MACHINE LEARNING

机器学习

Neural Networks

嵌入层与预训练技术

参考：
《神经网络与深度学习》

Machine Learning Course

嵌入层与词嵌入学习

- 定义

图像、声音之类的数据很自然地可以表示为一个连续的向量。相比，语言常被看作是一个字或词的序列，每个字或词都看成是离散的符号。而在计算机内部，每个字都是表示为无意义的编码。

嵌入层负责作将这种离散的符号进行转换，使其更适合输入到神经网络中。

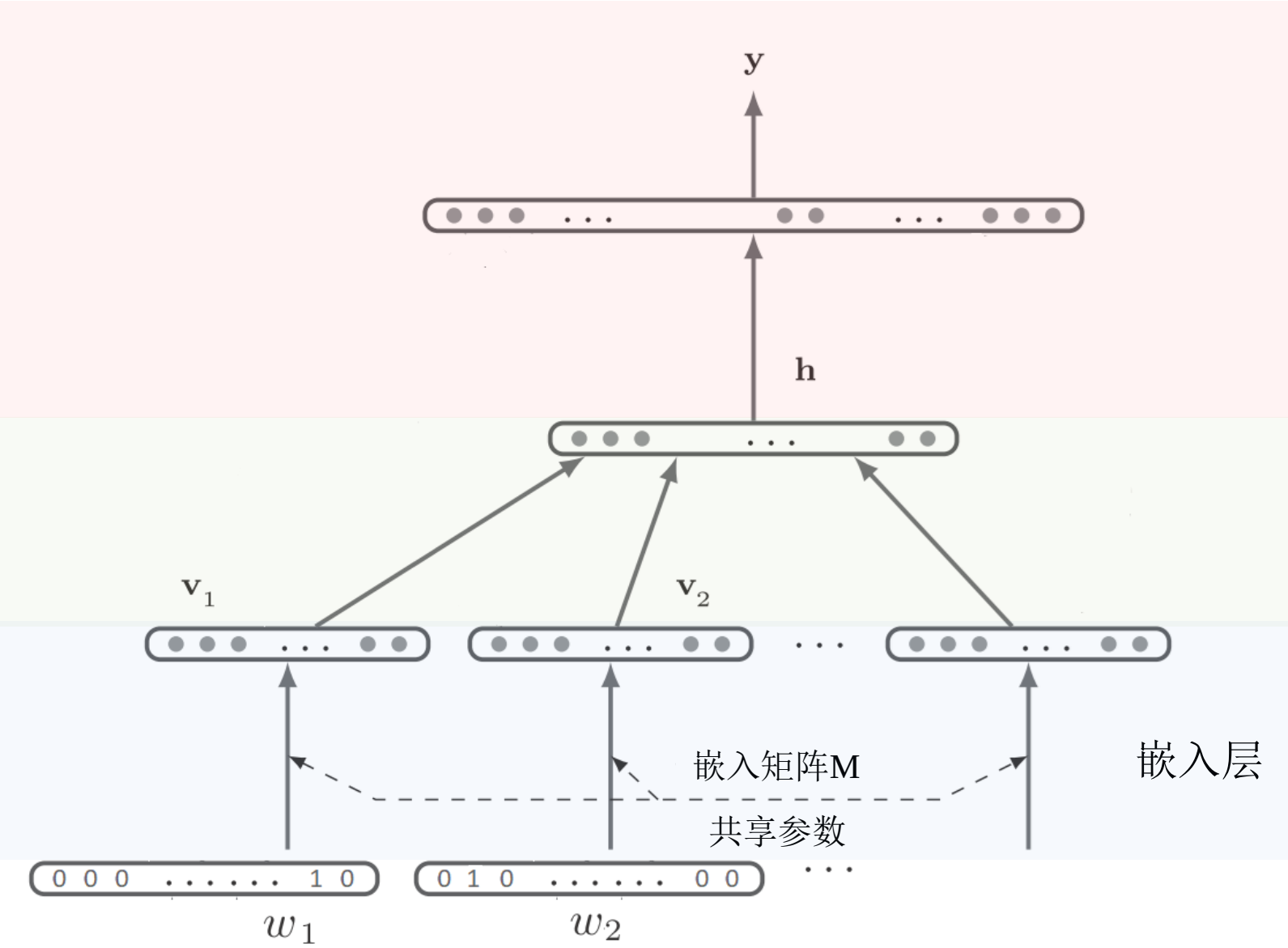
我们通常有两种方式来表示词：

- 一种方式是用one-hot向量的形式特定的词，这就相当于很多输入神经元中仅有一个神经元代表一个词。所以把这种表示方式叫做局部表示*Local Representation*，也称为离散表示或符号表示。
- 另一种方式是用一组神经元来表示一个特定的词，我们把这种表示方式叫做分布式表示 *Distributed Representation*。分布式表示通常对应了低维的稠密向量。所以输入的神经元的个数将大幅度减少。

颜色	局部表示	分布式表示
琥珀色	$[1, 0, 0, 0]^T$	$[1.00, 0.75, 0.00]^T$
天蓝色	$[0, 1, 0, 0]^T$	$[0.00, 0.5, 1.00]^T$
中国红	$[0, 0, 1, 0]^T$	$[0.67, 0.22, 0.12]^T$
咖啡色	$[0, 0, 0, 1]^T$	$[0.44, 0.31\ 0.22]^T$

