \leftarrow \theta - \alpha\nabla$
end while
return θ

Stochastic Gradient Descent (SGD)

Goal: find parameters θ that reduce cost function $J(\theta)$

- Impact of learning rate α:
  - too low $\rightarrow$ learning proceeds slowly
  - initial α too low $\rightarrow$ learning may become stuck with high cost

Stochastic Gradient Descent (SGD)

Goal: find parameters θ that reduce cost function $J(\theta)$

- Important property of SGD (and related minibatch or online gradient-based optimization)
  - computation time per update does not grow with increasing number of training examples

Stochastic Gradient Descent (SGD)

$$\theta = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{19.998} \\ w_{19.999} \\ w_{20.000} \end{bmatrix}$$

Stochastic Gradient Descent (SGD)

$$\theta = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{19.998} \\ w_{19.999} \\ w_{20.000} \end{bmatrix}$$

$$-\nabla J(\theta) = \begin{bmatrix} 0.31 \\ 0.03 \\ -1.25 \\ \vdots \\ 0.78 \\ -0.37 \\ 0.16 \end{bmatrix}$$

Stochastic Gradient Descent (SGD)

$$\theta = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{19.998} \\ w_{19.999} \\ w_{20.000} \end{bmatrix}$$

$$-\nabla J(\theta) = \begin{bmatrix} 0.31 \\ 0.03 \\ -1.25 \\ \vdots \\ 0.78 \\ -0.37 \\ 0.16 \end{bmatrix}$$

w_0 *should increase somewhat*
w_1 *should increase a little*
w_2 *should decrease a lot*
$\vdots$
$w_{19.998}$ *should increase a lot*
$w_{19.999}$ *should decrease somewhat*
$w_{20.000}$ *should increase a little*

Stochastic Gradient Descent (SGD)

$$\theta = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{19.998} \\ w_{19.999} \\ w_{20.000} \end{bmatrix}$$

$$-\nabla J(\theta) = \begin{bmatrix} 0.31 \\ 0.03 \\ -1.25 \\ \vdots \\ 0.78 \\ -0.37 \\ 0.16 \end{bmatrix} \qquad \begin{matrix} w_0 \textit{ should increase somewhat} \\ w_1 \textit{ should increase a little} \\ w_2 \textit{ should decrease a lot} \\ \vdots \\ w_{19.998} \textit{ should increase a lot} \\ w_{19.999} \textit{ should decrease somewhat} \\ w_{20.000} \textit{ should increase a little} \end{matrix}$$

Average over all training data

Encodes the relative importance of each weight

Stochastic Gradient Descent (SGD)

- Make a **forward pass** through the network to compute the output
- Take the output that the network predicts
- Take the output that it *should predict*
- Compute the total cost of the network $J(\theta)$

Propagate the error back through the network

Stochastic Gradient Descent (SGD)

- Make a **forward pass** through the network to compute the output
- Take the output that the network predicts
- Take the output that it *should predict*
- Compute the total cost of the network $J(\theta)$

Propagate the error back through the network

$\Rightarrow$ Backpropagation

- procedure to compute the gradient of the cost function:

Compute the partial derivatives $\frac{\partial J(\theta)}{\partial w}$ and $\frac{\partial J(\theta)}{\partial b}$ of the cost function $J(\theta)$ with respect to any weight w or bias b in the network.

Stochastic Gradient Descent (SGD)

- Make a **forward pass** through the network to compute the output
- Take the output that the network predicts
- Take the output that it *should predict*
- Compute the total cost of the network $J(\theta)$

Propagate the error back through the network

$\Rightarrow$ Backpropagation

- procedure to compute the gradient of the cost function:

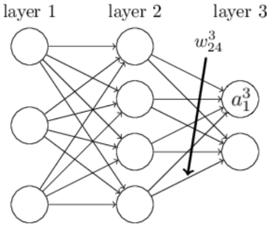
  How do we have to change the weights and biases in order to change the cost?

