The ACM Conference Series on Recommender Systems

RECSYS | Vancouver, BC | 2018

Concept to Code: Learning Distributed Representation of Heterogeneous sources for Recommendation

Omprakash Sonie
(Flipkart, India)

Sudeshna Sarkar
(IIT Kharagpur, India)

Surender Kumar
(Flipkart, India)

Agenda

- Introduction to Recommendation Systems and Data
- Neural Network and MultiLayer Perceptron
- Distributed representation aka Embedding via prod2vec for RecSys
- Convolutional Neural Network and use in Recsys
- Recurrent Neural Network and use in recommendation
- Modelling, Code walk-through

1. Introduction

- A recommender system identifies and provides items of interest to a user. Tasks:
  - Rating prediction: Recommender system predicts the rating that a user will give to an item
  - Top-K Item recommendation: Given a user and a context rank the items in the order of the likelihood that the user will interact (purchase/click) with them
- Types of recommendation algorithms
  - Collaborative Filtering : based on user item interactions (domain agnostic)
    - Memory based: Nearest neighbours - U2U and I2I similarities
    - Model based: Matrix Factorization
  - Content based: Based on user or item features - regression based methods
  - Hybrid: Combines both - RLFM
- Other features: Temporal | Context-aware| Sequential

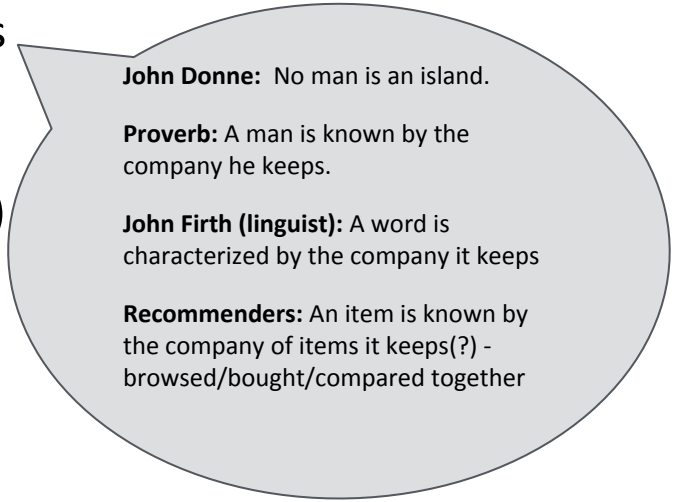
Types of data

Heterogeneity of data in e-commerce

- Text: Reviews
- Numerical/Ordinal: Ratings
- Boolean: implicit feedback - clicks, add to carts, orders
- Binary: Images

Distributed Representation and Traditional methods - I

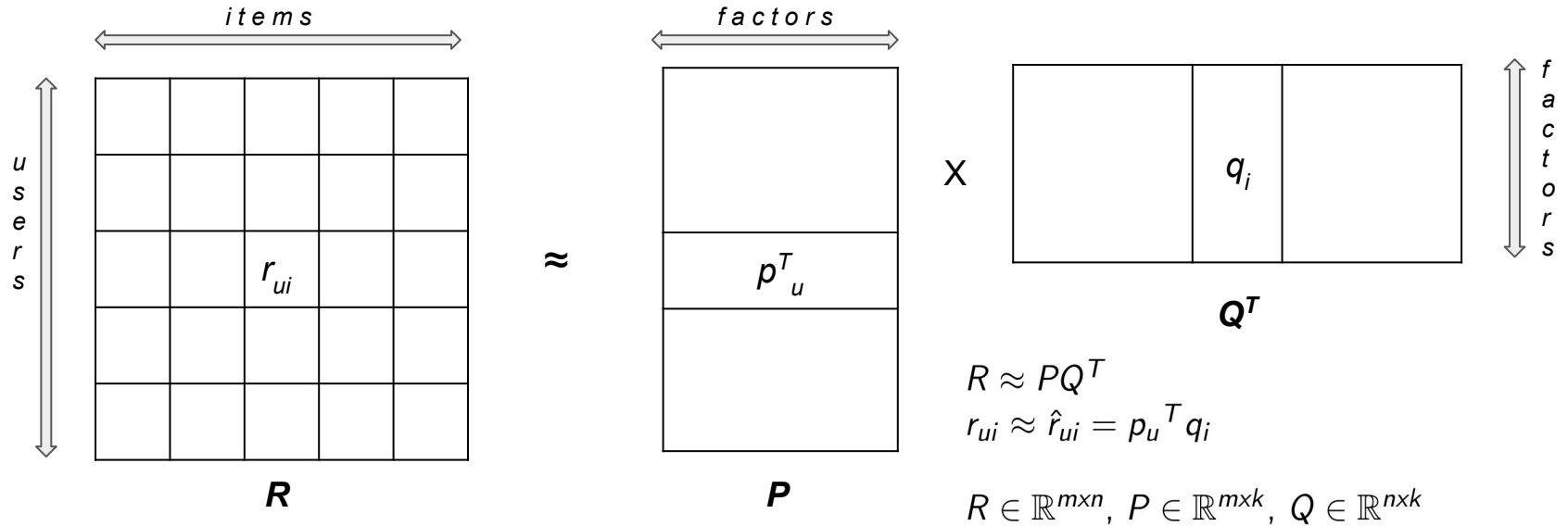
- Data type
    - Explicit - ratings
    - Implicit - items with clicks, add to carts, purchase
- Collaborative Filtering for traditional embeddings a.k.a distributed representations



- SVD like - matrix factorization (low rank approximation)
    - Point wise: L2 loss + L1/L2 regularization
    - Pairwise: Bayesian Personalized Ranking on top of Matrix Factorization
- Probabilistic
    - PMF (Gaussian, Poisson)
    - Mixed effects models (Deepak Agarwal et al*) - subsumes PMF

*https://dl.acm.org/citation.cfm?id=1557029

Traditional Embeddings - Matrix Factorization



$R \approx PQ^T$

$r_{ui} \approx \hat{r}_{ui} = {p_u}^T q_i$

$R \in \mathbb{R}^{mxn}$, $P \in \mathbb{R}^{mxk}$, $Q \in \mathbb{R}^{nxk}$

$p_u \in \mathbb{R}^k$, $q_i \in \mathbb{R}^k$

Optimize via SGD | ALS | ALS-WR

$\min_{P,Q} \sum_{u,i} w_{ui}(r_{ui} - \hat{r}_{ui})^2 + \lambda_1 ||p_u||^2 + \lambda_2 ||q_i||^2$

Probabilistic Matrix Factorization

PMF*: Uncertainty in parameters r_{ui}, p_u and q_i

Distribution via MCMC or point estimates (MAP) by maximizing posterior.

Advantage over other embeddings: Bayesian setting. Multivariate Random variable. Uncertainty in estimates allows for explore-exploit.

$$r_{ui} = p_u{}^T q_i + \epsilon$$
$$\epsilon \sim \mathcal{N}(0, \sigma^2)$$

likelihood: $p(R|P, Q, \sigma^2) = \prod_u \prod_i r_{ui}$

priors: Zero mean spherical Gaussian on factors

$$p(P|\sigma_U^2) = \prod_u p_u;\ p_u \sim \mathcal{N}(0, \lambda_1^{-1})$$
$$p(Q|\sigma_Q^2) = \prod_i q_i;\ q_i \sim \mathcal{N}(0, \lambda_2^{-1})$$

* Ruslan, Andriy : http://papers.nips.cc/paper/3208-probabilistic-matrix-factorization.pdf | http://icml2008.cs.helsinki.fi/papers/600.pdf

Traditional methods - II

- Data: Images
- Application: Similar product recommendations using images
- Traditional computer vision
  - Learn features explicitly (feature detectors)
  - Sobel/Prewitt filter:
    $\begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \Big| \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix}$
  - Cross correlation/convolution as finite difference method for gradient.
  - Horizontal/Vertical Edges
  - Histogram of gradients [HOG]
- Deep learning - distributed/hierarchical representation learning - auto feature detection using CNN. **[Flipkart VisNet*]**
  - learnt filter. No need to predefine the weights. Convolution operation: linear. So add non-linearity.

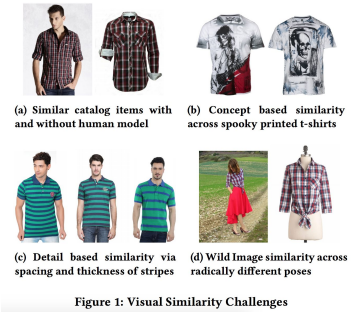
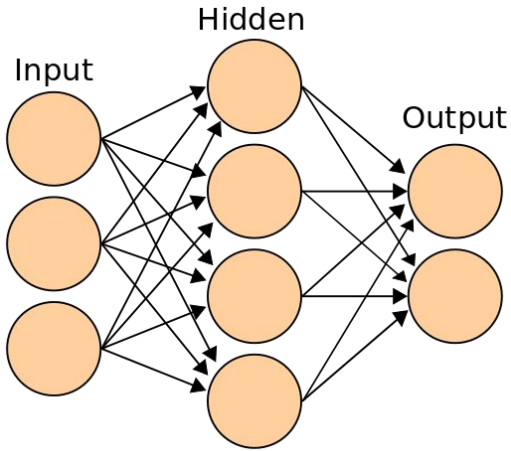


Figure 1: Visual Similarity Challenges

* Visnet @Flipkart: https://arxiv.org/pdf/1703.02344.pdf [Devashish Shankar et al]

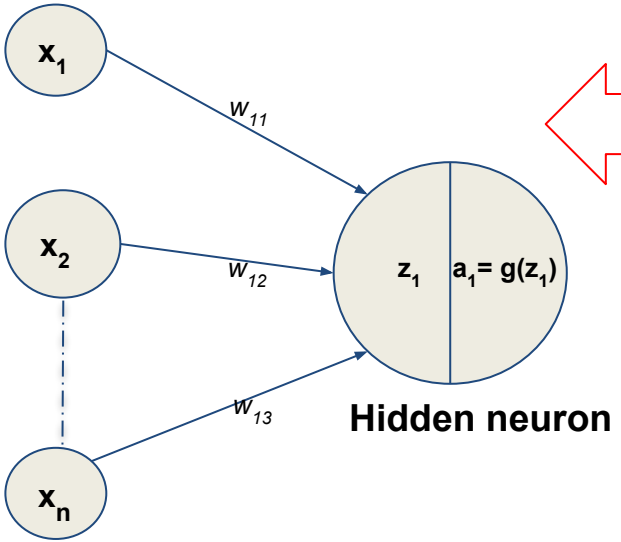
2. Feed forward network

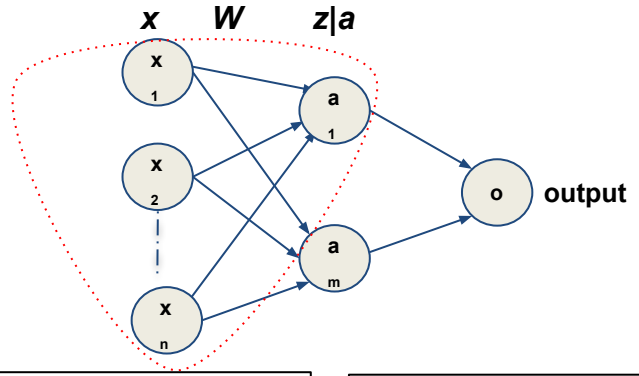
- Connectionism: Loosely based on human brain with neurons and synapses
- An artificial neuron
  - Receives inputs
  - Applies activation - non linearity helps!
  - Produces an output for next layer neurons
- Universal Approximation Theorem
  - Single hidden layer with linear output unit can approximate any continuous function
  - G. Cybenko [1989] | Hornik [1991]
- Connections weights learnt through technique called back propagation

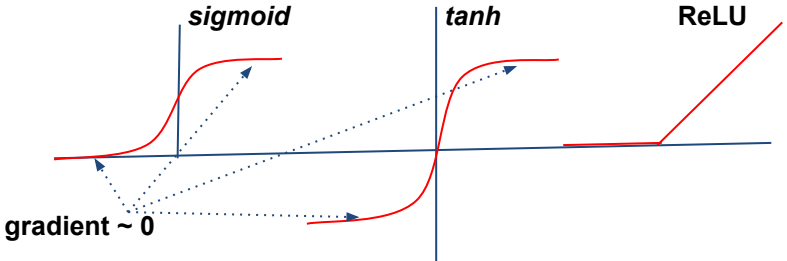


Source:https://en.wikipedia.org/wiki/Connectionism

MultiLayer Perceptron







$z_1 = w_1^T x$

activation $a_1 = g(z_1) = \frac{1}{1+e^{-z_1}}$ (sigmoid)

$z = Wx; \quad a = g(z) = \frac{1}{1+e^{-z}}$

$$W = \begin{bmatrix} -w_1^T x- \\ -w_2^T x- \\ \cdots \\ -w_m^T x- \end{bmatrix} \quad z = [z_1\ z_2\ \ldots\ z_m]^T$$

some other common activation g(z):

tanh: $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

ReLU: $g(z) = max(0, z)$

Leaky ReLU: $g(z) = max(0.01z, z)$

Activation function

1. non-linearity
2. otherwise just a linear combination of input variables
3. Recent success: ReLU
4. Sigmoid in last layer when classification
5. Sigmoid/tanh: vanishing gradient/stuck nodes
6. Leaky ReLU: for -ve o/p

MultiLayer Perceptron - loss functions

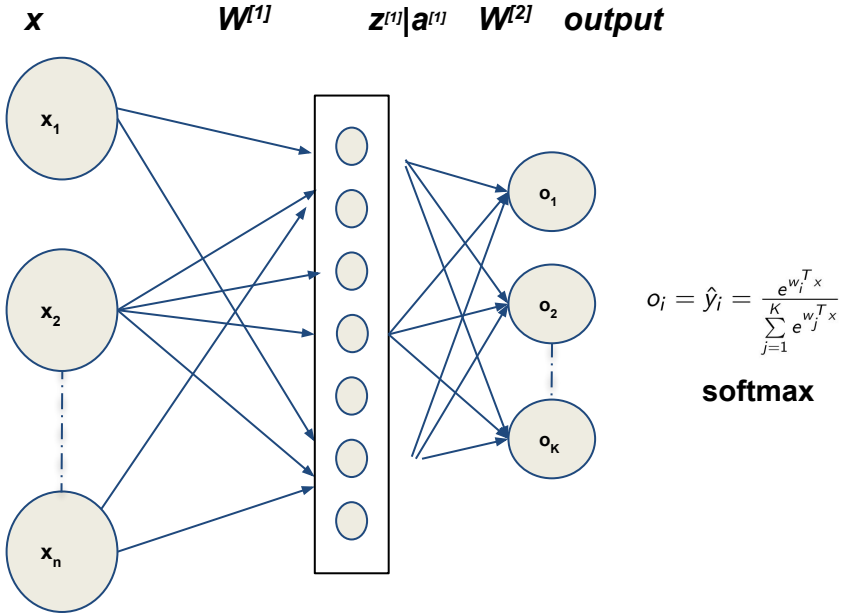
Learn weights by minimizing the loss

Loss between actual and predicted: $\mathcal{L}(y, \hat{y})$

- Regression: Squared loss $\mathcal{L}(y, \hat{y}) = \sum_i (y_i - \hat{y}_i)^2$
- Classification: Cross Entropy $\mathcal{L}(y, \hat{y}) = -\sum_i y_i ln(\hat{y}_i)$

Common $\hat{y}$ for classification: **softmax** (logistic for 2-class)

- Output of a softmax: categorical distribution over K possible outcomes
- $\hat{y}_i = P(y_i = 1|x; W) = \frac{e^{w_i^T x}}{\sum_{j=1}^{K} e^{w_j^T x}}$



