Advanced Artificial Intelligence

Lab 11

Outline

- Review
  - Regularization
    - Ridge regression
    - Lasso regression
- Exercise
  - Automatic feature engineering method ---- Word2vec

Regularization

- Basic idea:
  - The more features matter in the model, the bigger complexity.
  - Regularization = introducing penalty for complexity and reducing #features
- Interpretation:
  - It biases the model toward lower complexity (fewer features).
  - Application of Occam's razor: the model should be simple (fewer coefficients).
  - Bayesian viewpoint: regularization = imposing prior knowledge that the world is simple on the learning model.

Review: Multivariate Linear Regression

- Given: data $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, and output $\boldsymbol{y} \in \mathbb{R}^{N \times 1}$.
  - N: #samples, D: #features
- Aim: find $\boldsymbol{\theta} \in \mathbb{R}^{D \times 1}$ to minimize $\frac{1}{2}\left|\left|\boldsymbol{X\theta} - \boldsymbol{y}\right|\right|_2^2$
- Solution: $\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$

Examples of Regularization Methods

- Ridge regression
- Lasso regression
- Smoothing splines
- *Support vector machines*
- *Regularized neural networks*
- *……*

Ridge regression

- Linear regression
  - A linear model: $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$
  - Cost function: minimizes the **sum of squared residuals (SSR)**

$$J = \frac{1}{n}\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)^2$$
$$= \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{w}^\top \boldsymbol{x}_i + b - y_i)^2$$

  - the **sum of squared residuals (SSR)** with **Ridge regression**

$$J = \frac{1}{n}\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)^2 + \lambda \|w\|_2^2$$

Lasso regression

- Linear regression
  - A linear model: $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$
  - Cost function: minimizes the **sum of squared residuals (SSR)**

$$J = \frac{1}{n}\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)^2$$

$$= \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{w}^\top \boldsymbol{x}_i + b - y_i)^2$$

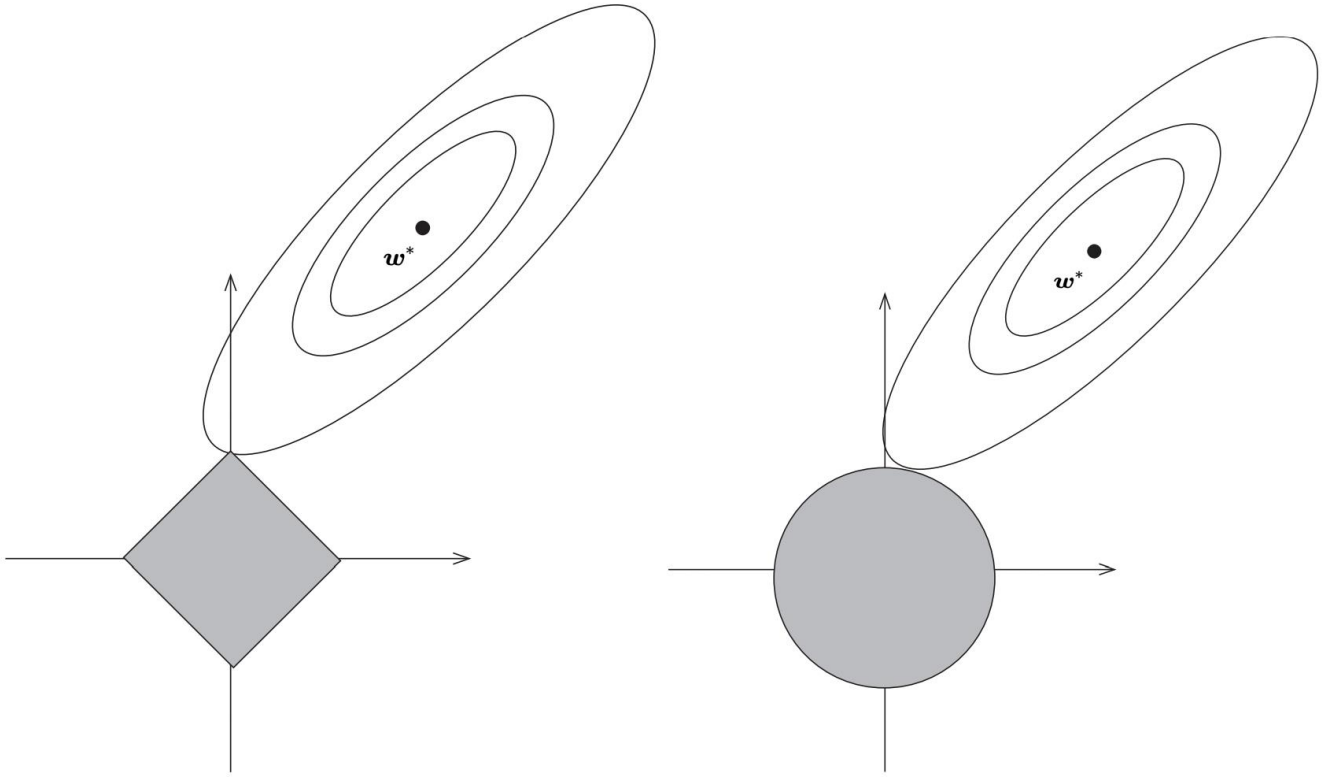
  - the **sum of squared residuals (SSR)** with **Lasso regression**

