word2vec

Yan Song

Outline

- Neural Representation Model for Words
- word2vec
  - Concept
  - Basics
  - Details with Implementation
- Evaluation
  - Word similarity measurement

Neural Representation Model for Words

- Words are the most fundamental units in language
  - Basic semantics
- Can be used in a flexible way in many tasks
- Easy to learn

- Of course there are more studies on sub-words ;-)

Neural Representation Model for Words

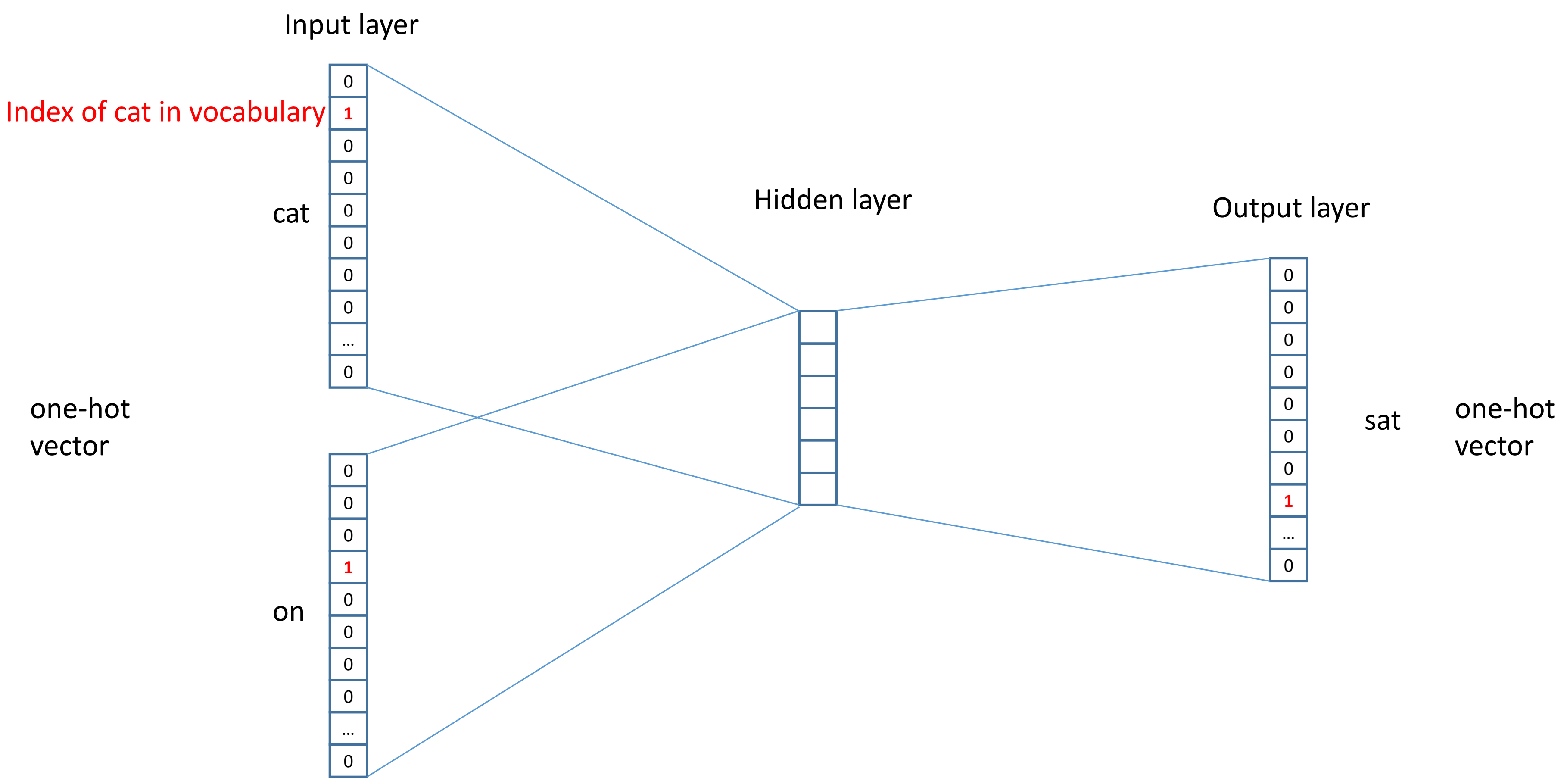
- What information can be leveraged?
  - Context!
- N-gram (word association) language modeling is the most effecient way for building word-word relations
- Two ways:
  - Top-down (GloVe)
  - Bottom-up (word2vec)