Parameter initialisation

- Before we start training the network we have to initialise the parameters
  - Why not use zero as initial values?
  - Not a good idea, outputs will be the same for all nodes
  - Instead, use small random numbers, e.g.:
    - use normally distributed values around zero $N(0, 0.1)$
    - use Xavier initialisation (Glorot and Bengio 2010)
    - for debugging: use fixed random seeds
- Now let's start the training:
  - predict labels
  - compute loss
  - update parameters

Forward pass

- Computes the output of the network
- Each node's output depends only on itself and on its incoming edges
- Traverse the nodes and compute the output of each node, given the already computed outputs of its predecessors

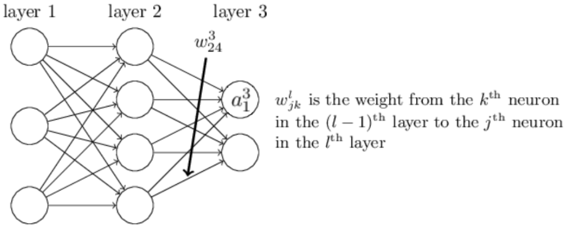


w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

Image taken from `http://neuralnetworksanddeeplearning.com/chap2.html`

Forward pass

- Computes the output of the network
- Each node's output depends only on itself and on its incoming edges
- Traverse the nodes and compute the output of each node, given the already computed outputs of its predecessors

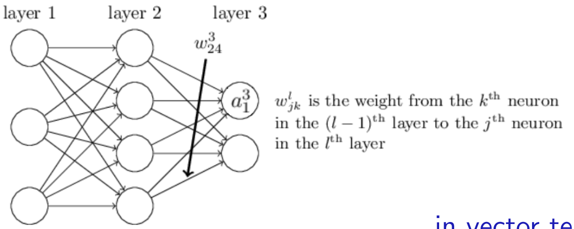


$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

Image taken from `http://neuralnetworksanddeeplearning.com/chap2.html`

Forward pass

- Computes the output of the network
- Each node's output depends only on itself and on its incoming edges
- Traverse the nodes and compute the output of each node, given the already computed outputs of its predecessors



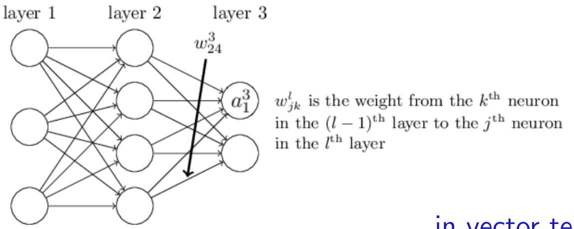
in vector terminology

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right)$$

Image taken from `http://neuralnetworksanddeeplearning.com/chap2.html`

Forward pass

- Computes the output of the network
- Each node's output depends only on itself and on its incoming edges
- Traverse the nodes and compute the output of each node, given the already computed outputs of its predecessors



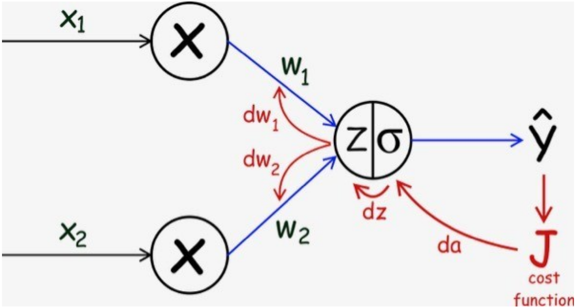
in vector terminology

$$z^l = w^l a^{l-1} + b^l \qquad a^l = \sigma(z^l)$$

Image taken from `http://neuralnetworksanddeeplearning.com/chap2.html`

Parameter update for a 1-layer network

- After a single forward pass, predict the output $\hat{y}$
- Compute the cost J (a single scalar value), given the predicted $\hat{y}$ and the ground truth y
- Take the derivative of the cost J w.r.t w and b
- Update w and b by a fraction (learning rate) of dw and db