$$J = \frac{1}{n}\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)^2 + \lambda\|w\|_1$$

Comparison (Ridge regression vs Lasso regression)

- Similarity
  - Can be used to solve the overfitting problem of standard linear regression.
- Difference
  - Lasso can be used for feature selection, but ridge regression cannot. In other words, Lasso is easier to make the weight become 0, while ridge is easier to make the weight close to 0.
  - From the perspective of Bayes, Lasso is equivalent to the *Laplace distribution as a* prior probability distribution of parameter W, while ridge regression is equivalent to the *Gaussian distribution* as the prior distribution of parameter W.

Comparison (Ridge regression vs Lasso regression)



Lasso (left) and ridge (right) regression.

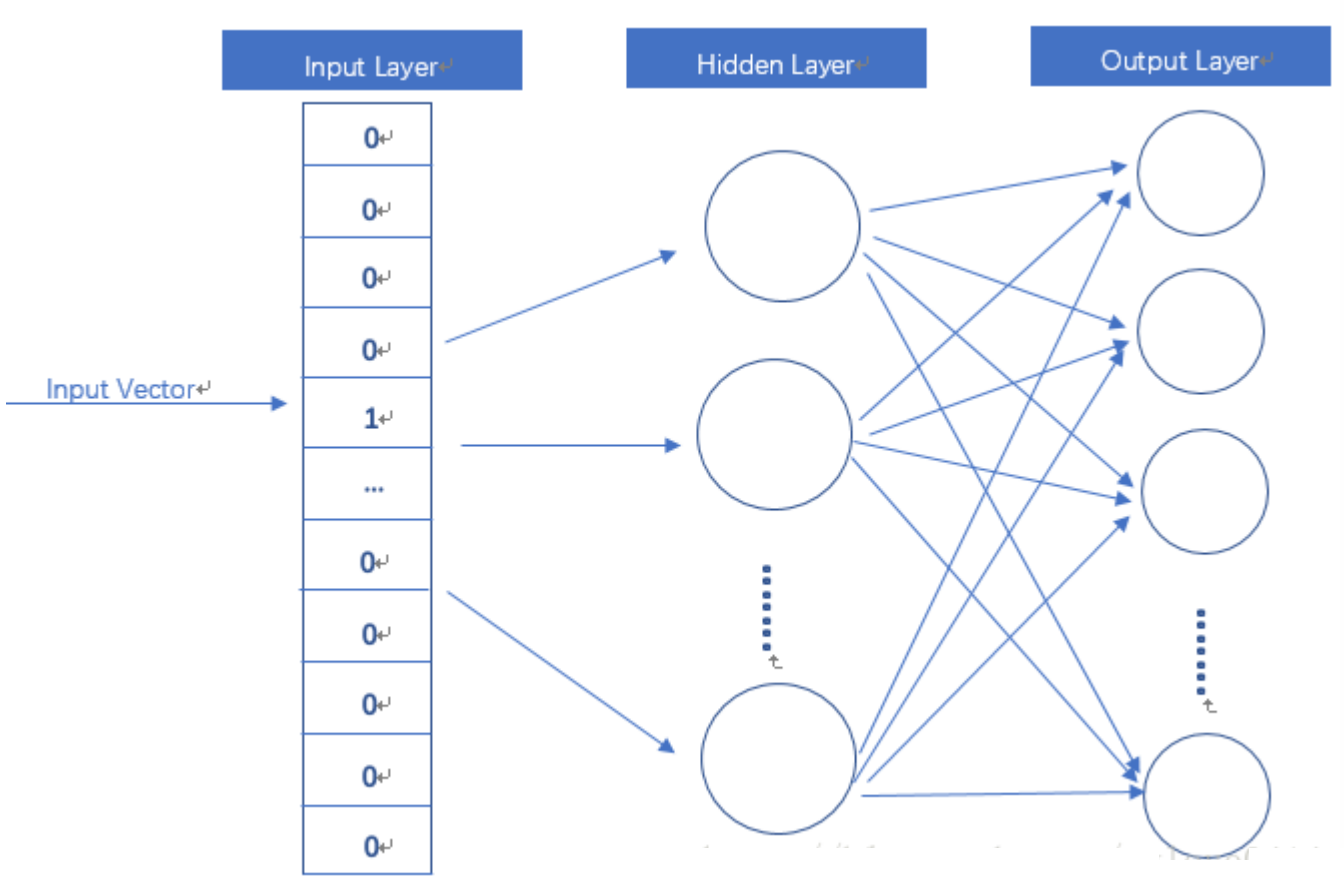
Automatic feature engineering method **Word2vec**