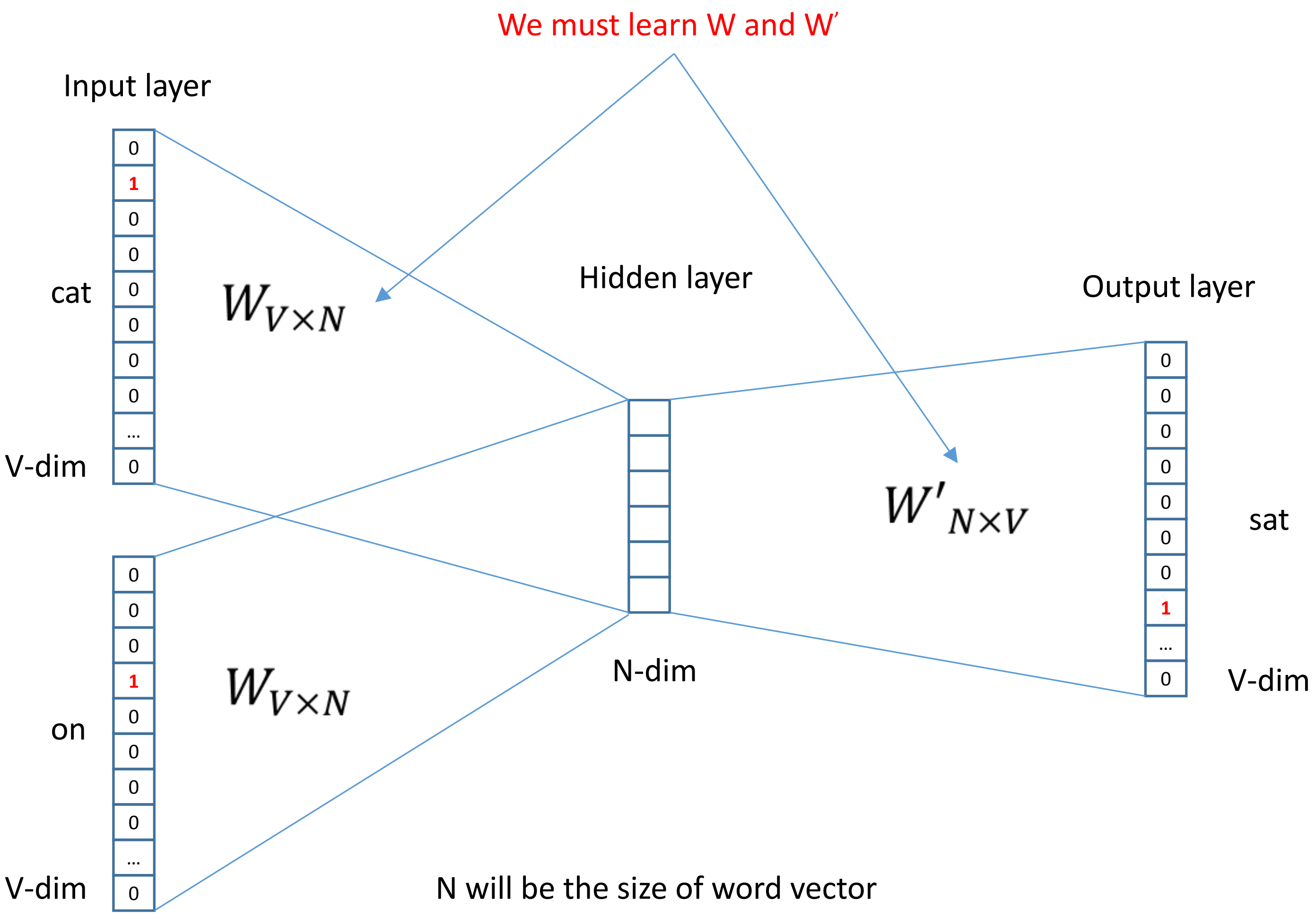
word2vec - Concept

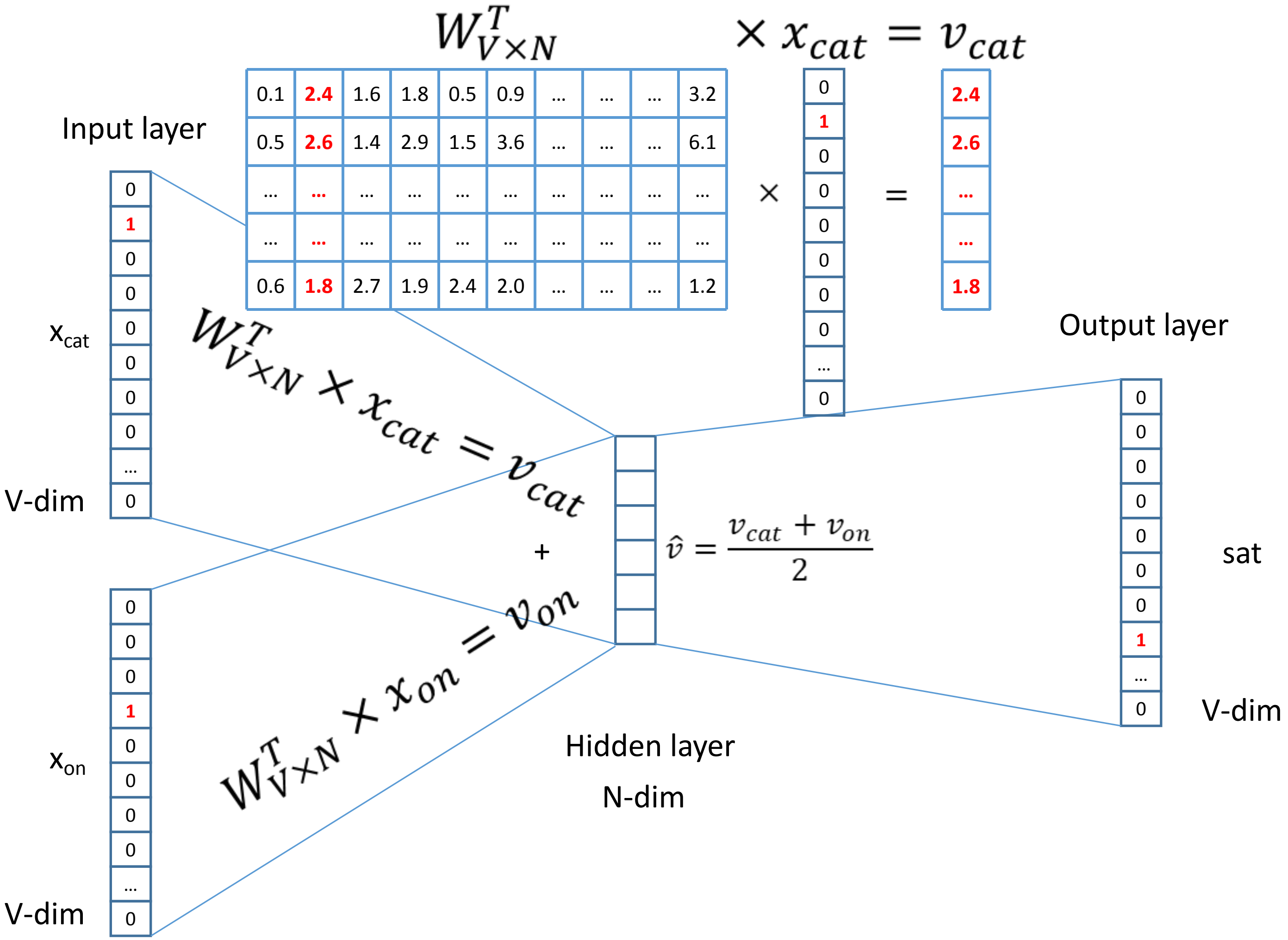
- Key idea: Predict surrounding words of every word
- Faster and can easily incorporate a new sentence/document or add a word to the vocabulary
- An indirect matrix decomposition way of learning dense representations (compared with GloVe)