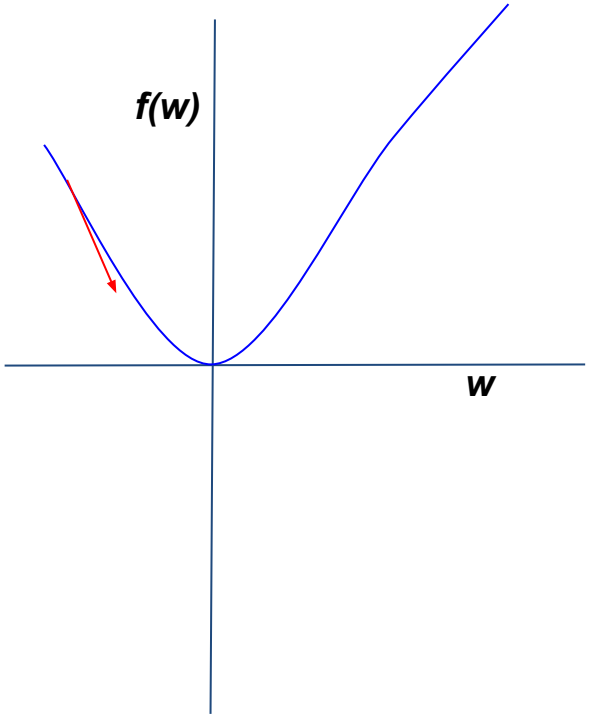
MultiLayer Perceptron - Optimization

Common method of loss minimization - **(Stochastic) Gradient Descent**

Gradient descent: First order method to minimize function $\mathcal{L}(w)$

- **Idea:** Descend against the direction of gradient.
- Linear approximation of the function at x
- Convergence properties studied for convex, Lipschitz function.
- Weight update eqn. $w^{new} = w^{old} - \eta \frac{\partial \mathcal{L}}{\partial w}$
- η : learning Rate
- **Variations:** Momentum(keeps a track of past gradients), Adapative - AdaGrad, RMSProp, Adam (adaptive momentum)
- **Distributed:** Hogwild, Downpour (works in practice)

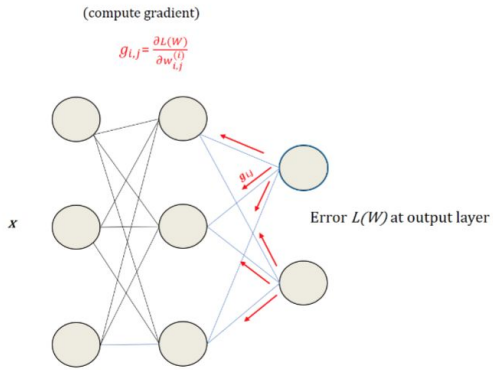
Many other methods: 2^{nd} order - Newton-Rhapson | (L-)BFGS, Conjugate descent, Hessian-free methods ...



MultiLayer Perceptron - Backpropagation

Common method in ANN for parameter learning through gradient descent - **BackPropagation**

Multi-layer ANN



Forward pass:

$z^{[1]} = W^{[1]}x;\ a^{[1]} = g(z^{[1]})$

$z^{[2]} = W^{[2]}a^{[1]};\ a^{[2]} = g(z^{[2]})$

$z^{[3]} = W^{[3]}a^{[2]};\ a^{[3]} = g(z^{[2]})$

Back Propagation:

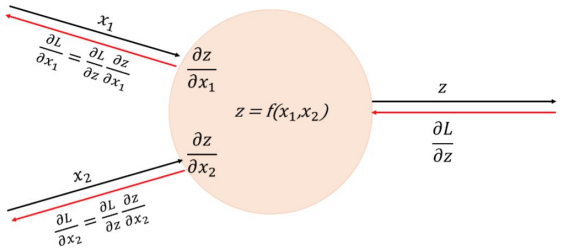
Propagate backwards the error proportionately to each weight

By chain rule of differentiation:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial W^{[2]}}$$

Many gradients are repeated, so care to re-use!



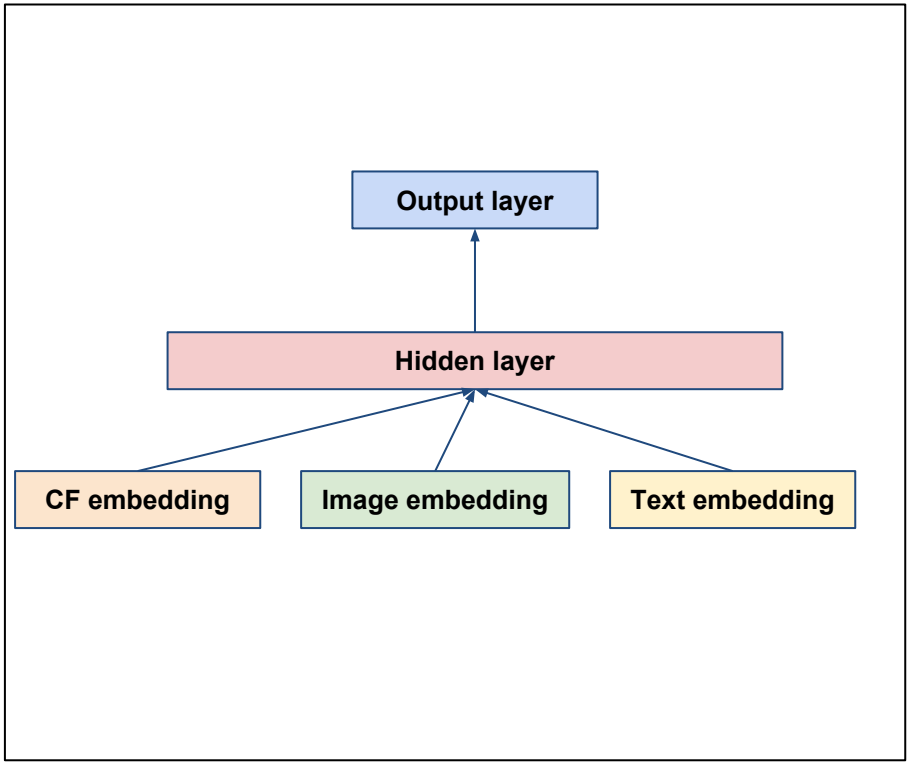
Representation

- Representation plays an important role in Recommender Systems
- Leverage multiple sources of data for rich representation
- Representation of users and items
  - Transactions
  - Content: Product description, Metadata, Reviews, Product Image
  - User demography
  - Product ontology

Using all embeddings

- Multiple ways of learning/combining representations
- **Example 1:** (Adapted from one experiment at Flipkart Recommender System)
  - Concatenate different pre-trained embeddings into a single vector
  - Learn a simple Logistic Regression

    OR
  - Feed into another ANN with softmax output (corresponding to purchase or clicks)
- **Example 2:** Joint learning of the embeddings
- **Similar products** - Locality Sensitive Hashing (LSH) on embeddings



Neural Models of Recommender Systems

- Neural models have become increasingly popular for Recommender Systems.
- Neural networks can be trained to predict ratings or interactions based on item and user attributes.
  - Restricted Boltzmann Machine (RBM)
  - Factorization Machine
  - Neural network models based on content: CNN, RNN
  - Neural models based on sequence of interactions: RNN
  - …
- Representations are important for the models to work

Representation of Items and Users

Representation of items and/or users is necessary for getting a recommender system to work.

- One-hot representation
- Embedding as vectors: pre-trained
- Embedding can be learned on the task

Item Representation

1. Based on item interaction sequences and item properties
   - Prod2vec, Meta Prod2vec etc
2. Based on user-item interaction
   - Neural collaborative filtering: Model user-item interactions
4. User and item information (content)

Product Embedding

Prod2vec or Item2vec : Product embedding

- Based on item-item co-occurrence from transaction sequences

  co-purhased products)
- Uses method of word embedding: low-dimensional, distributed embeddings of words based on word sequences in text documents

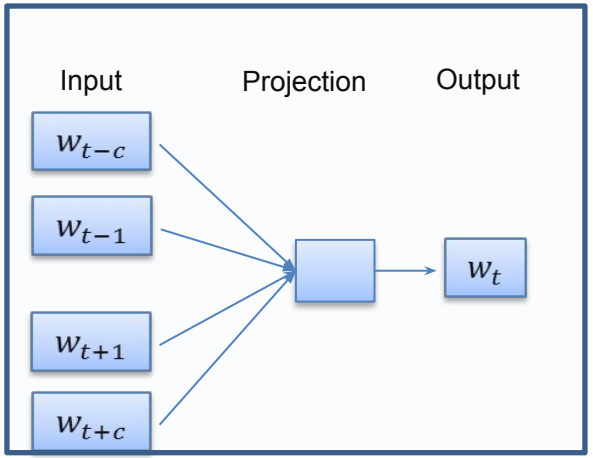
1. Barkan, Oren, and Noam Koenigstein. "Item2vec: neural item embedding for collaborative filtering." Machine Learning for Signal Processing (MLSP), 2016 IEEE 26th International Workshop on. IEEE, 2016.
2. Grbovic, Mihajlo, et al. "E-commerce in your inbox: Product recommendations at scale." *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015.

Word2vec

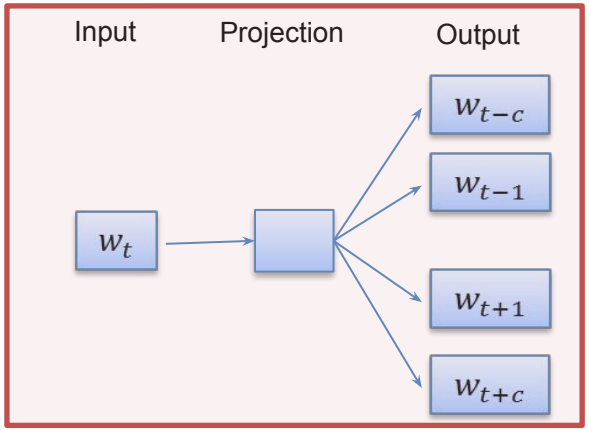
- Representation of words

``Similar words have similar contexts”

1. CBOW: P(Word|Context)

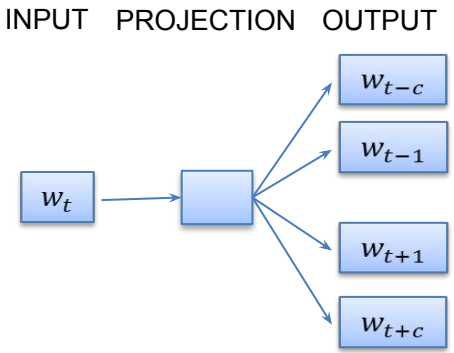


2. Skipgram: P(Context|Word)



Skipgram Model

- Input: Central word w_t
- Output: Words in its context: $\boldsymbol{w_{con}}$
  $\{w_{t-c}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+c}\}$
- Each input word represented by a 1-hot encoding of size V



Source Text:

Deep Learning attempts to learn multiple levels of representation from data.

Input output pairs :

Positive samples:

(representation, levels)

(representation, of)

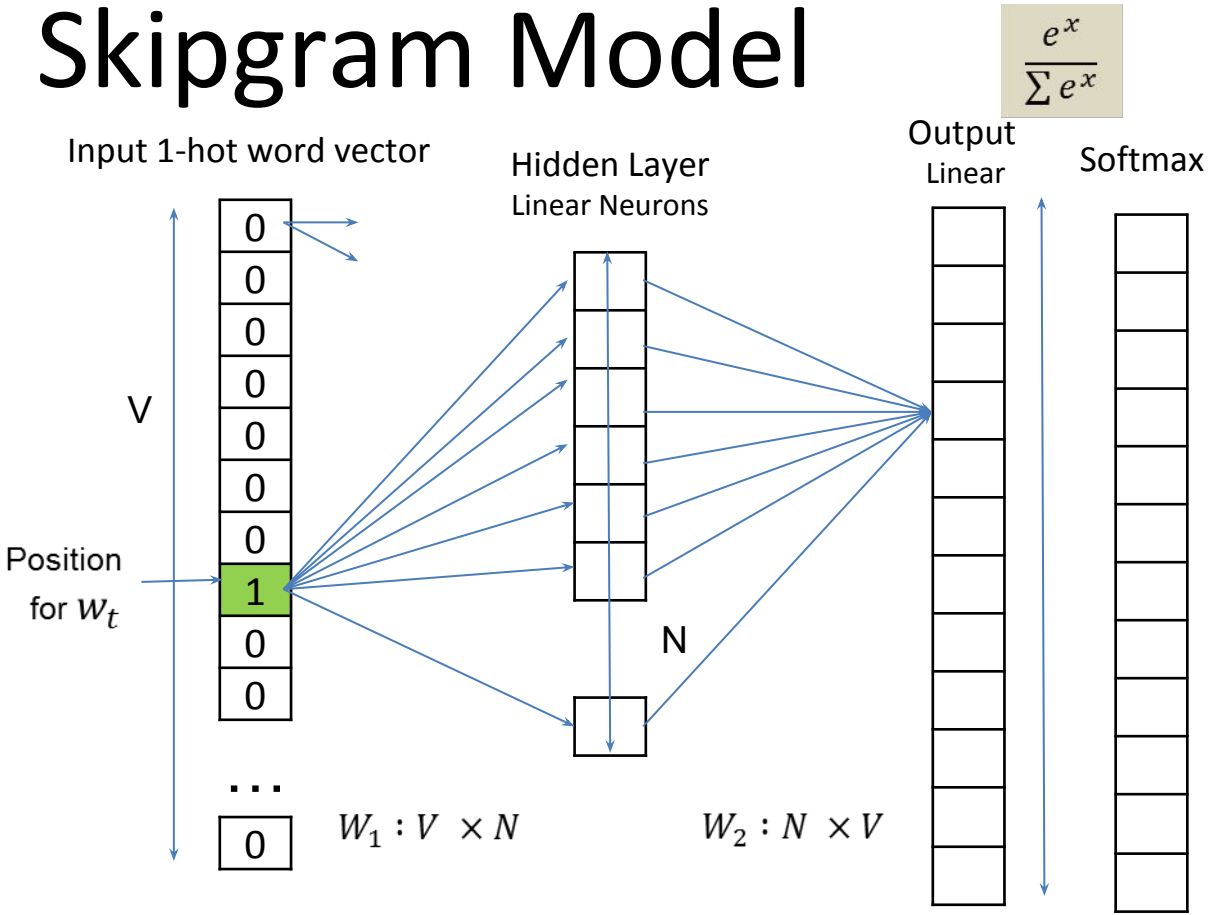
(representation, from)

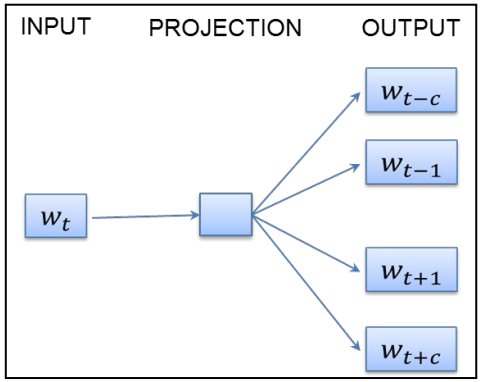
(representation, data)

Negative samples:

(representation, x)
[x: all other words except the 4 positive]

Skipgram Model





Probability that the word in a context position Is w_i

Positive sampling
Negative sampling

Skipgram: Loss function

- Maximize $\frac{1}{T}\sum_{t=1}^{T}\sum_{\text{context}} \log p(w_{\text{context}}|w_t)$

$p(w_{\text{con}}|w_t)$ is the output of softmax classifier

$$p(w_{Ccon}|w_t) = \frac{\exp\left(v'_{w_{con}} . v_{wt}\right)}{\sum_{w=1}^{W} exp(v'_w . v_{wt})}$$

Let the model parameters be θ. The solution is given by

$$\underset{\theta}{argmax} \sum_{(w_t, w_C)\in D} \log p(w_{con}|w_t;\theta)$$

$$= \sum_{(w_t, w_C)\in D} \left(\log e^{(v'_{w_{con}} . v_{wt})} - \log \sum_{x} e^{(v'_x . v_{wt})} \right)$$