One-hot:

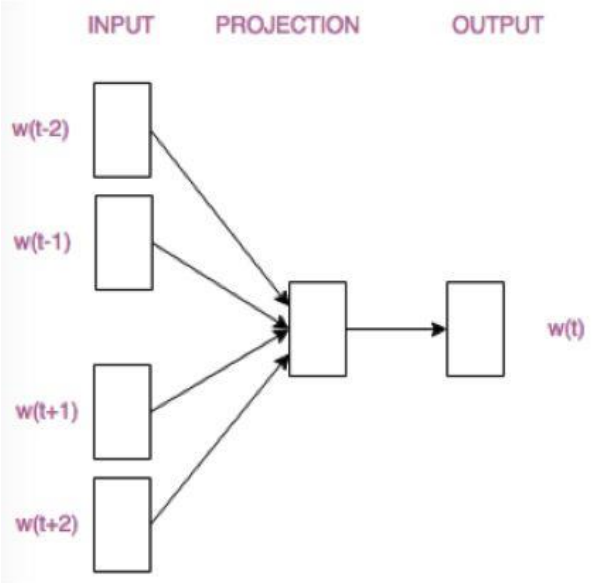
深圳 [0,0,0,0,0,0,0,1,0,……, 0,0,0,0,0,0,0]
上海 [0,0,0,0,1,0,0,0,0,……, 0,0,0,0,0,0,0]
杭州 [0,0,0,1,0,0,0,0,0,……, 0,0,0,0,0,0,0]
北京 [0,0,0,0,0,0,0,0,0,……, 1,0,0,0,0,0,0]