- Let's start from a typical process of neural language modeling

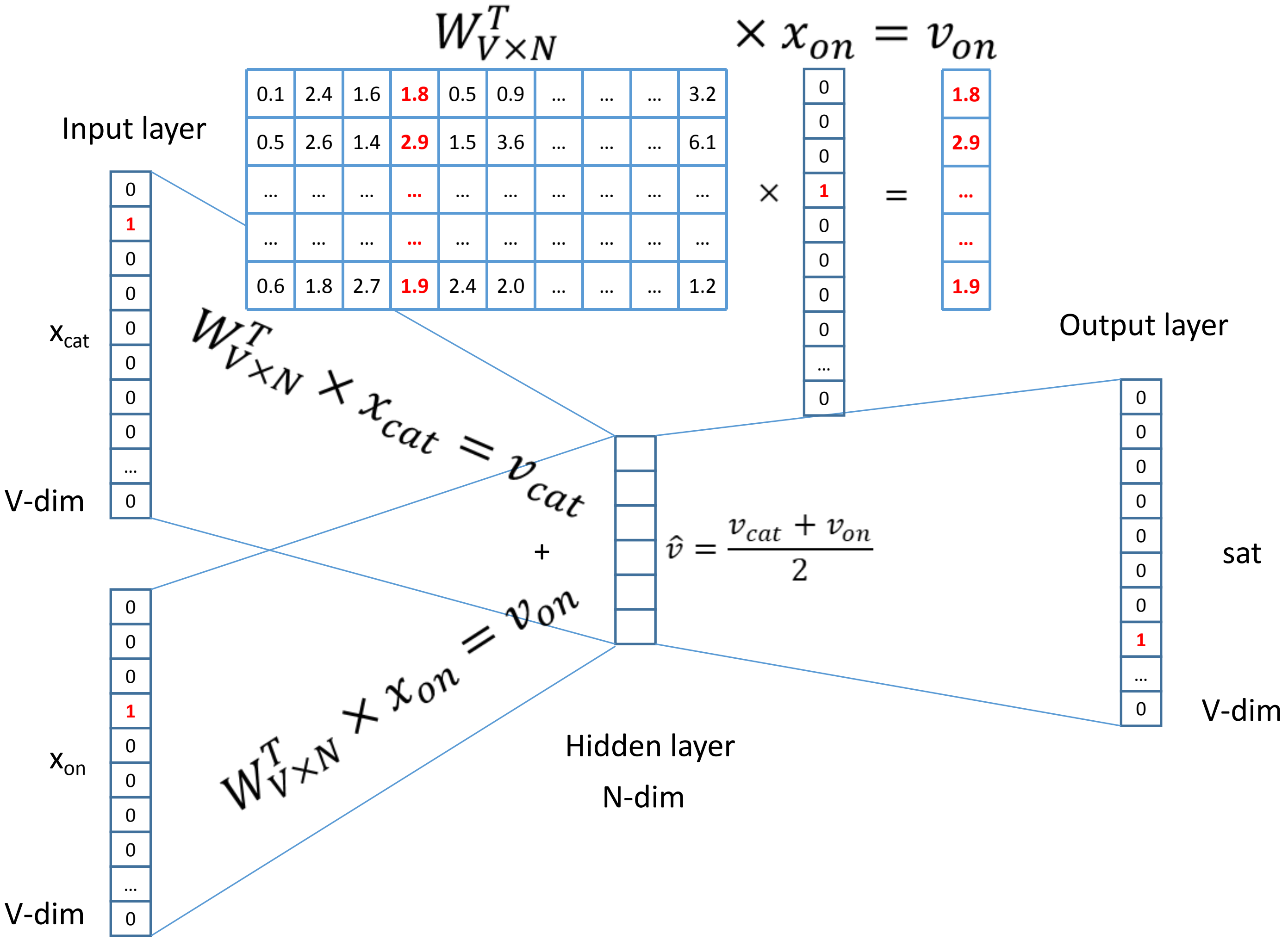
Illustration



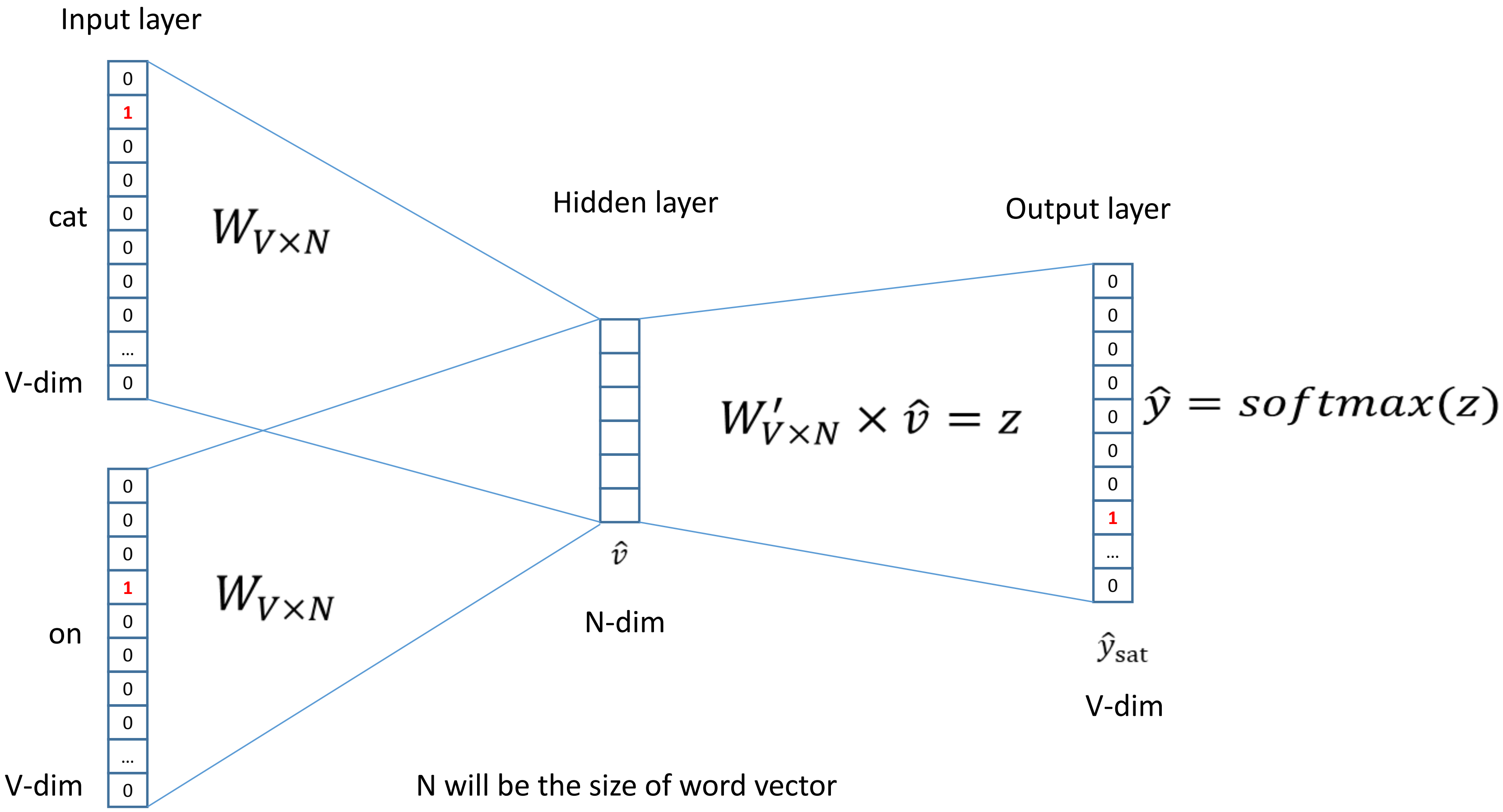

We must learn W and W′
Input layer
cat
V-dim
on
V-dim
$W_{V\times N}$
$W_{V\times N}$
Hidden layer
N-dim
Output layer
sat
V-dim
$W'_{N\times V}$
0
1
0
0
0
0
0
0
…
0
0
0
0
1
0
0
0
0
…
0
0
0
0
0
0
0
0
1
…
0
N will be the size of word vector