Parameter update for a 1-layer network

Forward pass:

$$Z = W^\top X + b$$
$$\hat{y} = A = \sigma(Z)$$

Use the chain rule:

$$\frac{dJ}{dW} = \frac{dJ}{dA}\frac{dA}{dZ}\frac{dZ}{dW}$$
$$\frac{dJ}{db} = \frac{dJ}{dA}\frac{dA}{dZ}\frac{dZ}{db}$$

Update w and b:

$$W = W - \alpha\frac{dJ}{dW}$$
$$b = b - \alpha\frac{dJ}{db}$$

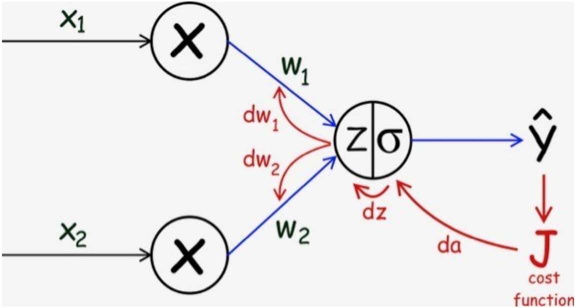


Image taken from http://www.adeveloperdiary.com/data-science/machine-learning/understand-and-implement-the-backpropagation-algorithm-from-scratch-in-python/

Parameter update for a 2-layer network

Forward pass:

$$Z^{[1]} = W^{[1]\top} X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$\hat{y} = A^{[2]} = \sigma(Z^{[2]})$$

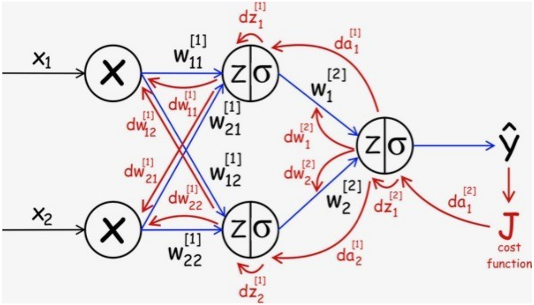
Use the chain rule:

$$dW^{[2]} = \frac{dJ}{dW^{[2]}} = \frac{dJ}{dA^{[2]}} \frac{dA^{[2]}}{dZ^{[2]}} \frac{dZ^{[2]}}{dW^{[2]}}$$

$$db^{[2]} = \frac{dJ}{db^{[2]}} = \frac{dJ}{dA^{[2]}} \frac{dA^{[2]}}{dZ^{[2]}} \frac{dZ^{[2]}}{db^{[2]}}$$

$$dW^{[1]} = \frac{dJ}{dW^{[2]}} = \frac{dJ}{dA^{[2]}} \frac{dA^{[2]}}{dZ^{[2]}} \frac{dZ^{[2]}}{dA^{[1]}} \frac{dA^{[1]}}{dZ^{[1]}} \frac{dZ^{[1]}}{dW^{[1]}}$$

$$db^{[1]} = \frac{dJ}{db^{[2]}} = \frac{dJ}{dA^{[2]}} \frac{dA^{[2]}}{dZ^{[2]}} \frac{dZ^{[2]}}{dA^{[1]}} \frac{dA^{[1]}}{dZ^{[1]}} \frac{dZ^{[1]}}{db^{[1]}}$$



Update w and b:

$$W^{[1]} = W^{[1]} - \alpha \frac{dJ}{dW^{[1]}}$$

$$b^{[1]} = b^{[1]} - \alpha \frac{dJ}{db^{[1]}}$$

$$W^{[2]} = W^{[2]} - \alpha \frac{dJ}{dW^{[2]}}$$

$$b^{[2]} = b^{[2]} - \alpha \frac{dJ}{db^{[2]}}$$

Training with SGD and backpropagation

- Randomly initialise parameters w and b
- For iteration 1 .. N; do
  - predict $\hat{y}$ based on w, b and x
  - compute the loss (or cost) J
  - find $\frac{dJ}{dW}$ and $\frac{dJ}{db}$
  - update w and b using dw and db