word embedding

Word2vec

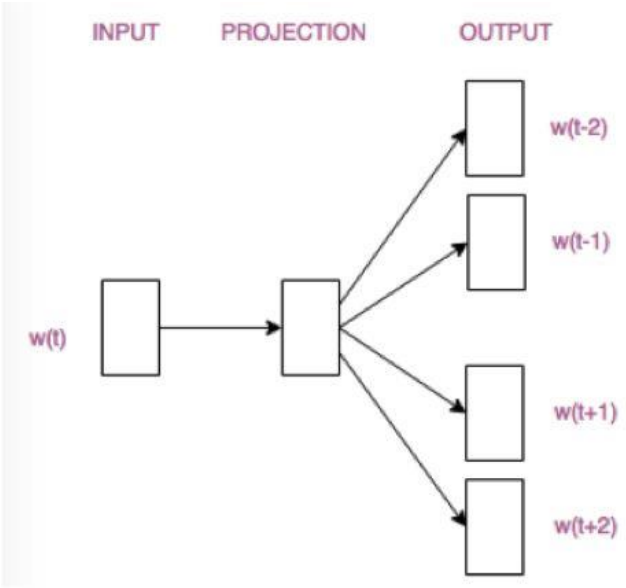

Input Layer
Hidden Layer
Output Layer
Input Vector
0
0
0
0
1
...
0
0
0
0
0
0

Word2vec



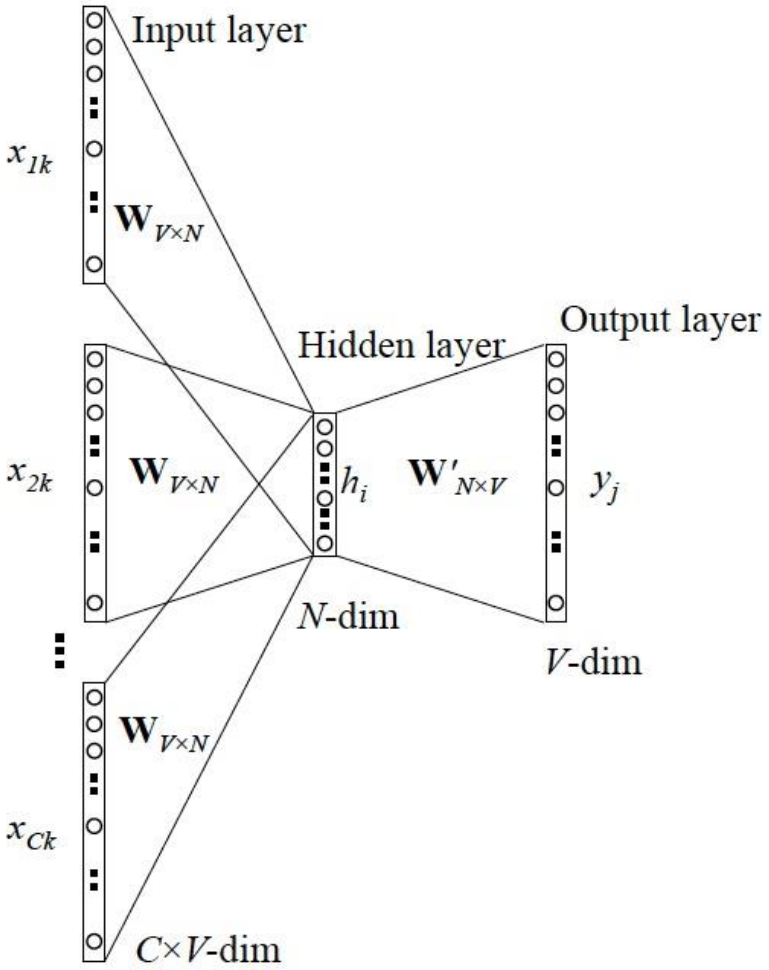
CBOW

Continuous Bag-of-Words



Skip-Gram

CBOW



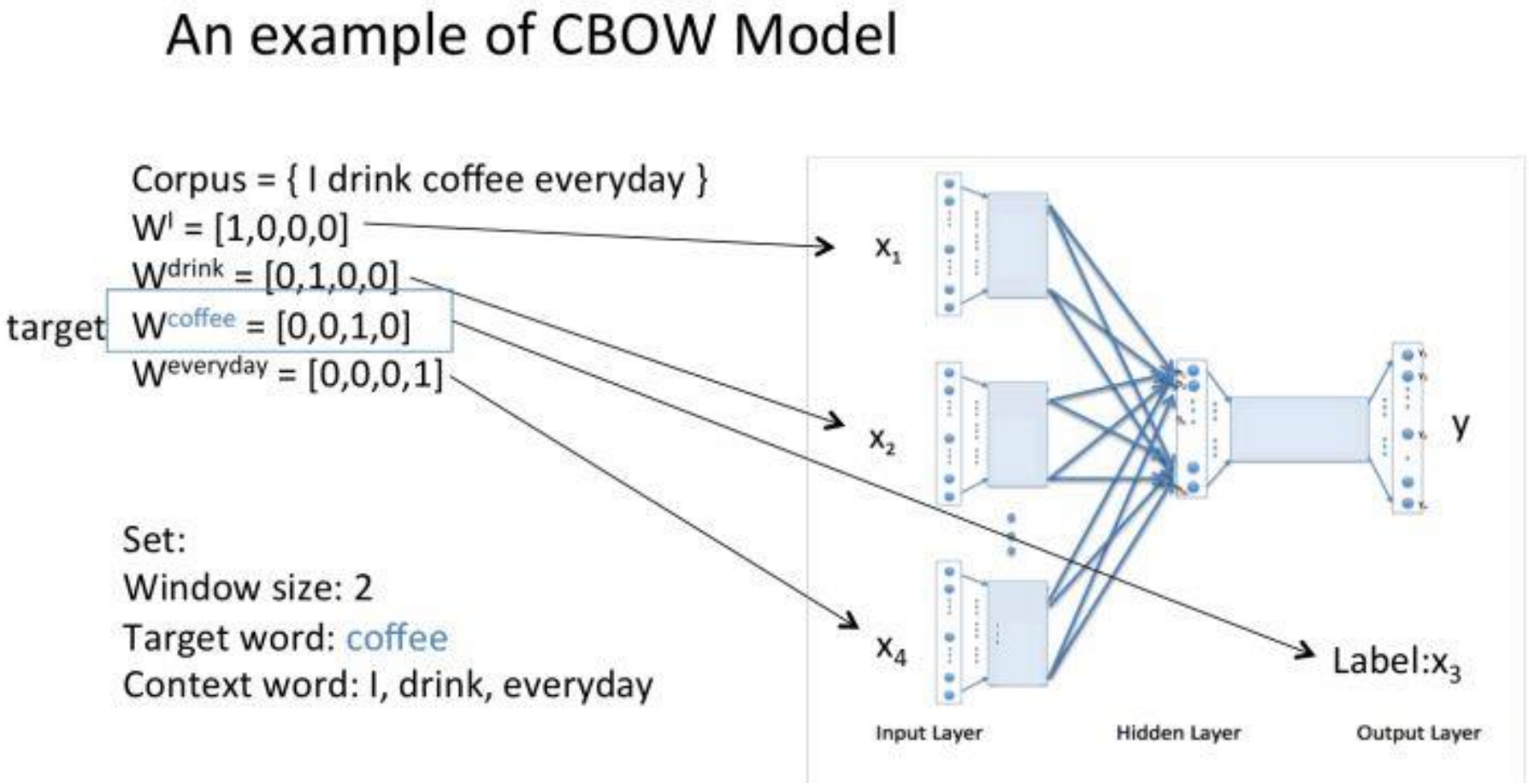
Input: one-hot (V-dim); C words

Hidden layer: mean(XW)->H(N-dim)

Output layer: HW’ -> Y(V-dim)

Loss function: cross entropy

CBOW


An example of CBOW Model
Corpus = { I drink coffee everyday }
W^I = [1,0,0,0]
W^drink = [0,1,0,0]
target
W^coffee = [0,0,1,0]
W^everyday = [0,0,0,1]
Set:
Window size: 2
Target word: coffee
Context word: I, drink, everyday
x1
x2
x4
y
Label:x3
Input Layer
Hidden Layer
Output Layer