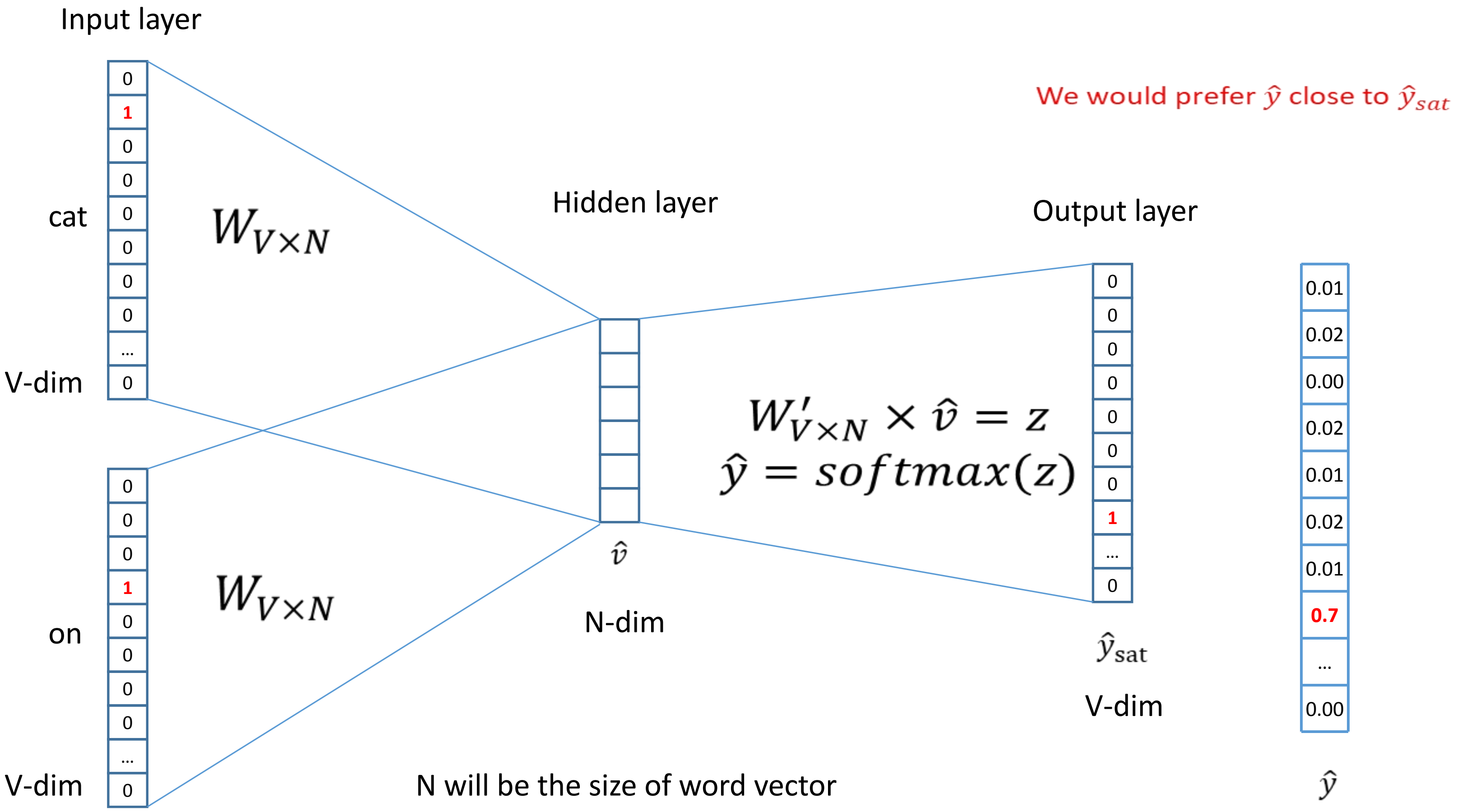
$W_{V\times N}^T \times x_{cat} = v_{cat}$
Input layer
Output layer
Hidden layer
N-dim
x_{cat}
x_{on}
V-dim
V-dim
V-dim
sat
$W_{V\times N}^T \times x_{cat} = v_{cat}$
$W_{V\times N}^T \times x_{on} = v_{on}$
+
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
0.1 2.4 1.6 1.8 0.5 0.9 … … … 3.2
0.5 2.6 1.4 2.9 1.5 3.6 … … … 6.1
0.6 1.8 2.7 1.9 2.4 2.0 … … … 1.2
2.4
2.6
1.8