Time O(V)
V: vocabulary size

Improve Efficiency
1. Hierarchical softmax:
$O(\log V)$

Skipgram: Loss function

- Maximize $\frac{1}{T}\sum_{t=1}^{T}\sum_{\text{context}} \log p(w_{\text{context}}|w_t)$

$p(w_{\text{con}}|w_t)$ is the output of softmax classifier

$$p(w_{Ccon}|w_t) = \frac{\exp\left(v'_{w_{con}}.v_{wt}\right)}{\sum_{w=1}^{W} exp(v'_w.v_{wt})}$$

Let the model parameters be θ. The solution is given by

$$\underset{\theta}{argmax} \sum_{(w_t,w_C)\in D} \log p(w_{con}|w_t;\theta)$$

$$= \sum_{(w_t,w_C)\in D}\left(\log e^{(v'_{w_{con}}.v_{wt})} - \log\sum_{x} e^{(v'_x.v_{wt})}\right)$$

2. Negative sampling: Sample instead of taking all contexts into account

$$\sum_{(w_t,w_C)\in D}\left(\log\sigma(v'_{w_{con}}.v_{wt}) + \sum_{i=1}^{k}\mathbb{E}_{w_i\sim P_n(w)}\left[\log\sigma\left(-v'_{w_i}.v_{w_t}\right)\right]\right)$$

Subsampling of frequent words

Prod2vec

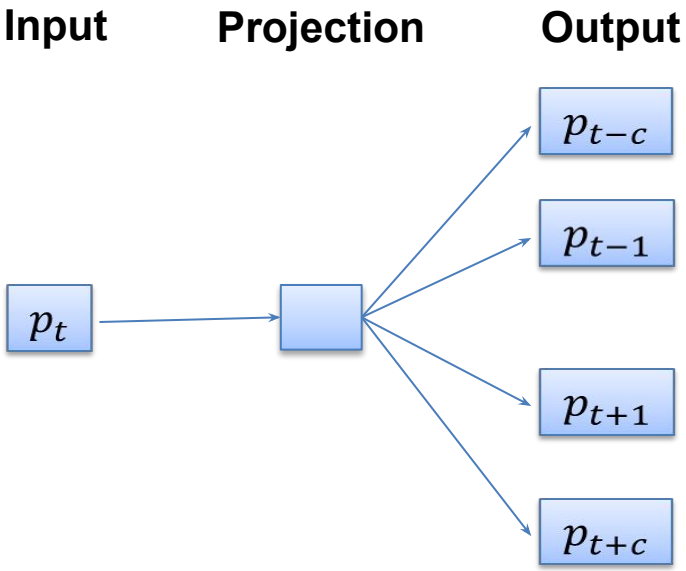
Use word2vec on co-purchased products

Purchase sequence of user *u*

- $p_{u1}, p_{u2}, \cdots, p_{uT}$

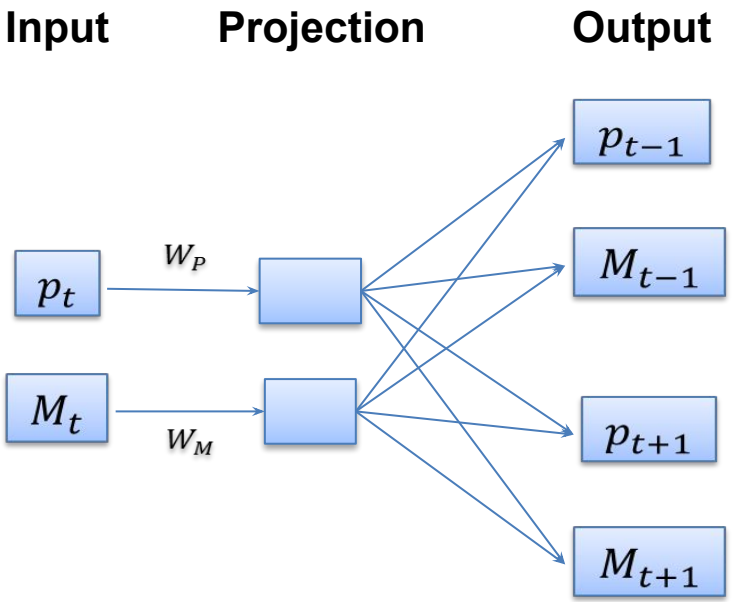
Positive samples:	Negative samples
$(p_{u,t}, p_{u,t-c})$ $(p_{u,t}, p_{u,t-1})$ $(p_{u,t}, p_{u,t+1})$ $(p_{u,t}, p_{u,t+c})$ (p_t, p_{con})	$(p_{u,t}, \mathrm{p})$ p is any product other than p_{con}

Skipgram applied on transaction sequence



Extensions of prod2vec

1. Meta prod2vec
   - Use product metadata in addition to transaction sequence
   - Category, Brand, Description, etc
   - Embedded metadata added both to input and context
3. Several other *2vec models



Recommending using prod2vec

1. Using product embedding directly for recommendation by using product to product similarity

   prod2vec-topK: Given a purchased product, calculate similarities to all other products and recommend the top K most similar products.

2. Using the embedding in different recommendation algorithms

   a) Pre-trained embedding

   b) Modify/ Learn embedding on the task

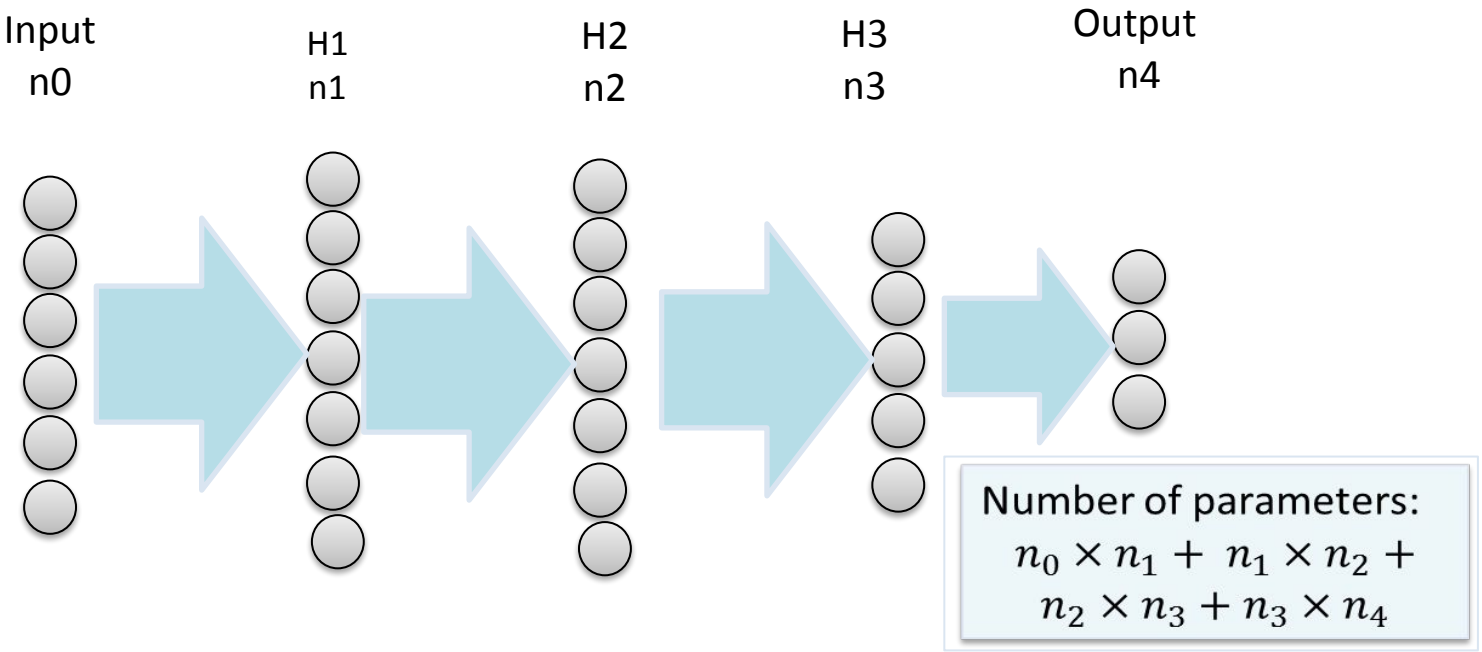
Incorporating Content into Recommendation

- The content of the products may include image or text or audio.
  - Product pictures, videos: Use image embeddings obtained by deep convolutional networks.
  - Product description, Reviews: Text embedding through 1-D CNN or RNN
  - Audio or Music: CNN and RNN

Convolutional Neural Network (CNN)

Fully Connected Network to CNN

- Too many parameters!
- Can some of the parameters be shared?



Local Patterns

1. Local connectivity: Some patterns are local to the input

   Small region -> Fewer parameters

2. Parameter Sharing: Patterns can appear in different regions.

- The sound quality is superb!
- The system has poor audio.

CNN

- A CNN has some convolutional layers. May have Pooling and Fully Connected layers
- A convolutional layer has a number of filters that does convolutional operation
- **Convolution operator**

1-dimensional:

$$(f * g)[x] = \sum_{i=-M}^{M} f[x-i]g[i]$$

2-dimensional:

$$(f * g)[x,y] = \sum_{i=-M}^{M} \sum_{j=-M}^{M} f[x-i, y=j]g[i,j]$$

Convolution

1-d input

3
0
-1
1
0
0
0
1

Filters

1
-1
-1

0.3
-.5
.17

$$(f * g)[x] = \sum_{i=-M}^{M} f[x-i]g[i]$$

2-d input: 8 x 8

3	0	1	2	0	2	-1	2
0	1	0	3	1	0	0	0
-1	-7	1	1	0	0	0	1
1	0	-1	-2	1	0	0	1
0	1	0	0	1	0	0	1
0	0	3	0	1	0	2	1
0	1	2	0	1	-3	0	0
1	-3	0	0	1	1	1	0