CBOW

An example of CBOW Model

Output: Probability distribution

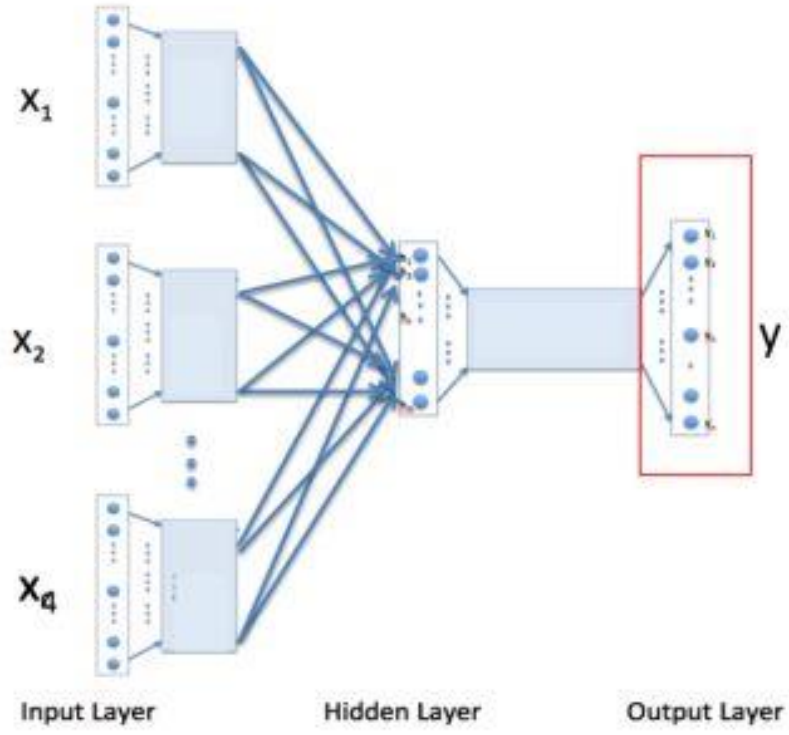
$$softmax(\mathbf{u}_o) = \mathbf{y}$$

$$softmax\left(\begin{bmatrix} 4.01 \\ 2.01 \\ 5.00 \\ 3.34 \end{bmatrix}\right) = \begin{bmatrix} 0.23 \\ 0.03 \\ 0.62 \\ 0.12 \end{bmatrix}$$

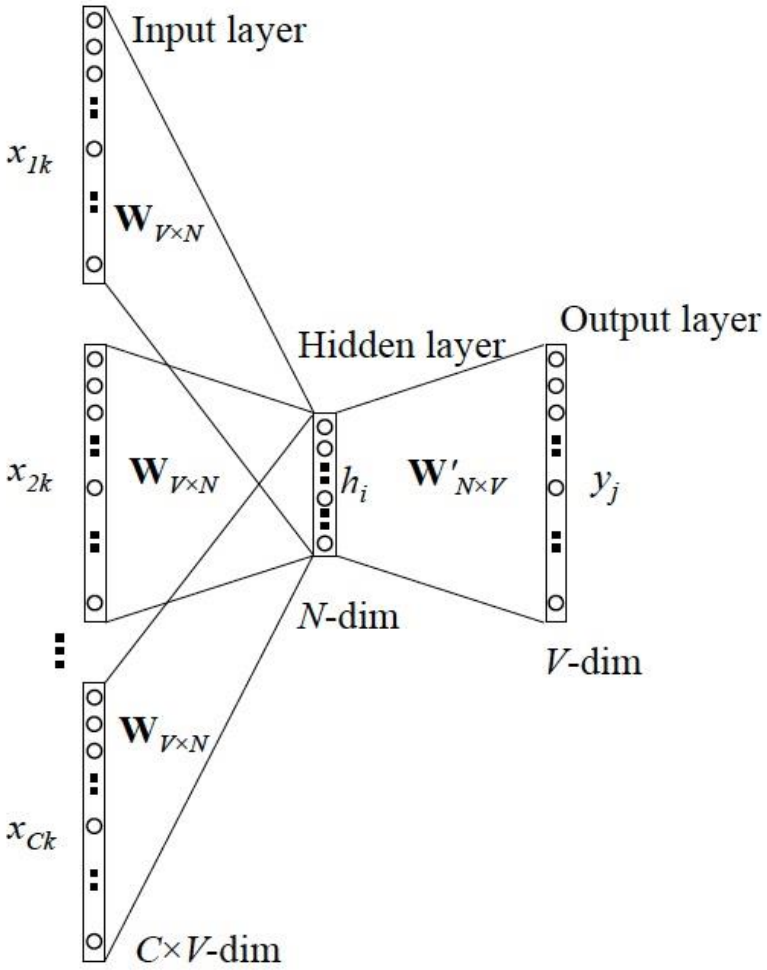
Probability of “coffee”