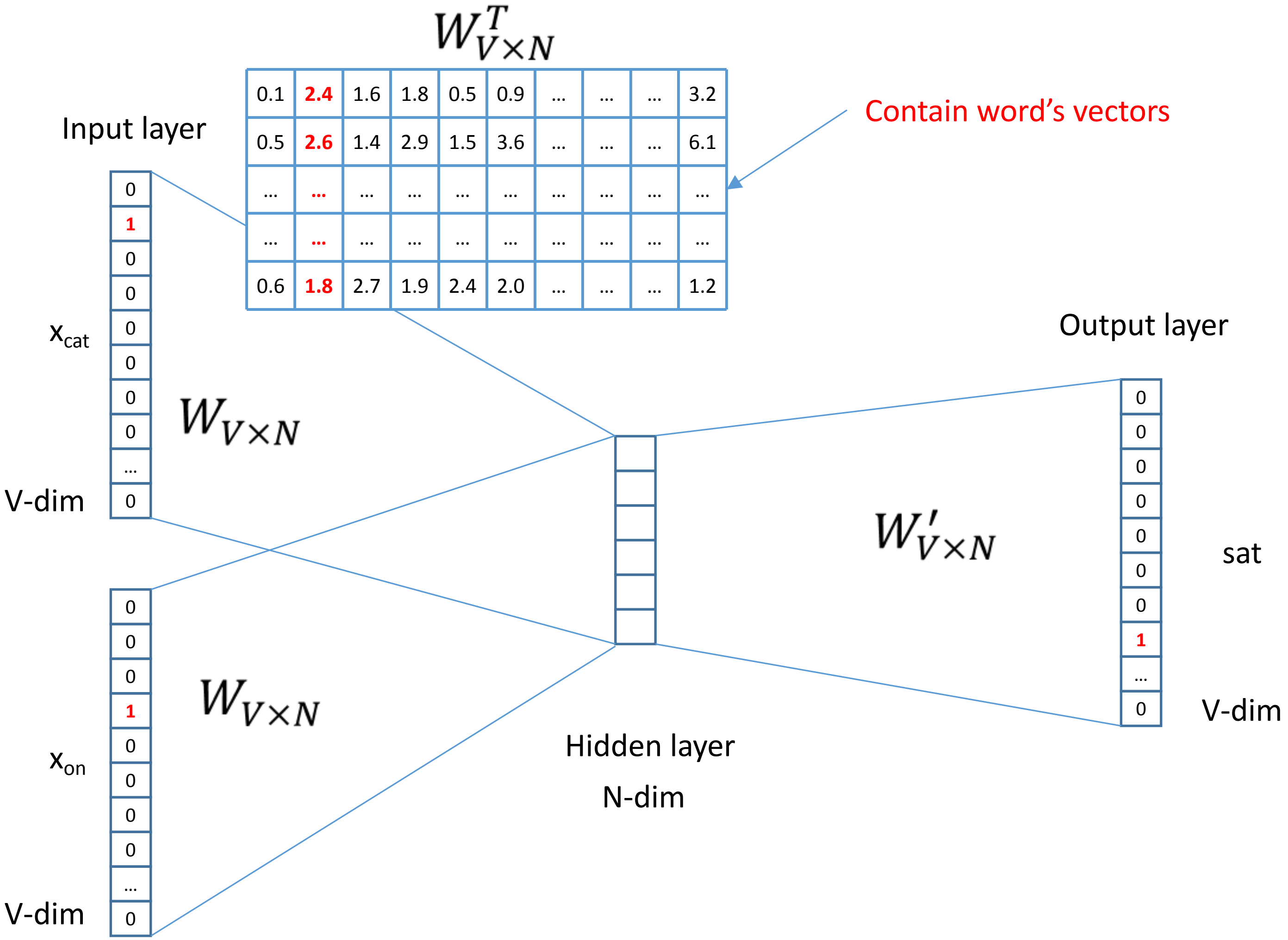
$W_{V\times N}^{T}$
$\times x_{on} = v_{on}$
Input layer
0.1 2.4 1.6 1.8 0.5 0.9 … … … 3.2
0.5 2.6 1.4 2.9 1.5 3.6 … … … 6.1
0.6 1.8 2.7 1.9 2.4 2.0 … … … 1.2
×
=
1.8
2.9
1.9
Output layer
x_{cat}
$W_{V\times N}^{T} \times x_{cat} = v_{cat}$
V-dim
+
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
$W_{V\times N}^{T} \times x_{on} = v_{on}$
sat
V-dim
x_{on}
Hidden layer
N-dim
V-dim