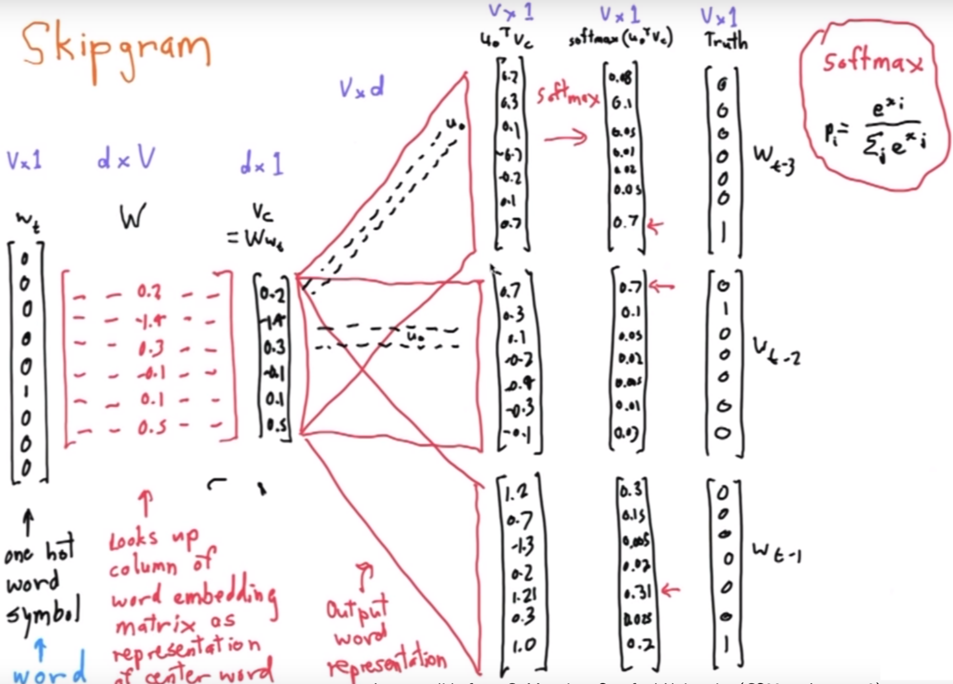
With increasing number of layers in the network: computation complexity increases exponentially

$\Rightarrow$ use dynamic programming

Training with SGD and backpropagation

- Backpropagation:
  - efficient method for computing gradients in a directed computation graph (e.g. a NN)
  - implementation of chain rule of derivatives,
  - allows us to compute all required partial derivatives in linear time in terms of the graph size
- Stochastic Gradient Descent
  - optimisation method, based on the analysis of the gradient of the objective function
- Backpropagation is often used in combination with SGD

Gradient computation: backprop Optimisation: SGD, Adam, Rprop, BFGS, ...



Lecture slide from C. Manning, Stanford University (CS224n, Lecture 2)

Gradient Descent – Sup up

- To minimise $J(\theta)$ over the entire corpus: compute gradients for all windows

- Updates for each element of θ

$$\theta_j^{new} = \theta_j^{old} - \alpha \nabla_\theta J(\theta)$$

- α step size (or learning rate)

Gradient descent is the most basic tool to minimise functions

- But: very inefficient for large corpora!
  Instead: Update parameters after each window t

$\rightarrow$ Stochastic gradient descent (SGD)

$$\theta_j^{new} = \theta_j^{old} - \alpha \nabla_\theta J_t(\theta)$$

Skipgram in a nutshell

- Train a simple neural network with a single hidden layer
- Throw away the network, only keep the learned weights of the hidden layer $\Rightarrow$ word embeddings

Skipgram in a nutshell

- Train a simple neural network with a single hidden layer
- Throw away the network, only keep the learned weights of the hidden layer ⇒ word embeddings
- Limitations of the model
  - Normalisation factor is computationally expensive

$$p(o|c) = \frac{exp(u_o^\top v_c)}{\sum_{w=1}^{V} exp(u_w^\top v_c)}$$

  - Solution: Skipgram with negative sampling (randomly sample “negative” instances from the copurs)