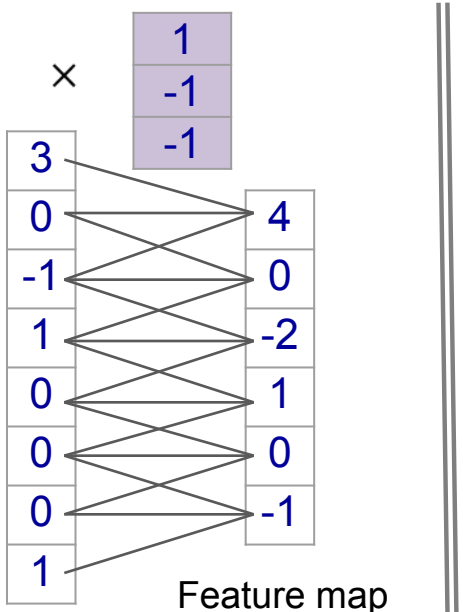
Filters

1	-1	-1
-1	1	-1
-1	-1	1

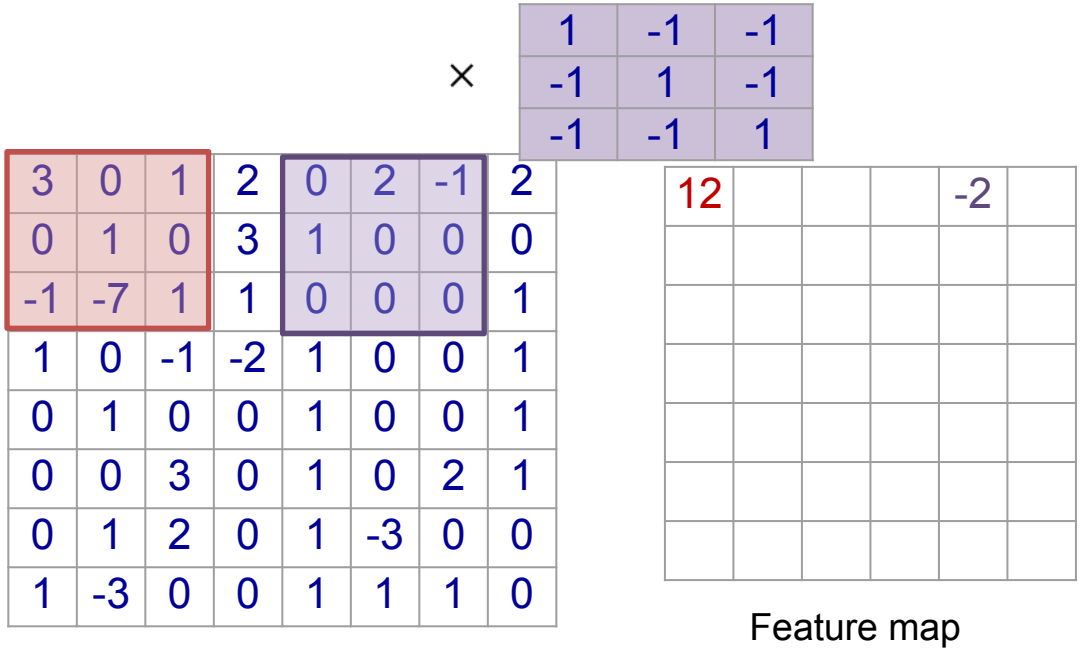
0.3	0.2	-.7
-.5	.21	-.1
.17	.7	-.8

$$(f * g)[x,y] = \sum_{i=-M}^{M} \sum_{j=-M}^{M} f[x-i, y=j]g[i,j]$$

1-D and 2-D Convolution operation

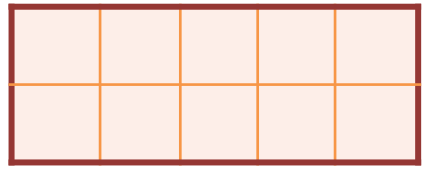


$$(f * g)[x] = \sum_{i=-M}^{M} f[x-i]g[i] + b$$

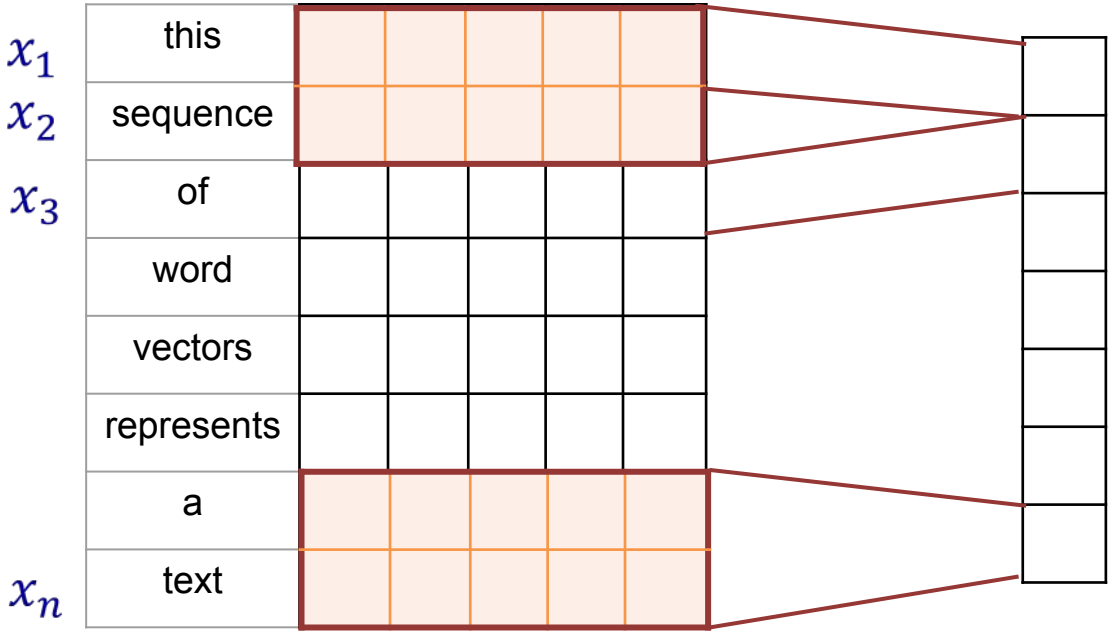


$$(f * g)[x, y] = \sum_{i=-M}^{M} \sum_{j=-M}^{M} f[x-i, y=j]g[i, j] + b$$

Convolution for Text



Filter of size $2 \times k$

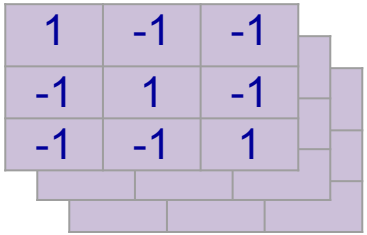


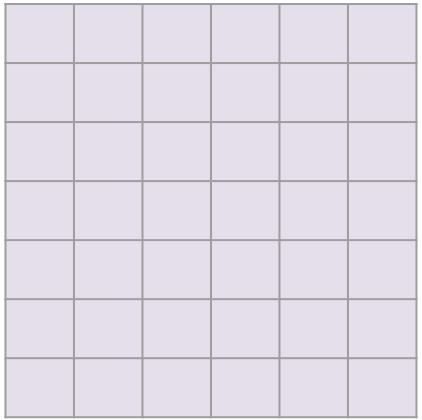
$n \times k$ representation of sentence

$$c_i = f(w \cdot x_{i:i+h-1} + b)$$

Feature map of 1 filter

Multiple channels in input
Depth in layers





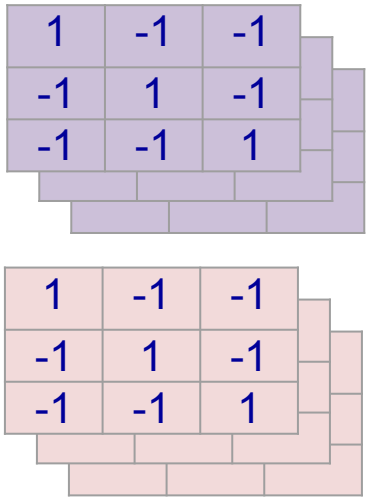
Input: $n \times n \times \mathbf{3}$

Filter: $k \times k \times \mathbf{3}$

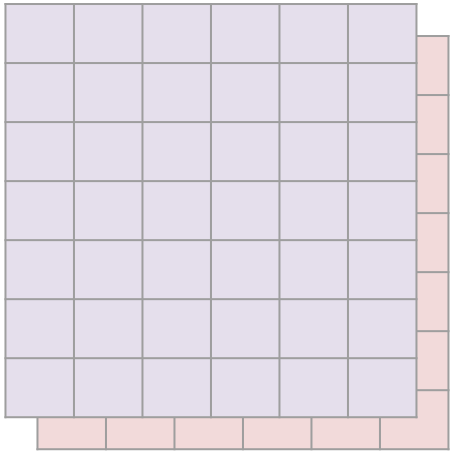
Feature map

Multiple Filters

Input: $n \times n \times 3$

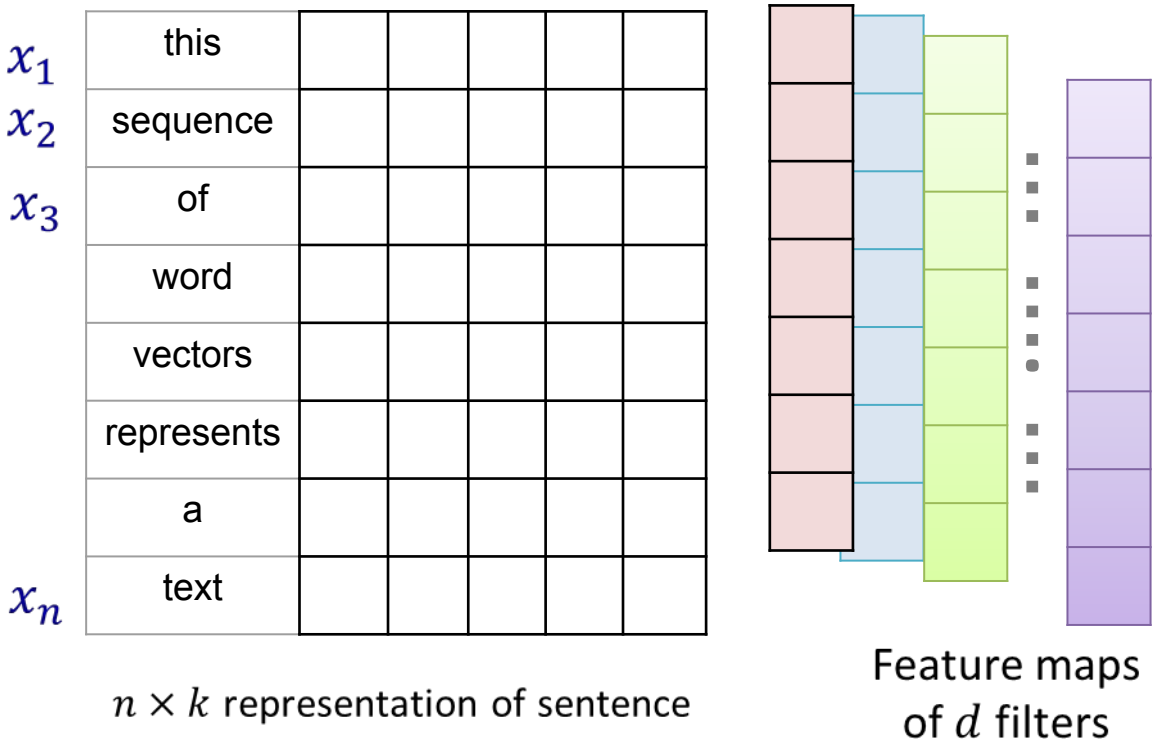


Filter: $k \times k \times 3$



Multiple
Feature maps
(Depth of layer)

CNN for Text


x_1 this
x_2 sequence
x_3 of
word
vectors
represents
a
x_n text
$n \times k$ representation of sentence
Feature maps of d filters

Padding

0	0	0	0	0	0	0	0
0	3	1	1	2	8	4	0
0	1	0	7	3	2	6	0
0	2	3	5	1	1	3	0
0	1	4	1	2	6	5	0
0	3	2	1	3	7	2	0
0	9	2	6	2	5	1	0
0	0	0	0	0	0	0	0

Stride

Stride=2

0	0	0	0	0	0	0	0
0	3	1	1	2	8	4	0
0	1	0	7	3	2	6	0
0	2	3	5	1	1	3	0
0	1	4	1	2	6	5	0
0	3	2	1	3	7	2	0
0	9	2	6	2	5	1	0
0	0	0	0	0	0	0	0

Input size $n \times n$

Filter size $f \times f$

p: padding

s: stride

$$\left(\frac{n+2p-f}{s}+1\right) \times \left(\frac{n+2p-f}{s}+1\right) \times d$$

Pooling

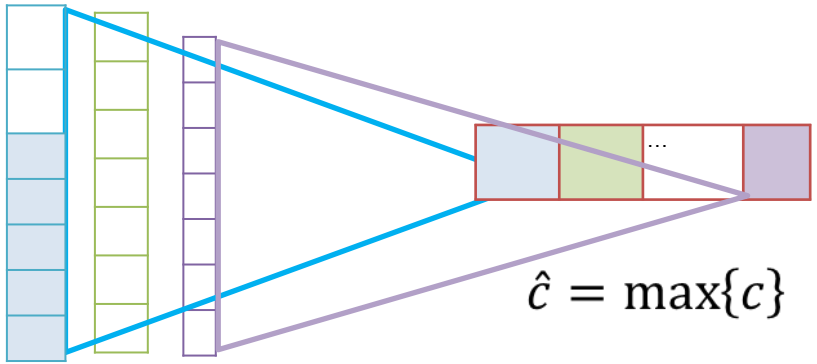
- Summarize the feature maps
  - Aggregate over region
  - MAX, AVERAGE, …
- Translation invariance

this					
sequence					
of					
words					
vectors					
represents					
a					
text					

1	2	-3	-1	0	3	-1	5
2	7	0	1	1	-1	0	2
0	0	1	1	0	-2	0	-1
1	0	0	0	1	-3	0	-1
0	3	0	2	1	0	-1	3
0	0	1	0	1	0	-2	3
-2	1	5	0	1	0	4	1
1	0	0	0	1	1	1	0

Max Pooling

7	5
5	4



Max pooling: Capture the most important feature— the highest valued—for each feature map.

Deep CNN for Images

IMAGENET and the Large Scale Visual Recognition Challenge (ILSVRC)

VGG -16

INPUT	Conv3-256
conv3-64	Conv3-256
conv3-64	maxpool
maxpool	Conv3-512
conv3-128	Conv3-512
Conv3-128	Conv3-512
maxpool	maxpool
Conv3-256	FC-4096
Conv3-256	FC-4096
Conv3-256	FC-1000
maxpool	Soft-max
Conv3-512	

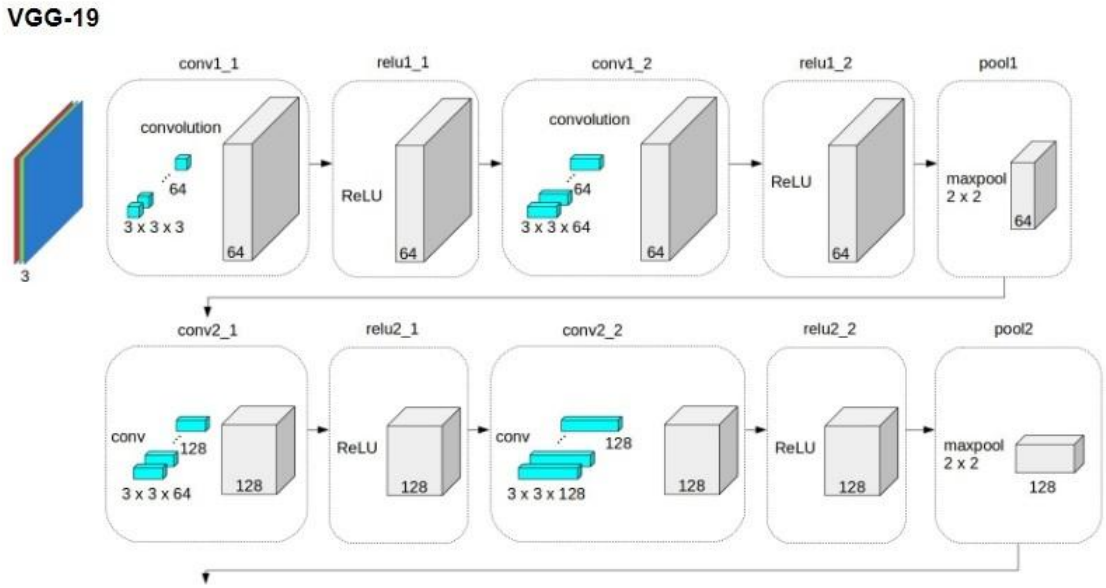
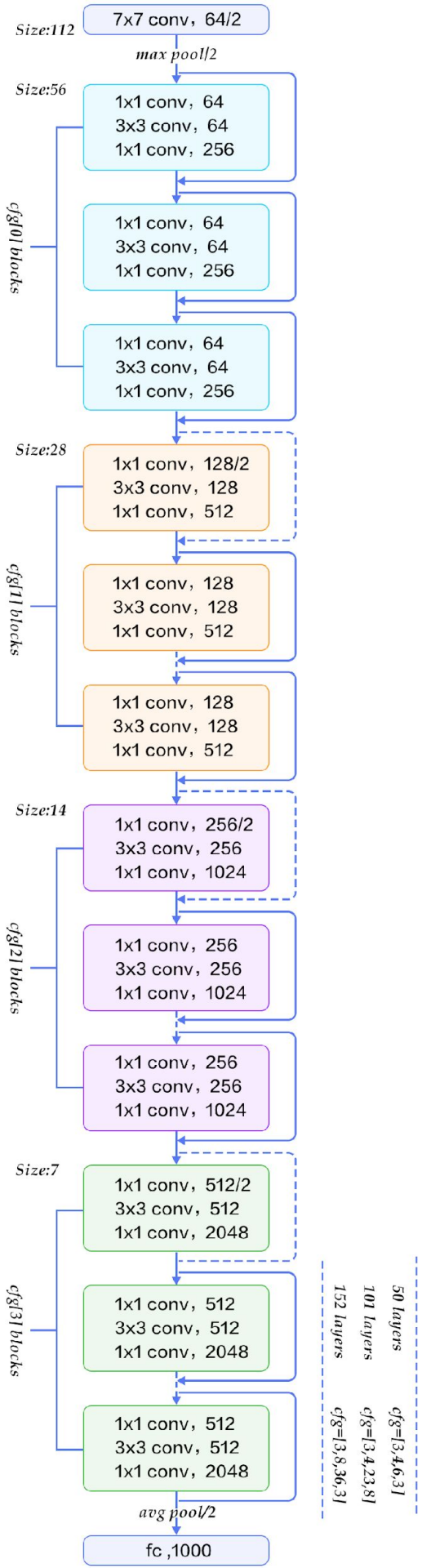
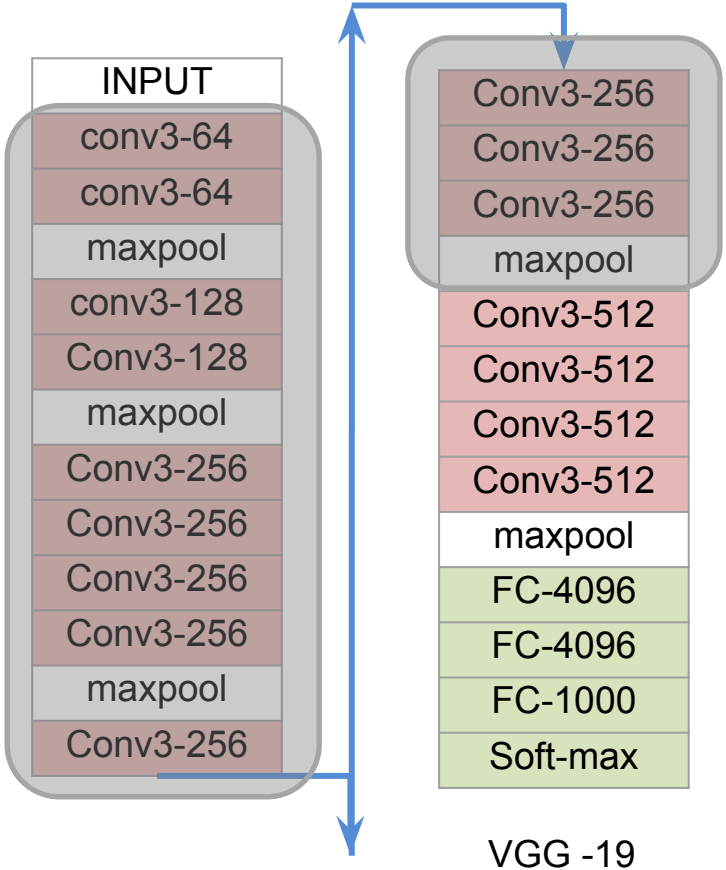


Diagram from http://www.cvc.uab.es/people/joans/slides_tensorflow/tensorflow_html/neural_art.html


Size:112
7x7 conv，64/2
max pool/2
Size:56
1x1 conv，64
3x3 conv，64
1x1 conv，256
cfg[0] blocks
1x1 conv，64
3x3 conv，64
1x1 conv，256
1x1 conv，64
3x3 conv，64
1x1 conv，256
Size:28
1x1 conv，128/2
3x3 conv，128
1x1 conv，512
cfg[1] blocks
1x1 conv，128
3x3 conv，128
1x1 conv，512
1x1 conv，128
3x3 conv，128
1x1 conv，512
Size:14
1x1 conv，256/2
3x3 conv，256
1x1 conv，1024
cfg[2] blocks
1x1 conv，256
3x3 conv，256
1x1 conv，1024
1x1 conv，256
3x3 conv，256
1x1 conv，1024
Size:7
1x1 conv，512/2
3x3 conv，512
1x1 conv，2048
cfg[3] blocks
1x1 conv，512
3x3 conv，512
1x1 conv，2048
1x1 conv，512
3x3 conv，512
1x1 conv，2048
avg pool/2
fc ,1000
50 layers
cfg=[3,4,6,3]
101 layers
cfg=[3,4,23,3]
152 layers
cfg=[3,8,36,3]