Input layer
cat
V-dim
on
V-dim
$W_{V\times N}$
$W_{V\times N}$
Hidden layer
$\hat{v}$
N-dim
Output layer
$W'_{V\times N} \times \hat{v} = z$
$\hat{y} = softmax(z)$
$\hat{y}_{\text{sat}}$
V-dim
N will be the size of word vector



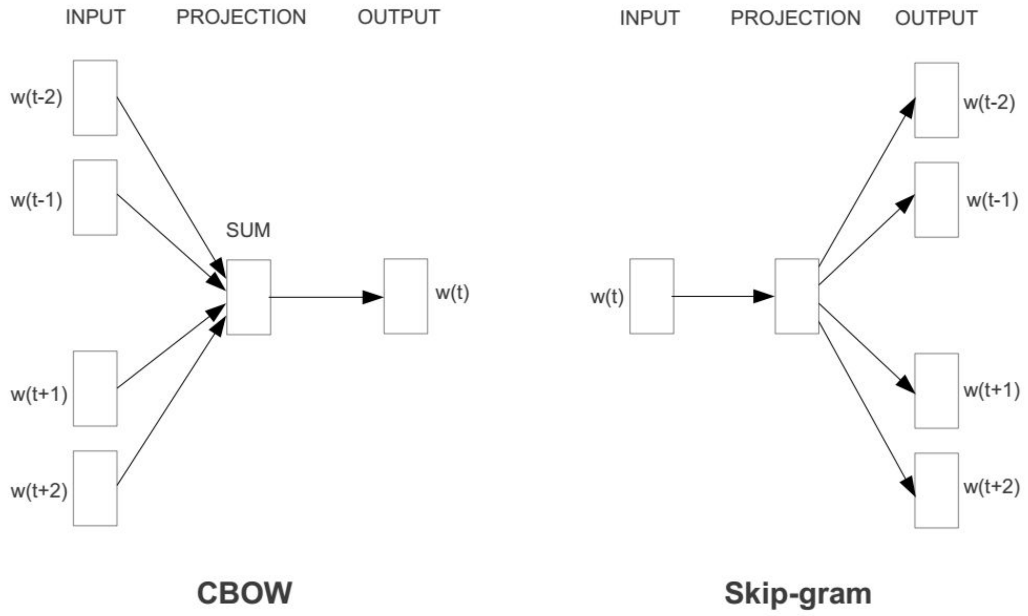
Input layer
We would prefer $\hat{y}$ close to $\hat{y}_{sat}$
cat
$W_{V\times N}$
Hidden layer
Output layer
V-dim
$W'_{V\times N} \times \hat{v} = z$
$\hat{y} = softmax(z)$
$\hat{v}$
N-dim
on
$W_{V\times N}$
$\hat{y}_{sat}$
V-dim
V-dim
N will be the size of word vector
$\hat{y}$
0.01
0.02
0.00
0.02
0.01
0.02
0.01
0.7
...
0.00



We can consider either W or W' as the word's representation. Or even take the average.

word2vec - Basics

- Two basic structures:
  - Continuous Bag of Word (CBOW): use a window of word to predict the middle word
  - Skip-gram (SG): use a word to predict the surrounding ones in window.



word2vec - Basics

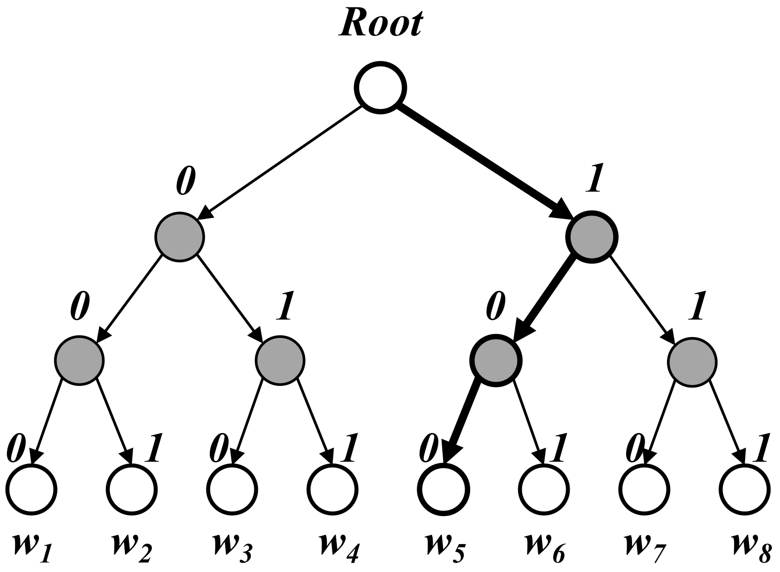
- Challenges of Implementation
  - Sparse output layer
  - Matrix (vector) computation over a huge corpus
  - Deal with words with different frequencies
  - Effecient gradient updating
  - ...