We desire probability generated to match the true probability(label) x_3 [0,0,1,0]

Use gradient descent to update W and W’



CBOW



W: look up table

word embedding: xW

Exercise

We are about to study the idea of a computational process. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called data. The evolution of a process is directed by a pattern of rules called a program. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

Exercise

Step 1: data pre-processed

```
CONTEXT_SIZE = 2  # 左右各2个单词
EMBEDDING_DIM = 10
raw_text = "We are about to study the idea of a computational process. Computational processes are
 abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things
 called data. The evolution of a process is directed by a pattern of rules called a program. People
 create programs to direct processes. In effect, we conjure the spirits of the computer with our
 spells.".split()

# 通过从 `raw_text` 得到一组单词，进行去重操作
vocab = set(raw_text)
vocab_size = len(vocab)

word_to_ix = {word: i for i, word in enumerate(vocab)}
data = []
for i in range(2, len(raw_text) - 2):
    context = [raw_text[i - 2], raw_text[i - 1],
               raw_text[i + 1], raw_text[i + 2]]
    target = raw_text[i]
    data.append((context, target))
print(data[0])
print(data[1])
print(data[2])
```

```
(['We', 'are', 'to', 'study'], 'about')
(['are', 'about', 'study', 'the'], 'to')
(['about', 'to', 'the', 'idea'], 'study')
```

Exercise

Step 2: modeling

```
class CBOW(nn.Module):

    def __init__(self, vocab_size, embedding_dim, context_size):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(2*context_size * embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((1, -1))
        out = F.relu(self.linear1(embeds))
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs

    def get_embedding(self, inputs):
        return self.embeddings(inputs)

def make_context_vector(context, word_to_ix):
    idxs = [word_to_ix[w] for w in context]
    return torch.tensor(idxs, dtype=torch.long)

losses = []
loss_function = nn.NLLLoss()
model = CBOW(len(vocab), EMBEDDING_DIM, CONTEXT_SIZE)
optimizer = optim.SGD(model.parameters(), lr=0.001)
print(model)
```

```
CBOW(
  (embeddings): Embedding(49, 10)
  (linear1): Linear(in_features=40, out_features=128, bias=True)
  (linear2): Linear(in_features=128, out_features=49, bias=True)
)
```

Exercise

Step 3: optimize

```
for epoch in range(100):
    total_loss = 0
    for context, target in data:
        # 步骤 1. 准备好进入模型的数据 (例如将单词转换成整数索引,并将其封装在变量中)
        context_idxs = make_context_vector(context, word_to_ix)

        # 步骤 2. 回调 *积累* 梯度. 在进入一个实例前,需要将之前的实力梯度置零
        model.zero_grad()

        # 步骤 3. 运行反向传播,得到单词的概率分布
        log_probs = model(context_idxs)

        # 步骤 4. 计算损失函数. (再次注意, Torch需要将目标单词封装在变量中)
        loss = loss_function(log_probs, torch.tensor([word_to_ix[target]], dtype=torch.long))

        # 步骤 5. 反向传播并更新梯度
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
    losses.append(total_loss)
```

Exercise

Step 4: test

```
test = "we"
print(model.get_embedding(torch.tensor(word_to_ix[test])))
```

```
tensor([ 0.7363, -0.1777,  0.0787, -0.4340, -0.3591, -1.2715,  1.9409, -1.8806,
        -0.0158,  0.2593], grad_fn=<EmbeddingBackward>)
```