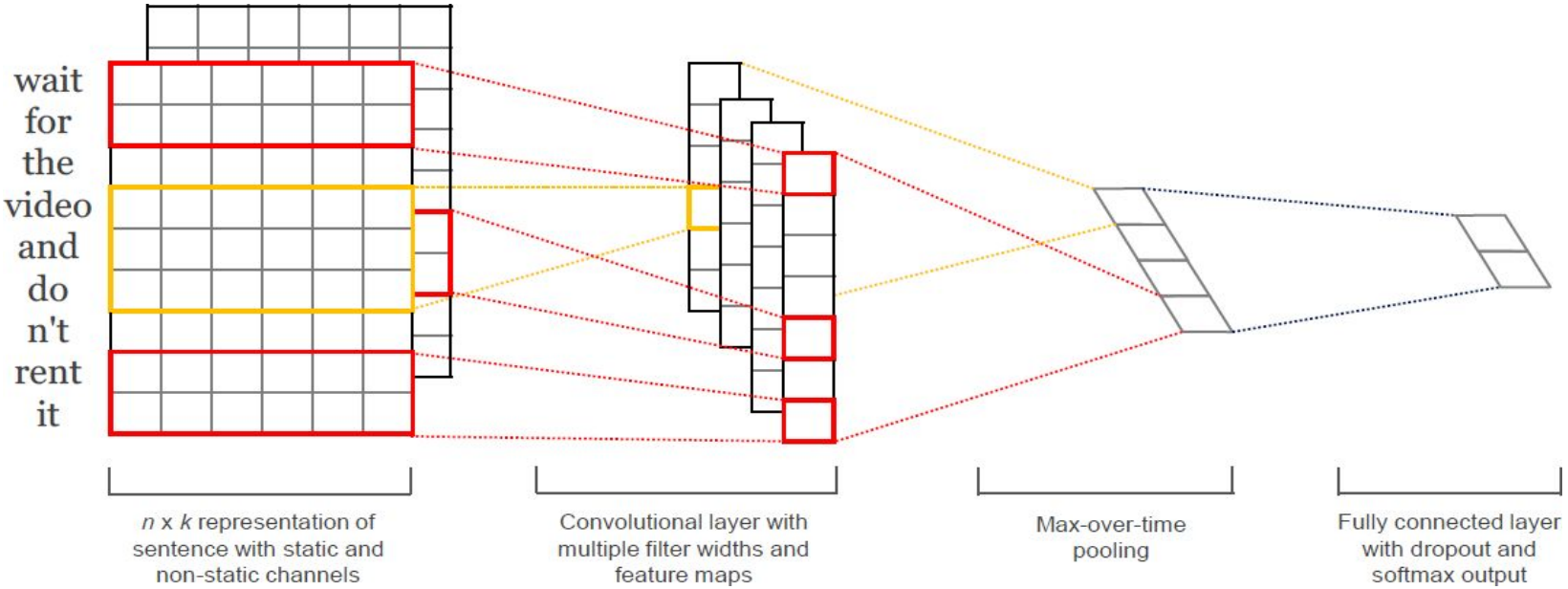
Transfer Learning from Pre-trained Network



VGG -19

- Pretrained from Imagenet
- Freeze some initial layers
- Train few layers near the output
- Tune the whole network

CNN for Text: Kim's CNN



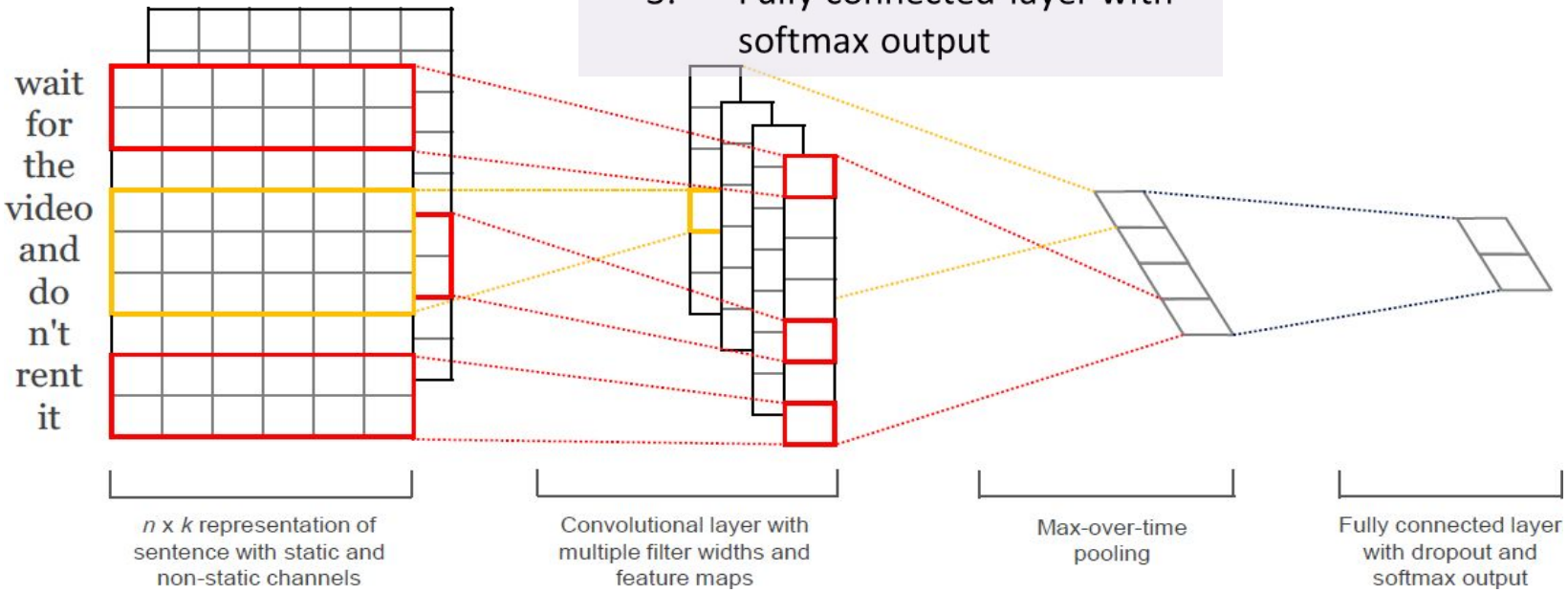
Kim, Yoon. "Convolutional neural networks for sentence classification." *arXiv preprint arXiv:1408.5882* (2014).

Kim's CNN

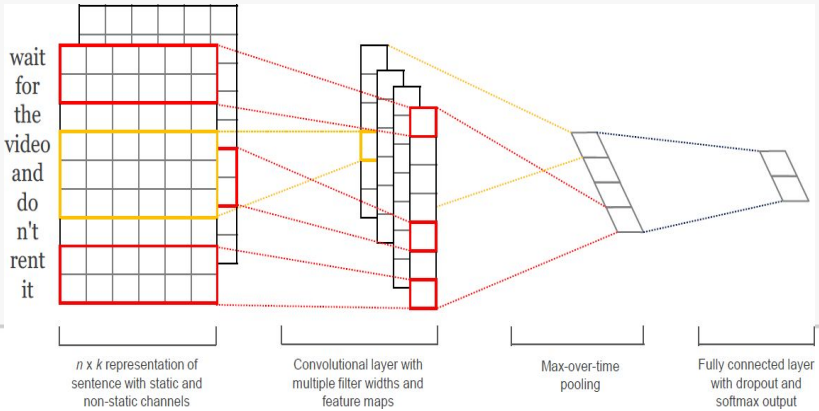
- Word embedding
- Padding to size *n*

1. Convolution layer
2. Max-pooling layer
   $\hat{c} = \max\{c\}$
3. Fully connected layer with softmax output

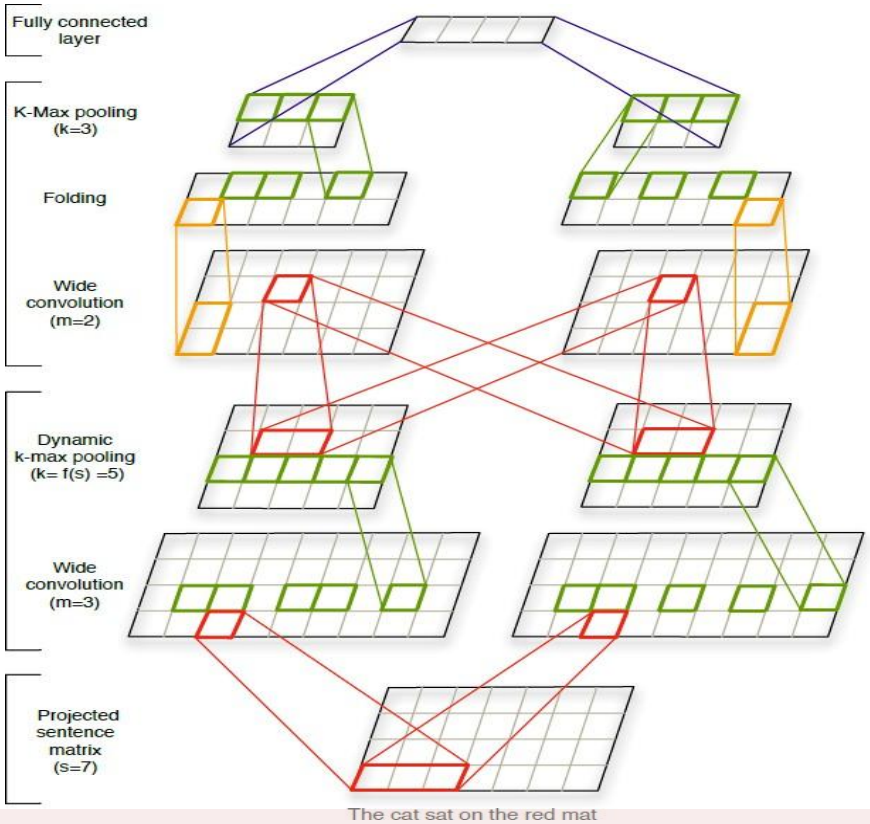
Dropout
Rescale the weight vectors of each class to fixed number s



```
text_seq_input = Input(shape=(MAX_TEXT_LEN,), dtype='int32')
text_embedding = Embedding(vocab_size, WORD_EMB_SIZE, input_length=MAX_TEXT_LEN,
filter_sizes = [3,4,5]
convs = []
for filter_size in filter_sizes:
        l_conv = Conv1D(filters=128, kernel_size=filter_size, padding='same', activation='relu')(text_embedding)
       l_pool = MaxPool1D(filter_size)(l_conv)
       convs.append(l_pool)
l_merge = Concatenate(axis=1)(convs)
l_cov1= Conv1D(128, 5, activation='relu')(l_merge)
# since the text is too long we are maxpooling over 100
l_pool1 = MaxPool1D(100)(l_cov1)
l_flat = Flatten()(l_pool1)
l_dense = Dense(128, activation='relu')(l_flat)
l_out = Dense(9, activation='softmax')(l_dense)
model_1 = Model(inputs=[text_seq_input], outputs=l_out)
```



Dynamic k-max pooling



- Handles sentences of varying length
- k is a function of the sentence length and network depth

$$k_l = \max\left(k_{top}, \left\lceil \frac{L-l}{L} \cdot s \right\rceil\right)$$

l : current convolution layer

L : Number of convolutional layers

s: Sentence length

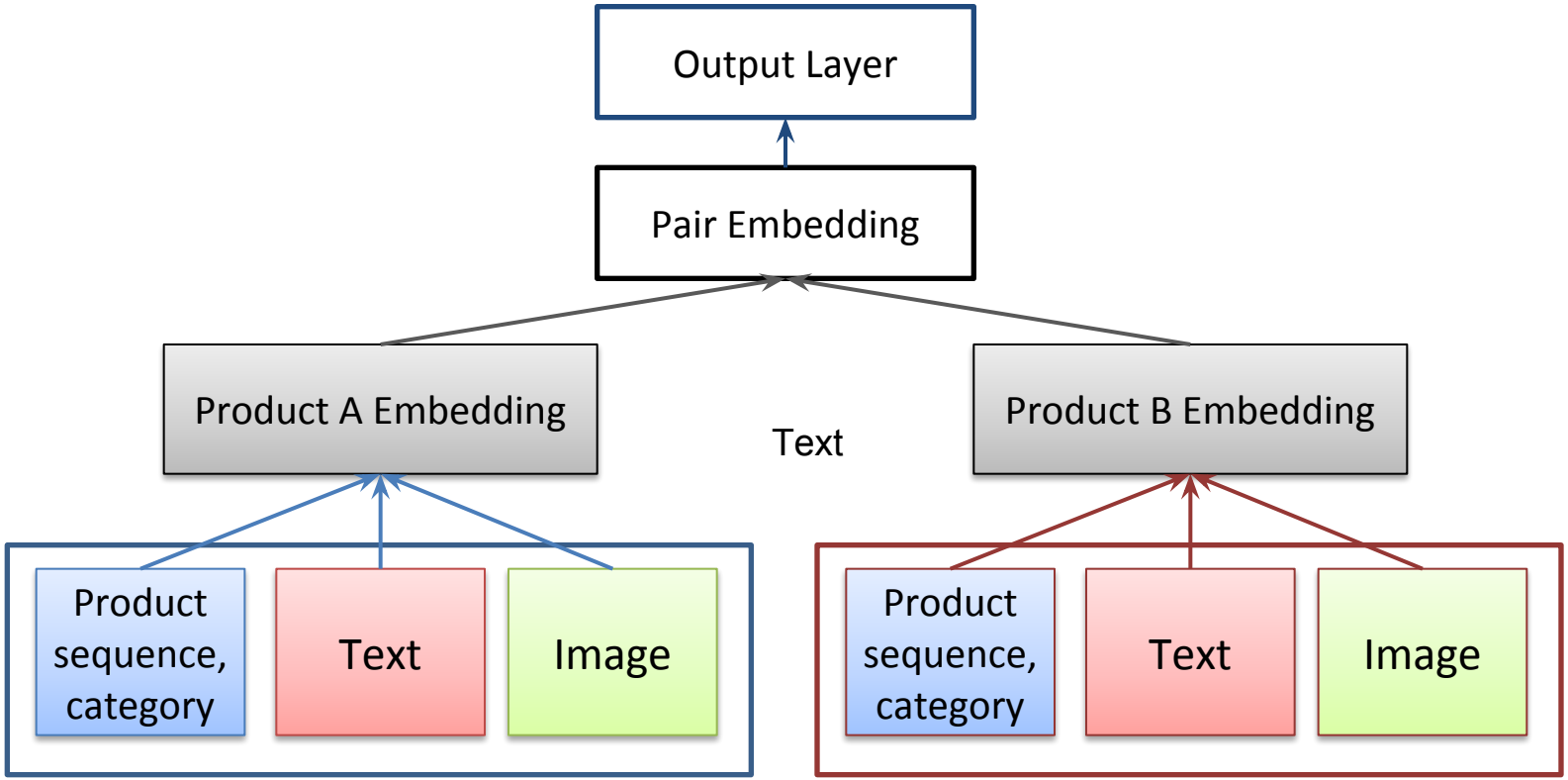
k_{top} : fixed pooling parameter for the topmost convolutional layer

Neural Network for Modeling Sentences" by Nal Kalchbrenner, Edward Grefenstette, Phil Blunsom arXiv, 2014

Enrich embedding with content

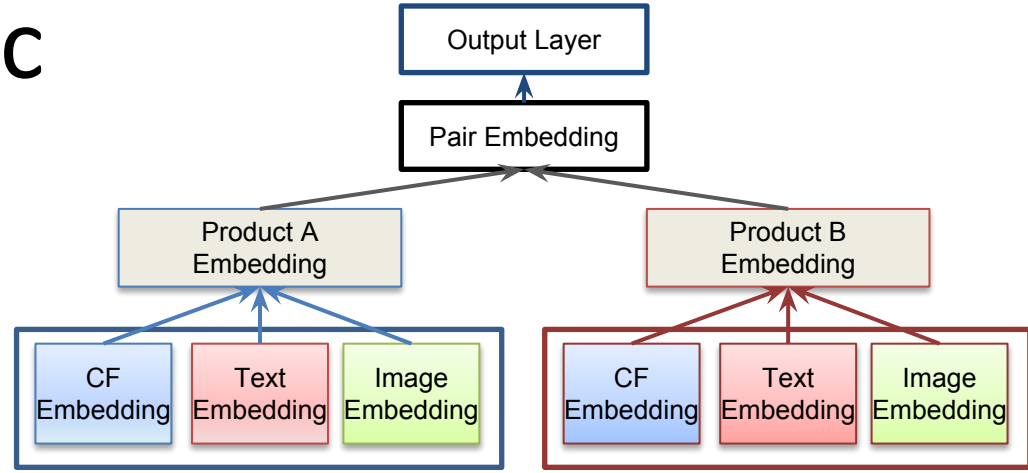
- Use content of object for content-specific embedding
- Content2vec
  1. Initialize the content-specific modules with embeddings from proxy tasks (classification for image, language modeling for text)
  2. Combine the multi-modal embeddings into a general product vector
  3. Tune network to the task of product recommendation: Define the product to product similarity based on co-purchase

Nedelec, Thomas, Elena Smirnova, and Flavian Vasile. "Content2vec: Specializing joint representations of product images and text for the task of product recommendation." (2016).