word2vec - Basics

- Two prediction strategies
  - Hierarchical Softmax
    - Several pipelined softmaxes go through a tree structure
    - Effective to capture words' information when they have similar patterns in frequencies
  - Negative sampling
    - A binary classification to distinguish if a word is in context or not
    - Effective when training data is huge (less softmax operations)

Hierarchical Softmax

- An illustration
- Softmax is done at each node along the path from root to leaves
- Sibling nodes may share more frequency patterns
- Effectively convert a high-dimensional classification problem into a series of small ones



word2vec - Implementation

- Workflow
  - Load data
  - Prepare data structure
  - Training (CBOW or SG)
    - if with HS
      - predict go through hierarchical path
      - update
    - else: with NS
      - randomly choose negative samples
      - predict and update
  - Save model

word2vec - Implementation

- Load data
  - The goal is to create a vocabulary
  - Related Functions
    - AddWordToVocab
    - GetWordHash
    - LearnVocabFromTrainFile
    - ReadVocab
    - ReadWord
    - ReadWordIndex
    - ReduceVocab
    - SearchVocab
    - SortVocab

word2vec - Implementation

- Some tricks
  - All embedddings are stored in a very long 1-D array.

```
a = posix_memalign((void **)&syn0, 128, (long long)vocab_size * layer1_size * sizeof(real));
if (syn0 == NULL) {printf("Memory allocation failed\n"); exit(1);}
```

  - Sigmoid function is precomputed

```
expTable = (real *)malloc((EXP_TABLE_SIZE + 1) * sizeof(real));
for (i = 0; i < EXP_TABLE_SIZE; i++) {
  expTable[i] = exp((i / (real)EXP_TABLE_SIZE * 2 - 1) * MAX_EXP); // Precompute the exp() table
  expTable[i] = expTable[i] / (expTable[i] + 1);                   // Precompute f(x) = x / (x + 1)
}
```

- important variables: syn0, syn1, neu1, neu1e

word2vec - Implementation

- Prepare data structure

```c
// Create binary Huffman tree using the word counts
// Frequent words will have short uniqe binary codes
void CreateBinaryTree() {
  long long a, b, i, min1i, min2i, pos1, pos2, point[MAX_CODE_LENGTH];
  char code[MAX_CODE_LENGTH];
  long long *count = (long long *)calloc(vocab_size * 2 + 1, sizeof(long long));
  long long *binary = (long long *)calloc(vocab_size * 2 + 1, sizeof(long long));
  long long *parent_node = (long long *)calloc(vocab_size * 2 + 1, sizeof(long long));
  for (a = 0; a < vocab_size; a++) count[a] = vocab[a].cn;
  for (a = vocab_size; a < vocab_size * 2; a++) count[a] = 1e15;
  pos1 = vocab_size - 1;
  pos2 = vocab_size;
  // Following algorithm constructs the Huffman tree by adding one node at a time
  for (a = 0; a < vocab_size - 1; a++) {
    // First, find two smallest nodes 'min1, min2'
    if (pos1 >= 0) {
      if (count[pos1] < count[pos2]) {
        min1i = pos1;
        pos1--;
      } else {
        min1i = pos2;
        pos2++;
      }
    } else {
      min1i = pos2;
      pos2++;
    }
    if (pos1 >= 0) {
      if (count[pos1] < count[pos2]) {
        min2i = pos1;
        pos1--;
      } else {
        min2i = pos2;
        pos2++;
      }
    } else {
      min2i = pos2;
      pos2++;
    }
    count[vocab_size + a] = count[min1i] + count[min2i];
    parent_node[min1i] = vocab_size + a;
    parent_node[min2i] = vocab_size + a;
    binary[min2i] = 1;
  }
  // Now assign binary code to each vocabulary word
  for (a = 0; a < vocab_size; a++) {
    b = a;
    i = 0;
    while (1) {
      code[i] = binary[b];
      point[i] = b;
      i++;
      b = parent_node[b];
      if (b == vocab_size * 2 - 2) break;
    }
    vocab[a].codelen = i;
    vocab[a].point[0] = vocab_size - 2;
    for (b = 0; b < i; b++) {
      vocab[a].code[i - b - 1] = code[b];
      vocab[a].point[i - b] = point[b] - vocab_size;
    }
  }
  free(count);
  free(binary);
  free(parent_node);
}
```

word2vec - Implementation

- CBOW

```
    if (cbow) {  //train the cbow architecture
      // in -> hidden
      cw = 0;
      for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
        c = sentence_position - window + a;
        if (c < 0) continue;
        if (c >= sentence_length) continue;
        last_word = sen[c];
        if (last_word == -1) continue;
        for (c = 0; c < layer1_size; c++) neu1[c] += syn0[c + last_word * layer1_size];
        cw++;
      }
      if (cw) {
        for (c = 0; c < layer1_size; c++) neu1[c] /= cw;
        if (hs) for (d = 0; d < vocab[word].codelen; d++) {
          f = 0;
          l2 = vocab[word].point[d] * layer1_size;
          // Propagate hidden -> output
          for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1[c + l2];
          if (f <= -MAX_EXP) continue;
          else if (f >= MAX_EXP) continue;
          else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
          // 'g' is the gradient multiplied by the learning rate
          g = (1 - vocab[word].code[d] - f) * alpha;
          // Propagate errors output -> hidden
          for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
          // Learn weights hidden -> output
          for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * neu1[c];
        }
        // NEGATIVE SAMPLING
        if (negative > 0) for (d = 0; d < negative + 1; d++) {
          if (d == 0) {
            target = word;
```