Output Layer
Pair Embedding
Product A Embedding
Text
Product B Embedding
Product sequence, category
Text
Image
Product sequence, category
Text
Image

Content2vec

- Text Embedding: Word2vec and Kim's CNN
- Image Embedding: AlexNet (Pre-trained from Imagenet)
- Product transaction sequence: Product2vec
- Product category: Meta Product2vec



Pair Embedding:

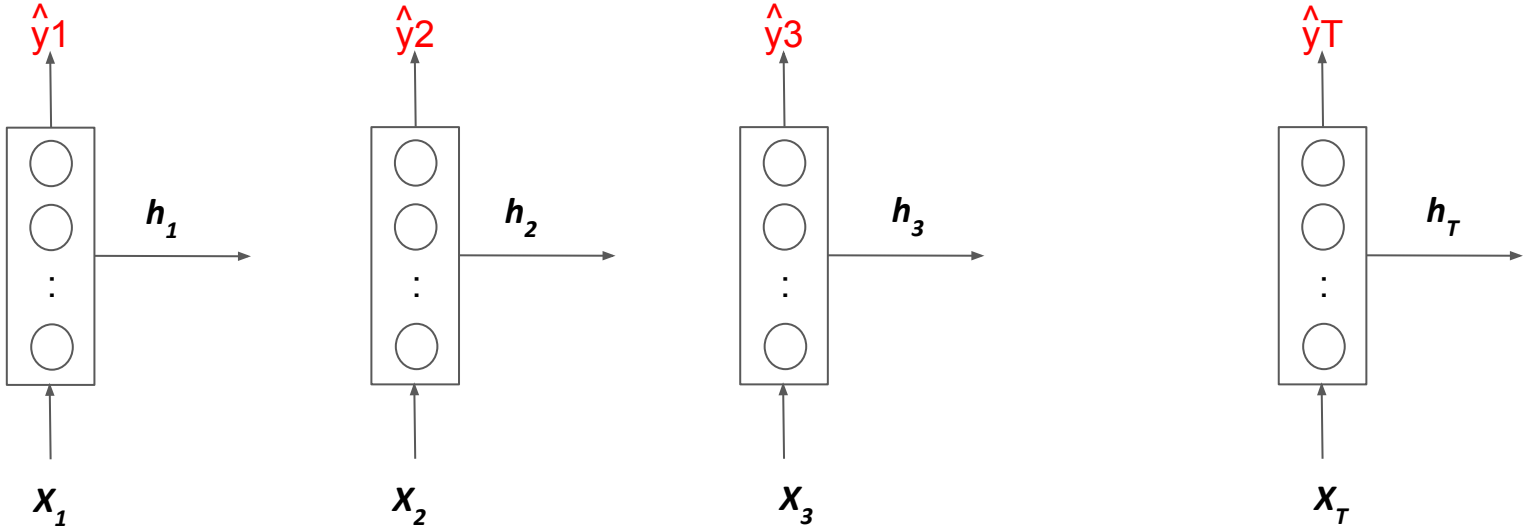
1. Pairwise Residual Unit
2. Embedded Pairwise Residual Unit

Recurrent Neural Network (RNN)

Motivation

- Traditional Neural Networks
  - Can’t handle sequential data and do not build on previous events
- Handle sequential data and capture long term dependencies
  - Product purchase sequence
  - User activity - Click / View sequence
  - Text
  - Audio
  - Sentiment Classification

RNNs: Introduction



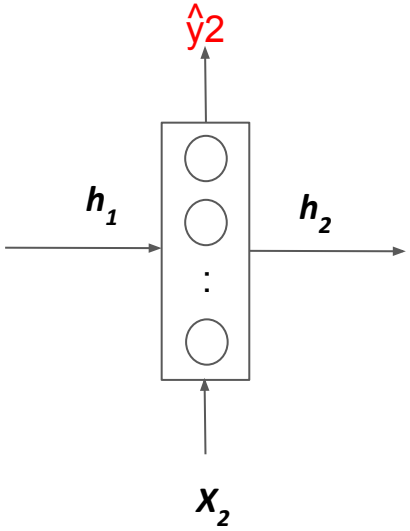
User activity	Product 1	Product 2	Product 3	…..	Product k
timestamp	1	2	3		T

$\hat{y}2$ uses information from X_2 and X_1

$\hat{y}3$ uses information from X_3, X_2 and X_1

But not from later sequence (X_4, X_5 etc.)

RNNs: Introduction



Hidden state - h_2 = some function-g1 (weighted addition of h_1 and X_2 + bias-h)

Output-$\hat{y}2$ = some function-g2 (weighted hidden state h_2 + bias-y)

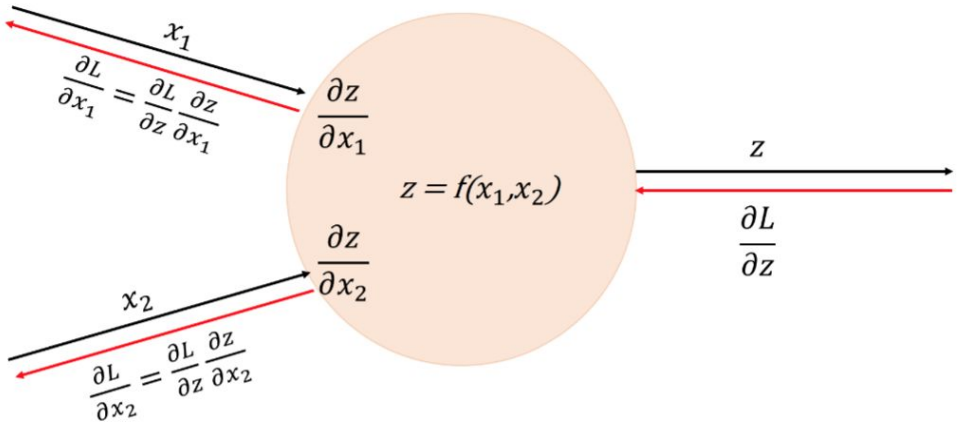
In General:

Hidden output- h_t = some function-g1 (weighted addition of h_{t-1} and X_t + bias-h)

Output-$\hat{y}t$ = some function-g2 (weighted hidden state h_t + bias-y)

Weights and Biases are shared

MultiLayer Perceptron - Backpropagation



Back Propagation:

Propagate backwards the error proportionately to each weight

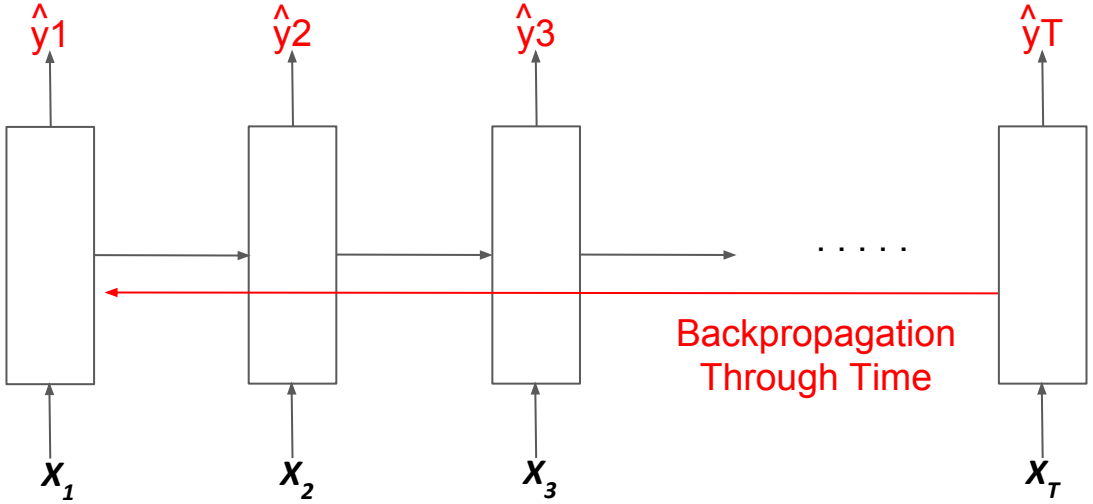
By chain rule of differentiation:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial W^{[2]}}$$

Vanishing Gradient

Timestamp = number of layers (for example 100)

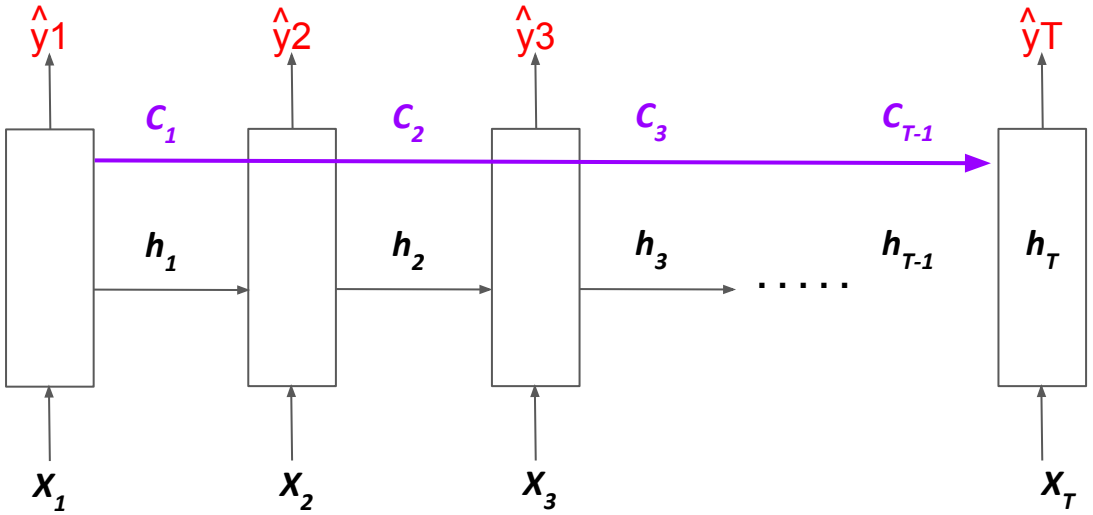


For learning long term dependencies

- Pass cell sate (or memory) through the entire chain
- Ability add or remove information to cell state

Long Short Term Memory (LSTMs)

- Provide highway to pass cell state (or memory) is passed through the entire chain
- Have gates to add or remove information to cell state, hidden state and output



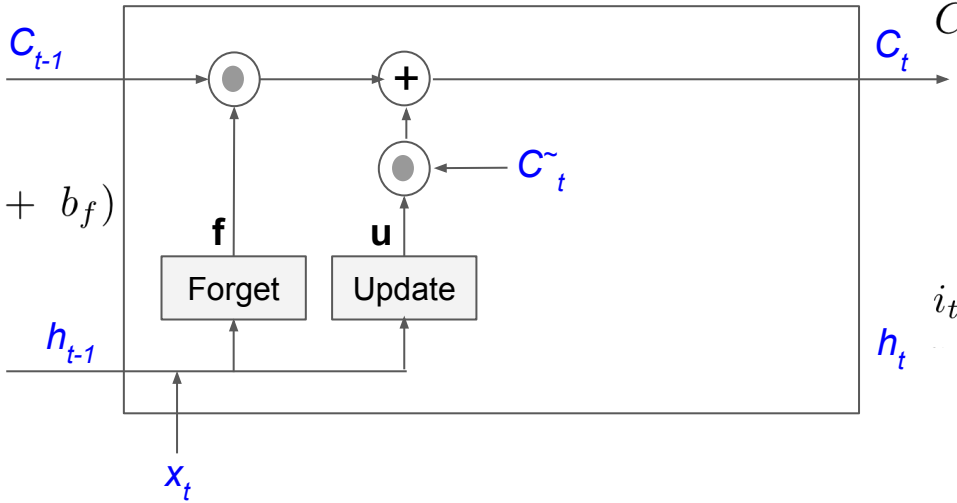
LSTMs - Gates

- Has three gates to control information flow
- **Forget gate:**
  - How much of cell state value should be carried forward (nothing (zero) to pass as it is (one))
- **Update gate:**
  - Update old state with new state
- **Output gate:**
  - What part of the cell state and how much we are going to output

LSTMs – Details (1/3)

- **Forget gate (f):** Function of (weighted-f sum of previous hidden state (h_{t-1}) and current input (x_t) + bias-f)
- **Update gate (u):** Function of (weighted-u sum of previous hidden state (h_{t-1}) and current input (x_t) + bias-u)
- **Output state (ct):** Forget gate and Update gate control previous cell state (c_{t-1}) and current internal state (c~) respectively

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

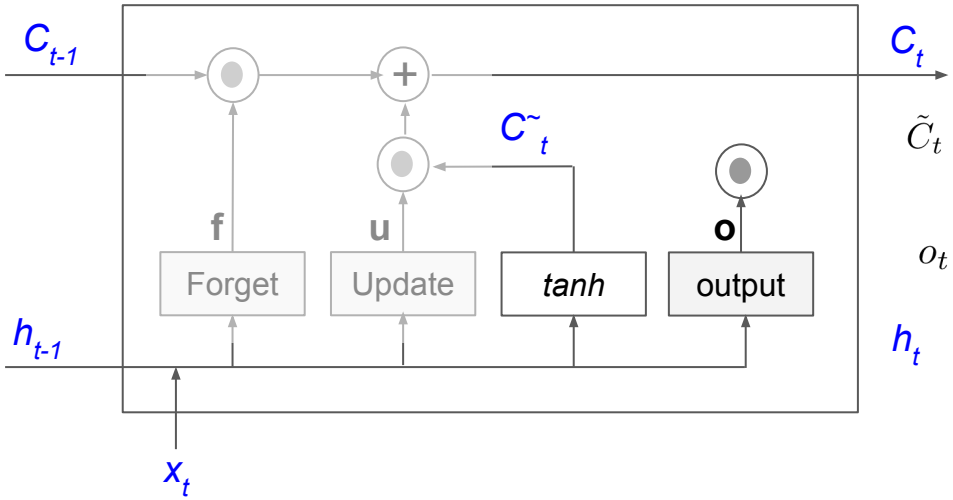


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

LSTMs – Details (2/3)

- **Internal current state (ct~):**
  - Function of (weighted-c sum of previous hidden state (h_{t-1}) and current input (x_t) + bias-c)
- **Output gate (o):** Function of (weighted-o sum of previous hidden state (h_{t-1}) and current input (x_t) + bias-o)

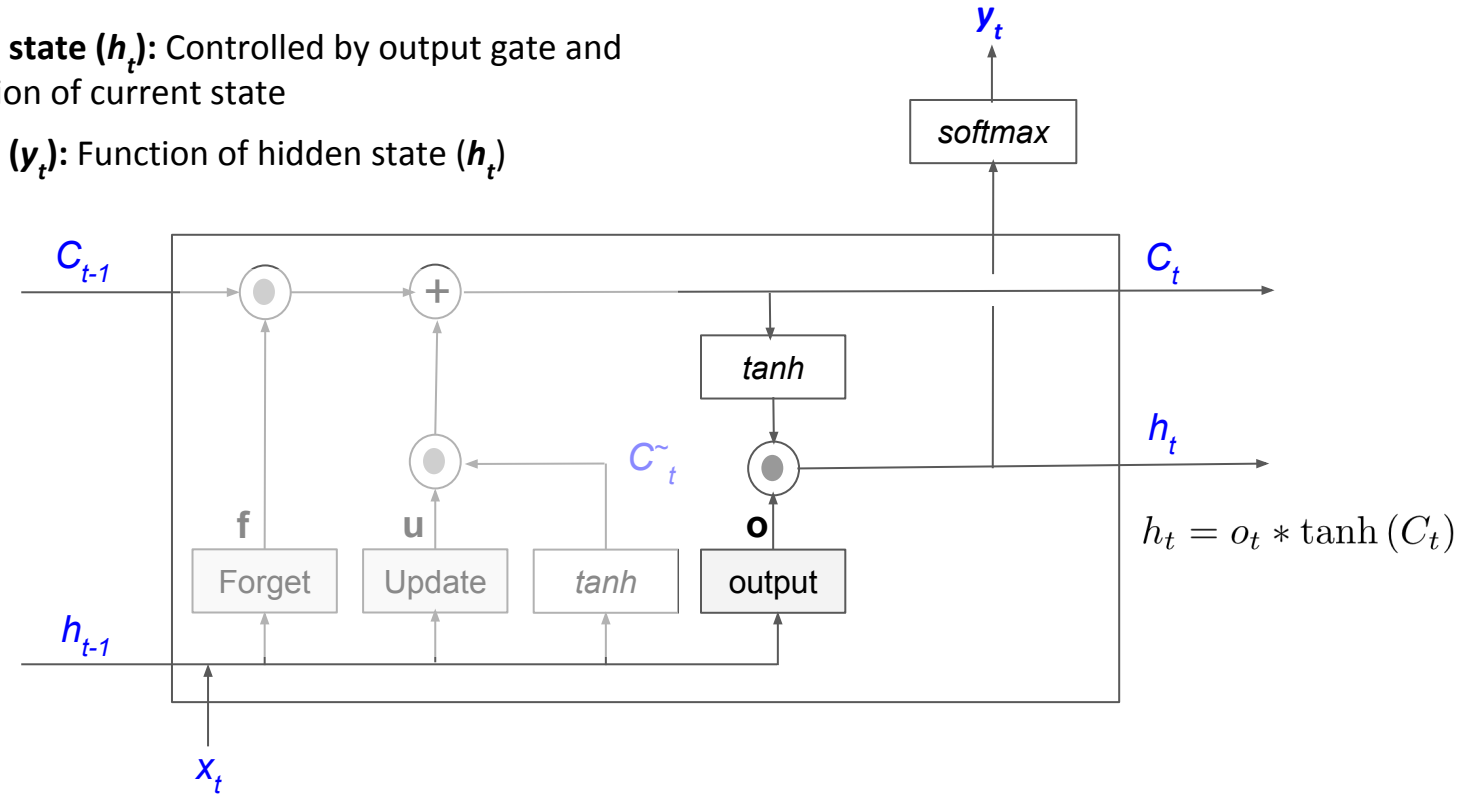


$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

$$o_t = \sigma\,(W_o \; [\, h_{t-1}, x_t] \; + \; b_o)$$

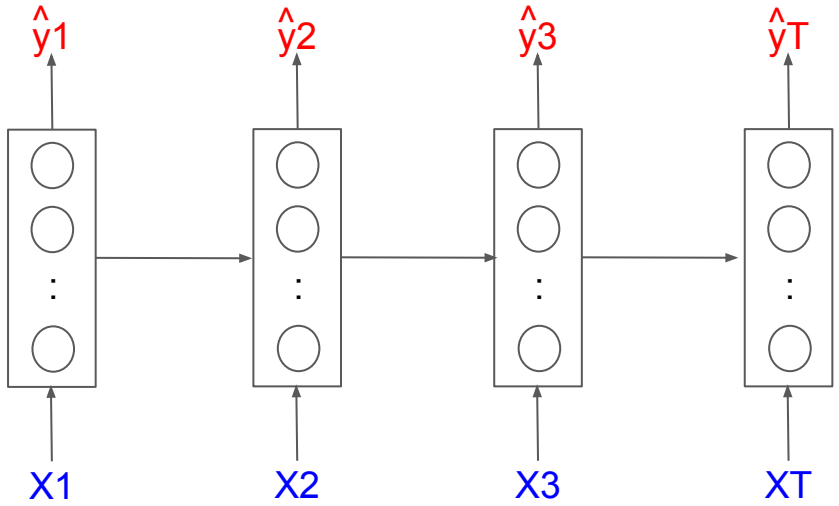
LSTMs – Details (3/3)

- **Hidden state (h_t):** Controlled by output gate and a function of current state
- **Output (y_t):** Function of hidden state (h_t)



Predicting Next Purchase

- Treat recommendations as a sequence classification problem.
- Input: sequence of user actions. Output: next action
- Input may be item/product id or item/product embedding

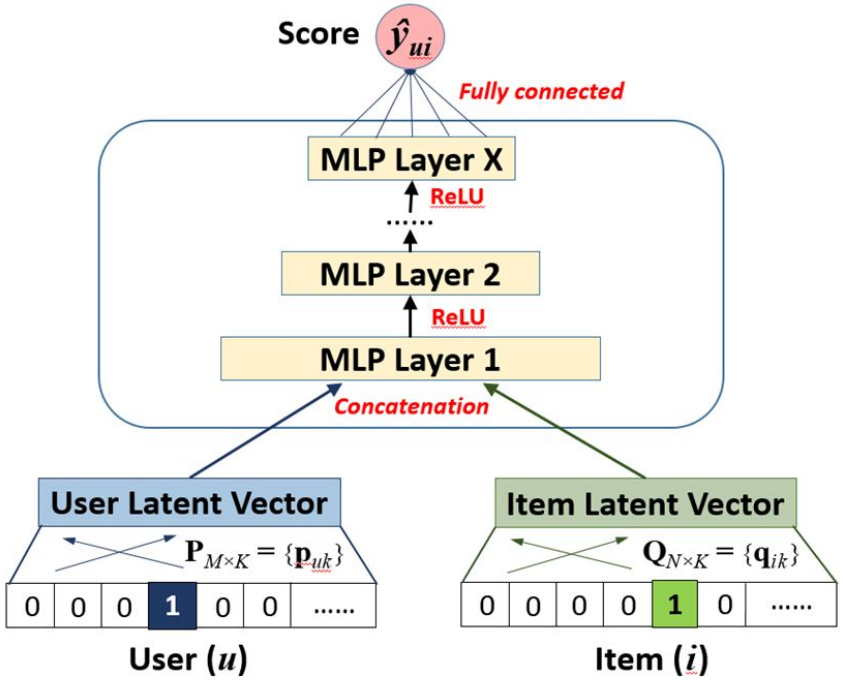


Modelling and Code Walkthrough

The code will be available on:
www.DeepThinking.ai
www.c2cRecSys.com

Multilayer Neural model

- $\hat{y}_{ui} = f(p_u, q_i)$
- NCF uses a multi-layer model to learn the user-item interaction function
- Can capture the complex structure of user interaction data
- NCF can endow more nonlinearities to learn the interaction function:



Neural Collaborative Filtering: He Xiangnan1, Liao Lizi1, Zhang Hanwang1, Nie Liqiang2, Hu Xia3, Tat-Seng Chua1, WWW 2017

Setup

Environment:

- Python 2.7.15
- Tensorflow 1.3.0
- Keras 2.0.8

Datasets

1. Rating
2. Images
3. Reviews

Simple Keras Model

In five steps.

In rest of the code walk through, we will focus on specific APIs for learning embeddings

```
# Binary Classification: Simple Model
from keras import layers

#1. Define
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

#2. Compile
model.compile(optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['accuracy'])

# Generate partial_x_train, partial_y_train,
                    x_val, y_val, x_test, y_test dataset
#3. Learn
history = model.fit(partial_x_train, partial_y_train,
    epochs=20, batch_size=512,
    validation_data=(x_val, y_val))

#4. Test
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)

#5. Predict:
model.predict(x_test)
```

1. Ratings - Dataset

Amazon Beauty data: 5-core_(198,502 reviews) http://jmcauley.ucsd.edu/data/amazon/links.html
file: reviews_Beauty_5.json


```
{

    "reviewerID": "A3G6XNM240RMWA",
    "asin": "7806397051",
    "overall": 4.0,
    "reviewerName": "Karen",
    "helpful": [0, 1],
    "summary": "great quality",
    "unixReviewTime": 1378425600,
    "reviewTime": "09 6, 2013"
    "reviewText": "The texture of this concealer pallet is fantastic, it has great coverage
     and a wide variety of uses, I guess it's meant for professional makeup artists and a lo
    t of the ... gives me a natural for and concealed my imperfections, therefore I highly r
    ecommend it :)",

}
```

1. Ratings - Dataset

```
print(n_users, n_items)
```

```
(22363, 12101)
```

```
ratings['rating'].describe()
```

```
count    198502.000000
mean          4.190391
std           1.166580
min           1.000000
25%           4.000000
50%           5.000000
75%           5.000000
max           5.000000
Name: rating, dtype: float64
```

```
ratings.head()
```

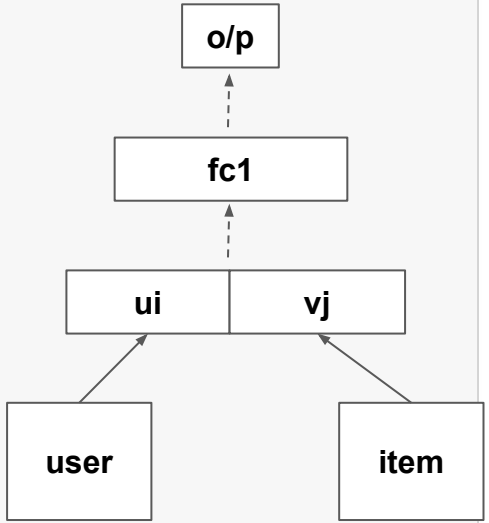
	user_id	item_id	rating	timestamp
13584	13069	818	5.0	01 1, 2008
24119	13069	1629	5.0	01 1, 2008
58956	21102	4024	5.0	01 1, 2009
1053	21055	57	4.0	01 1, 2009
67190	17404	4574	5.0	01 1, 2009

1. Ratings - Embedding

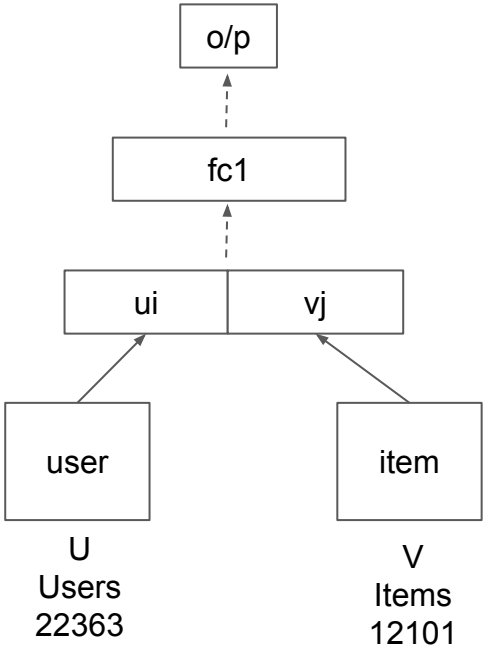
```
# input_dim: int > 0. Size of the vocabulary, i.e. maximum integer index + 1.
# output_dim: int >= 0. Dimension of the dense embedding.
# input_length: Length of input sequences, when it is constant.

user_id_input = Input(shape = [1], name = 'user')
item_id_input = Input(shape = [1], name = 'item')

embedding_size = 30
user_embedding = Embedding(output_dim = embedding_size, input_dim = n_users + 1,
                           input_length = 1, name = 'user_embedding')(user_id_input)
item_embedding = Embedding(output_dim = embedding_size, input_dim = n_items + 1,
                           input_length = 1, name = 'item_embedding')(item_id_input)

# reshape from shape: (samples, input_length, embedding_size)
# to shape: (samples, input_length * embedding_size) which is
# equal to shape: (samples, embedding_size)
user_vecs = Flatten()(user_embedding)
item_vecs = Flatten()(item_embedding)

# concatenate user_vecs and item_vecs
input_vecs = Concatenate()([user_vecs, item_vecs])
input_vecs = Dropout(0.5)(input_vecs)

# Include RELU as activation layer
x = Dense(64, activation='relu')(input_vecs)
y = Dense(1)(x)


model = Model(inputs=[user_id_input, item_id_input], outputs=y)
model.compile(optimizer='adam', loss='mae')
```

```
print(n_users, n_items)
```

```
(22363, 12101)
```



1. Ratings - Parameters




```
print(n_users, n_items)
```

```
(22363, 12101)
```

```
#Model Summary
model.summary()
```

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
user (InputLayer)            (None, 1)                 0
__________________________________________________________________________________________________
item (InputLayer)            (None, 1)                 0
__________________________________________________________________________________________________
user_embedding (Embedding)   (None, 1, 30)             670920      user[0][0]
__________________________________________________________________________________________________
item_embedding (Embedding)   (None, 1, 30)             363060      item[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)          (None, 30)                0           user_embedding[0][0]
__________________________________________________________________________________________________
flatten_2 (Flatten)          (None, 30)                0           item_embedding[0][0]
__________________________________________________________________________________________________
dot_1 (Dot)                  (None, 1)                 0           flatten_1[0][0]
                                                                   flatten_2[0][0]
==================================================================================================
Total params: 1,033,980
Trainable params: 1,033,980
Non-trainable params: 0
__________________________________________________________________________________________________
```

User Embedding:
22363 x 30 (670890)+30
= 670920

Item Embedding:
12101 x 30 (363030) +30
= 363060

1. Ratings - Train the model

```
# Training the model
history = model.fit([user_id_train, item_id_train], rating_train,
                    batch_size=64, epochs=5, validation_split=0.1,
                    shuffle=True)
```

```
Train on 142920 samples, validate on 15881 samples
Epoch 1/5
142920/142920 [==============================] - 13s - loss: 4.1770 - val_loss: 4.1171
Epoch 2/5
142920/142920 [==============================] - 13s - loss: 3.7710 - val_loss: 3.5313
Epoch 3/5
142920/142920 [==============================] - 13s - loss: 2.9960 - val_loss: 2.9219
Epoch 4/5
142920/142920 [==============================] - 15s - loss: 2.2455 - val_loss: 2.3839
Epoch 5/5
142920/142920 [==============================] - 14s - loss: 1.6374 - val_loss: 1.9744
```

```
# weights and shape
weights = model.get_weights()
[w.shape for w in weights]
```

```
[(22364, 30), (12102, 30)]
```

1. Ratings - Embeddings

```
user_embeddings = weights[0]
item_embeddings = weights[1]
print("First item name from metadata:", ratings["item_id"][1])
print("Embedding vector for the first item:")
print(item_embeddings[1])
print("shape:", item_embeddings[1].shape)
```