Projection

```
            label = 1;
          } else {
            next_random = next_random * (unsigned long long)25214903917 + 11;
            target = table[(next_random >> 16) % table_size];
            if (target == 0) target = next_random % (vocab_size - 1) + 1;
            if (target == word) continue;
            label = 0;
          }
          l2 = target * layer1_size;
          f = 0;
          for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1neg[c + l2];
          if (f > MAX_EXP) g = (label - 1) * alpha;
          else if (f < -MAX_EXP) g = (label - 0) * alpha;
          else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
          for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
          for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * neu1[c];
        }
        // hidden -> in
        for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
          c = sentence_position - window + a;
          if (c < 0) continue;
          if (c >= sentence_length) continue;
          last_word = sen[c];
          if (last_word == -1) continue;
          for (c = 0; c < layer1_size; c++) syn0[c + last_word * layer1_size] += neu1e[c];
        }
      }
```

word2vec - Implementation

- SG

Projection

```
} else {  //train skip-gram
  for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
    c = sentence_position - window + a;
    if (c < 0) continue;
    if (c >= sentence_length) continue;
    last_word = sen[c];
    if (last_word == -1) continue;
    l1 = last_word * layer1_size;
    for (c = 0; c < layer1_size; c++) neu1e[c] = 0;
    // HIERARCHICAL SOFTMAX
    if (hs) for (d = 0; d < vocab[word].codelen; d++) {
      f = 0;
      l2 = vocab[word].point[d] * layer1_size;
      // Propagate hidden -> output
      for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1[c + l2];
      if (f <= -MAX_EXP) continue;
      else if (f >= MAX_EXP) continue;
      else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
      // 'g' is the gradient multiplied by the learning rate
      g = (1 - vocab[word].code[d] - f) * alpha;
      // Propagate errors output -> hidden
      for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
      // Learn weights hidden -> output
      for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * syn0[c + l1];
    }
    // NEGATIVE SAMPLING
    if (negative > 0) for (d = 0; d < negative + 1; d++) {
      if (d == 0) {
        target = word;
        label = 1;
      } else {
        next_random = next_random * (unsigned long long)25214903917 + 11;
        target = table[(next_random >> 16) % table_size];
        if (target == 0) target = next_random % (vocab_size - 1) + 1;
        if (target == word) continue;
        label = 0;
      }
      l2 = target * layer1_size;
      f = 0;
      for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1neg[c + l2];
      if (f > MAX_EXP) g = (label - 1) * alpha;
      else if (f < -MAX_EXP) g = (label - 0) * alpha;
      else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
      for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
      for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * syn0[c + l1];
    }
    // Learn weights input -> hidden
    for (c = 0; c < layer1_size; c++) syn0[c + l1] += neu1e[c];
  }
}
sentence_position++;
if (sentence_position >= sentence_length) {
  sentence_length = 0;
  continue;
}
}
```

word2vec - Implementation

- A closer look of HS

```
// HIERARCHICAL SOFTMAX
if (hs) for (d = 0; d < vocab[word].codelen; d++) {
  f = 0;
  l2 = vocab[word].point[d] * layer1_size;
  // Propagate hidden -> output
  for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1[c + l2];
  if (f <= -MAX_EXP) continue;
  else if (f >= MAX_EXP) continue;
  else f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
  // 'g' is the gradient multiplied by the learning rate
  g = (1 - vocab[word].code[d] - f) * alpha;
  // Propagate errors output -> hidden
  for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
  // Learn weights hidden -> output
  for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * syn0[c + l1];
}
```

Go through the path

word2vec - Implementation

- A closer look of NS

```c
// NEGATIVE SAMPLING
if (negative > 0) for (d = 0; d < negative + 1; d++) {
  if (d == 0) {
    target = word;
    label = 1;
  } else {
    next_random = next_random * (unsigned long long)25214903917 + 11;
    target = table[(next_random >> 16) % table_size];
    if (target == 0) target = next_random % (vocab_size - 1) + 1;
    if (target == word) continue;
    label = 0;
  }
  l2 = target * layer1_size;
  f = 0;
  for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1neg[c + l2];
  if (f > MAX_EXP) g = (label - 1) * alpha;
  else if (f < -MAX_EXP) g = (label - 0) * alpha;
  else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
  for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
  for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * syn0[c + l1];
}
```

Positive

Negative

word2vec - Implementation

- Update the input embeddings
  - CBOW

```
// hidden -> in
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
  c = sentence_position - window + a;
  if (c < 0) continue;
  if (c >= sentence_length) continue;
  last_word = sen[c];
  if (last_word == -1) continue;
  for (c = 0; c < layer1_size; c++) syn0[c + last_word * layer1_size] += neu1e[c];
}
```

  - SG

```
// Learn weights input -> hidden
for (c = 0; c < layer1_size; c++) syn0[c + l1] += neu1e[c];
```

Evaluation

- We already know cosine similarity is the main metric for intrinsic evaluation
- But, how to sysmatically evaluate?
  - Prepare a human annotated similarity/relatedness dataset
    - A list of word pairs
  - Compute the similarities of all pairs using the resulted embeddings
  - Compute the correlation between human annotations and the similarities

Evaluation

- Spearman's correlation

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}.$$

	x	y
tiger cat	7.35	0.517977544
tiger tiger	10.00	1.0
plane car	5.77	0.217732013636
train car	6.31	0.197087634791
television radio	6.77	0.257970738766
media radio	7.42	0.212014745796
bread butter	6.19	0.7017263618
cucumber potato	5.92	0.396596853365
doctor nurse	7.00	0.382799131727
professor doctor	6.62	0.288006966845
student professor	6.81	0.19973114775
smart stupid	5.81	0.228546918905
wood forest	7.73	0.229380996826

Hw4

- Train your word2vec embedding using the Text8 corpus
  - with HS (CBOW, SG)
  - with NS (CBOW, SG)
- Evaluate using spearman's correlation
- Submit a result.txt to me with
  - The results from five embeddings (4 word2vec and 1 GloVe)
  - Your observations on HS v.s. NS on CBOW and SG