```
('First item name from metadata:', 0)
Embedding vector for the first item:
[-0.24441464  0.18797666 -0.34060067 -0.19173615 -0.2029438  -0.137593
  0.26437578 -0.23433381  0.35428327  0.34944484  0.29993126 -0.31245074
  0.12804551 -0.3820832   0.34728435  0.1300899  -0.10583965  0.31810758
  0.04773919  0.19603658 -0.3850561   0.05007624 -0.43756446  0.38865262
 -0.3326309  -0.42817637  0.07683372  0.33330366  0.46818402 -0.2553382 ]
('shape:', (30,))
```

```
model.save('rating_embedding.h5')
```

1. Rating: User embedding: 22363 x 30

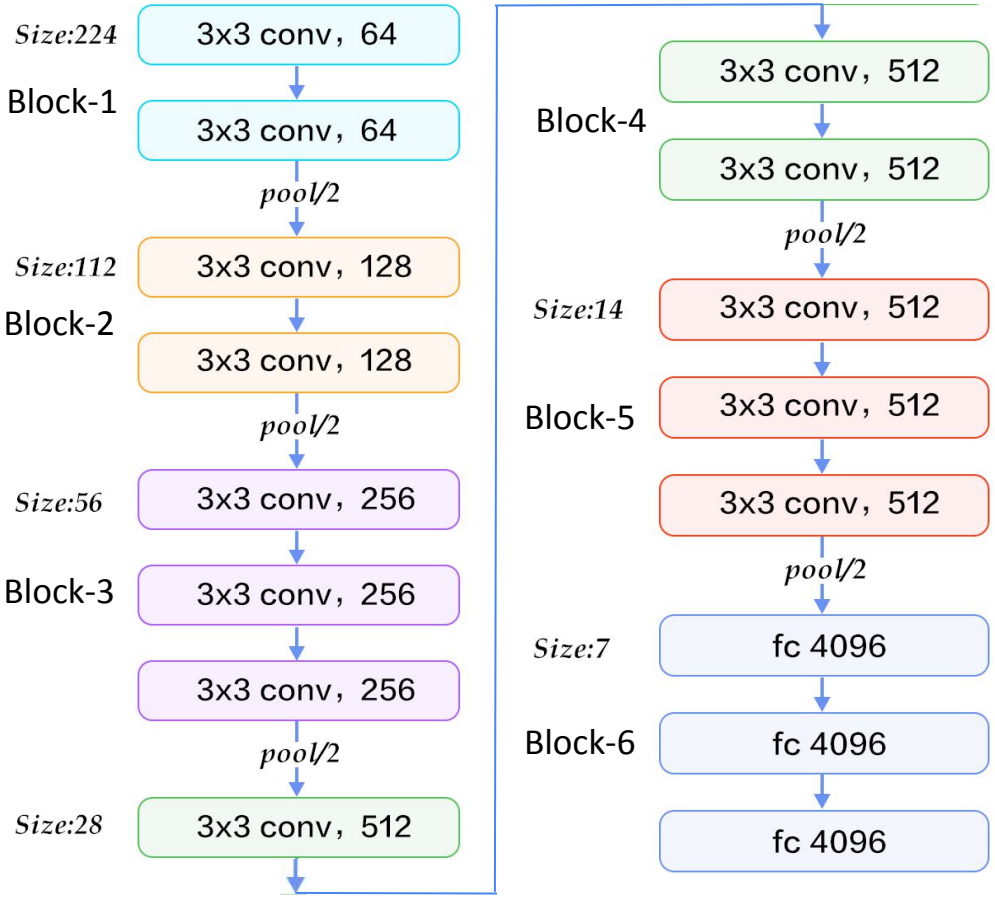
	22	23	24	25	26	27	28	29
22336	-0.10081...	0.322264...	-0.32614...	0.3609215	0.242819...	-0.11390...	0.4021436	-0.34924...
22337	-0.30850...	-0.28597...	0.401794...	0.173555...	0.3483822	-0.38651...	-0.20181...	-0.24300...
22338	-0.31386...	-0.26828...	-0.07133...	0.036313...	0.142945...	-0.03411...	0.021769...	-0.18000...
22339	-0.32681...	0.428842...	-0.13518...	-0.46035...	0.260106...	-0.26393...	0.434683...	-0.18212...
22340	-0.24629...	0.338364...	-0.42854...	0.055687...	0.018140...	0.101859...	0.219654...	-0.57400...
22341	-0.05166...	0.278361...	-0.03361...	0.053366...	0.2065327	-0.20420...	0.2359955	-0.253082
22342	-0.25565...	0.296106...	0.028338...	0.122890...	-0.07630...	-0.11203...	0.264135...	-0.14093...
22343	-0.35049...	0.5227021	-0.28026...	0.039121...	0.4385153	0.250231...	0.474850...	-0.42906...
22344	-0.29622...	0.345067...	-0.37515...	-0.45191...	0.090784...	0.3528427	0.309...	
22345	-0.12968...	0.045902...	-0.17071...	-0.27716...	0.334063...	0.099617...	0.14...	
22346	-0.34864...	0.186864...	-0.105133	-0.38506...	0.3180669	0.320272...	0.31...	
22347	-0.31514...	0.224782...	0.016189...	-0.24061...	0.099817...	-0.17598...	0.450...	
22348	-0.16820...	0.2323637	-0.11367...	-0.04629...	0.117115...	0.270221...	0.151325...	-0.12971...
22349	-0.38604...	0.143539...	-0.06238...	-0.17938...	0.115976...	0.061243...	0.009204...	-0.21799...
22350	0.004504...	0.037722...	-0.21071...	-0.13796...	-0.06046...	-0.13780...	0.355367	-0.19653...
22351	0.039510...	0.291751...	0.019376...	-0.13693...	0.179130...	0.3145283	0.150251...	-0.36480...
22352	-0.20759...	-0.08145...	-0.23695...	-0.13195...	0.2285506	0.022974...	0.154543...	-0.07553...
22353	-0.15800...	0.266126...	0.044948...	-0.23778...	-0.23225...	-0.02677...	0.196073...	0.0329111
22354	-0.32183...	0.016196...	-0.27543...	-0.29455...	-0.24303...	0.2064006	-0.25263...	0.136554...
22355	-0.287111	0.262544...	0.271309...	-0.03468...	0.336197...	0.3533659	0.3365365	0.2690742
22356	-0.30922...	0.2484099	0.200894...	-0.10129...	0.1933774	0.171147...	0.205975...	-0.01624...
22357	-0.06340...	-0.12140...	-0.25240...	0.239808...	0.165291...	-0.29023...	-0.24699...	-0.25034...
22358	-0.27323...	0.266289	-0.19525...	-0.29388...	0.156570...	0.2934468	0.166164...	-0.20104...
22359	-0.10823...	0.120870...	-0.17219...	-0.11601...	0.232262...	0.081642...	0.079854...	0.066588...
22360	-0.11411...	0.293592...	-0.23696...	-0.24245...	0.155442...	0.478006...	0.431586...	-0.31106...
22361	-0.01427...	0.2715699	-0.11238...	-0.14940...	0.358354...	0.125763...	0.402812...	-0.00672...
22362	-0.19791...	0.065775...	-0.26127...	-0.21181...	0.039901...	0.045982...	0.018361...	0.3062876
22363	-0.00157...	0.012827...	-0.01413...	-0.00345...	0.021519...	-0.00831...	0.013276...	-0.02291...

```
print(n_users, n_items)
```

```
(22363, 12101)
```

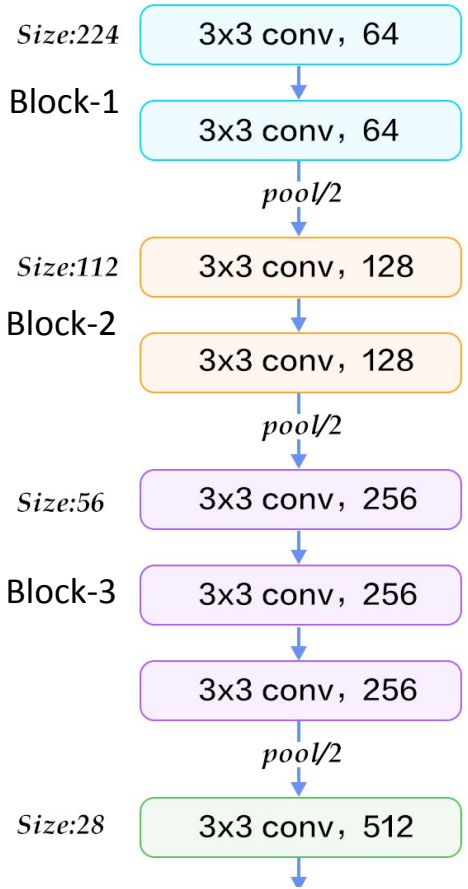
2. Images

2. VGG16 - Layers



Layer		
	INPUT:	[224x224x3]
1	CONV3-64:	[224x224x64]
2	CONV3-64:	[224x224x64]
3	CONV3-128:	[112x112x128]
4	CONV3-128:	[112x112x128]
	POOL2:	[56x56x128]
5	CONV3-256:	[56x56x256]
6	CONV3-256:	[56x56x256]
7	CONV3-256:	[56x56x256]
	POOL2:	[28x28x256]
8	CONV3-512:	[28x28x512]
9	CONV3-512:	[28x28x512]
10	CONV3-512:	[28x28x512]
	POOL2:	[14x14x512]
11	CONV3-512:	[14x14x512]
12	CONV3-512:	[14x14x512]
13	CONV3-512:	[14x14x512]
	POOL2:	[7x7x512]
14	FC:	[1x1x4096]
15	FC:	[1x1x4096]
16	FC:	[1x1x1000]

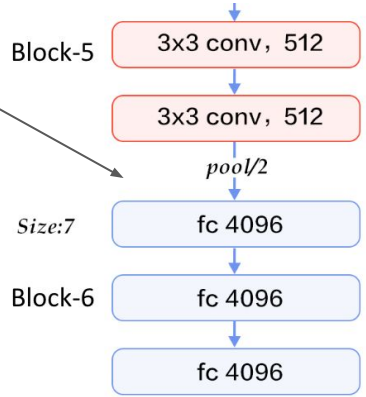
2. VGG16 - Defining Model




```
# The default input size for this model is 224x224
# keras.layers.Conv2D(filters, kernel_size, padding, strides, ...)
# filters: Integer, the dimensionality of the output space
#    (i.e. the number of output filters in the convolution).
# kernel_size: An integer or tuple/list of 2 integers,
#    specifying the height and width of the 2D convolution window.

# Block 1
x = layers.Conv2D(64, (3, 3),
                  activation='relu',
                  padding='same',
                  name='block1_conv1')(img_input)
x = layers.Conv2D(64, (3, 3),
                  activation='relu',
                  name='block1_conv2')(x)
x = layers.MaxPooling2D((2, 2), strides=(2, 2),
                  name='block1_pool')(x)

# Block 2
x = layers.Conv2D(128, (3, 3),
                  activation='relu',
                  name='block2_conv1')(x)
x = layers.Conv2D(128, (3, 3),
                  activation='relu',
                  name='block2_conv2')(x)
x = layers.MaxPooling2D((2, 2), strides=(2, 2),
                        name='block2_pool')(x)
```

2. VGG 16 - Extract weights from intermediate layer

```
# include_top: whether to include the
#    3 fully-connected layers at the top of the network.
# input_shape: optional shape tuple,
#    only to be specified if include_top is False
#    otherwise the input shape has to be  (224, 224, 3)

from keras.applications import VGG16
conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(224, 224, 3))
```




```
from keras import backend as K

# with a Sequential model
get_13th_layer_output = K.function([model.layers[0].input],
                                   [model.layers[13].output])
image_embedding_output = get_13th_layer_output([x])[0]

# Save weights
```

3. Review

```
# https://radimrehurek.com/gensim/models/word2vec.html
# model definition with parameters
# size (int, optional) – Dimensionality of the word vectors.
# window (int, optional) – Maximum distance between the current and predicted word within a sentence.
# sg ({0, 1}, optional) – Training algorithm: 1 for skip-gram; otherwise CBOW.
# hs ({0, 1}, optional) – If 1, hierarchical softmax will be used for model training.
#    If 0, and negative is non-zero, negative sampling will be used.
# negative (int, optional) – If > 0, negative sampling will be used,
#    the int for negative specifies how many “noise words” should be drawn
#    (usually between 5-20). If set to 0, no negative sampling is used.
# min_count (int, optional) – Ignores all words with total frequency lower than this.
# workers (int, optional) – Use these many worker threads to train the model
#    (=faster training with multicore machines).

from gensim.models import Word2Vec
def word2vec_model(train_data):
    model = Word2Vec(train_data,size=100,window = 3, min_count =1, sg =1, iter=20)
    return model
```

```
# train the model on train_data only and save model
train_data= train_df.groupby("userid")['item_id'].apply(list).values
wv_model=word2vec_model(train_data)
wv_model.save('word2vec_model')
```

Conclusion

Conclusion

- Covered Concept to Code for key methods and their use in Recommender Systems
  - Neural Network and MultiLayer Perceptron (NN & MLP)
  - Distributed representation aka Embedding (Skipgram)
  - Convolutional Neural Network (CNN)
  - Recurrent Neural Network (RNN)
- Help in exploring Recommender System with heterogeneous data

References

References (1/2)

[1] Jingyuan Chen, Hanwang Zhang, Xiangnan He, Liqiang Nie, Wei Liu, and Tat- Seng Chua. 2017. Attentive collaborative filtering: Multimedia recommendation with item-and component-level attention. In Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval. ACM, 335–344.
[2] Mihajlo Grbovic, Vladan Radosavljevic, Nemanja Djuric, Narayan Bhamidipati, Jaikit Savla, Varun Bhagwan, and Doug Sharp. 2015. E-commerce in your inbox: Product recommendations at scale. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 1809– 1818.
[3] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In proceedings of the 25th international conference on world wide web. International World Wide Web Conferences Steering Committee, 507–517.
[4] Ruining He and Julian McAuley. 2016. VBPR: Visual Bayesian Personalized Ranking from Implicit Feedback.. In AAAI. 144–150.
[5] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In Proceedings of the 26th International Conference on World Wide Web. International World Wide Web Conferences Steering Committee, 173–182.
[6] Alexandros Karatzoglou and Balázs Hidasi. 2017. Deep learning for recommender systems. In Proceedings of the Eleventh ACM Conference on Recommender Systems. ACM, 396–397.
[7] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In International Conference on Machine Learning. 1188–1196.

References (2/2)

[8] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. In Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, 43–52.
[9] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems. 3111–3119.
[10] Thomas Nedelec, Elena Smirnova, and Flavian Vasile. 2016. Content2vec: Specializing joint representations of product images and text for the task of product recommendation. (2016).
[11] Xin Rong. 2014. word2vec parameter learning explained. arXiv preprint arXiv:1411.2738 (2014).
[12] Devashish Shankar, Sujay Narumanchi, HA Ananya, Pramod Kompalli, and Krishnendu Chaudhury. 2017. Deep learning based large scale visual recommendation and search for e-commerce. arXiv preprint arXiv:1703.02344 (2017).
[13] Flavian Vasile, Elena Smirnova, and Alexis Conneau. 2016. Meta-prod2vec: Product embeddings using side-information for recommendation. In Proceedings of the 10th ACM Conference on Recommender Systems. ACM, 225–232.
[14] Chao-YuanWu, Amr Ahmed, Alex Beutel, Alexander J Smola, and How Jing. 2017. Recurrent recommender networks. In Proceedings of the tenth ACM international conference on web search and data mining. ACM, 495–503.
[15] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In European conference on computer vision. Springer, 818–833.

Acknowledgement

We gratefully acknowledge the help of

1. Srilakshmi Madiraju, IIT Kharagpur, India
2. Gourab Chowdhury, IIT Kharagpur, India

for preparing some of the codes.

Thank You.

Contact:
omsonie@gmail.com
omprakash.s@flipkart.com

The code will be available on:
www.DeepThinking.ai
www.c2cRecSys.com