嵌入层与词嵌入学习

- 嵌入层



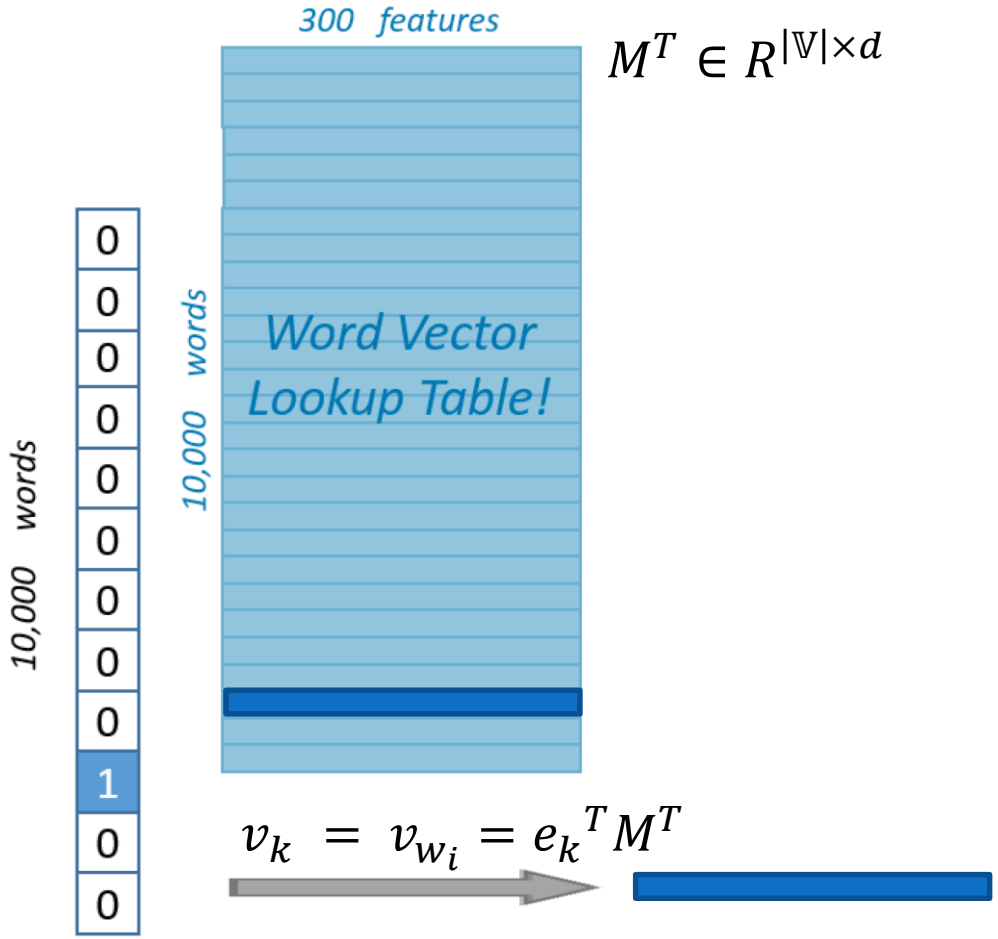
嵌入层与词嵌入学习

- 嵌入层

- 嵌入矩阵/查询表

$e_k \in R^{|\mathbb{V}|\times 1}$：词$w_i$对应的one-hot表达

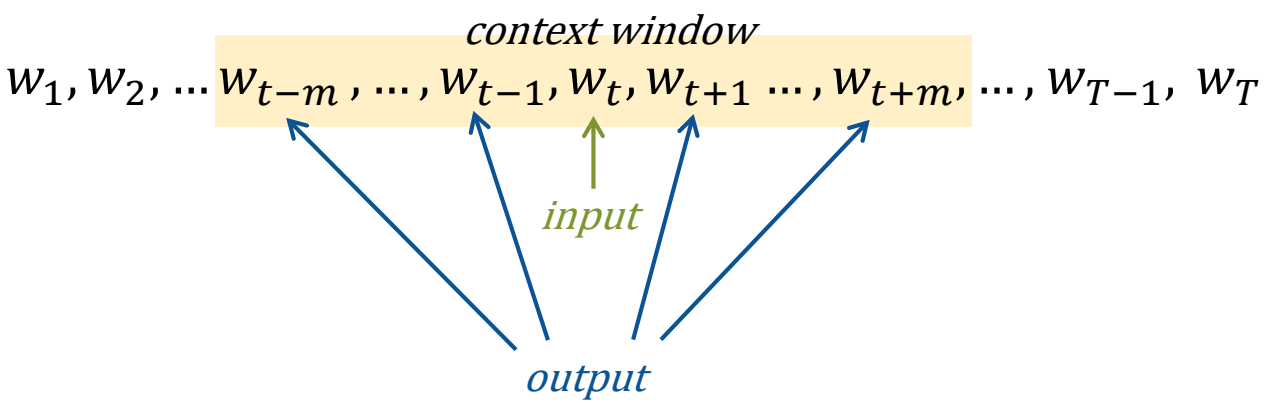
$M \in R^{d\times|\mathbb{V}|}$：嵌入矩阵/查询矩阵/嵌入表，将每个符号直接映射成向量表示



嵌入层与词嵌入学习

- 词嵌入学习

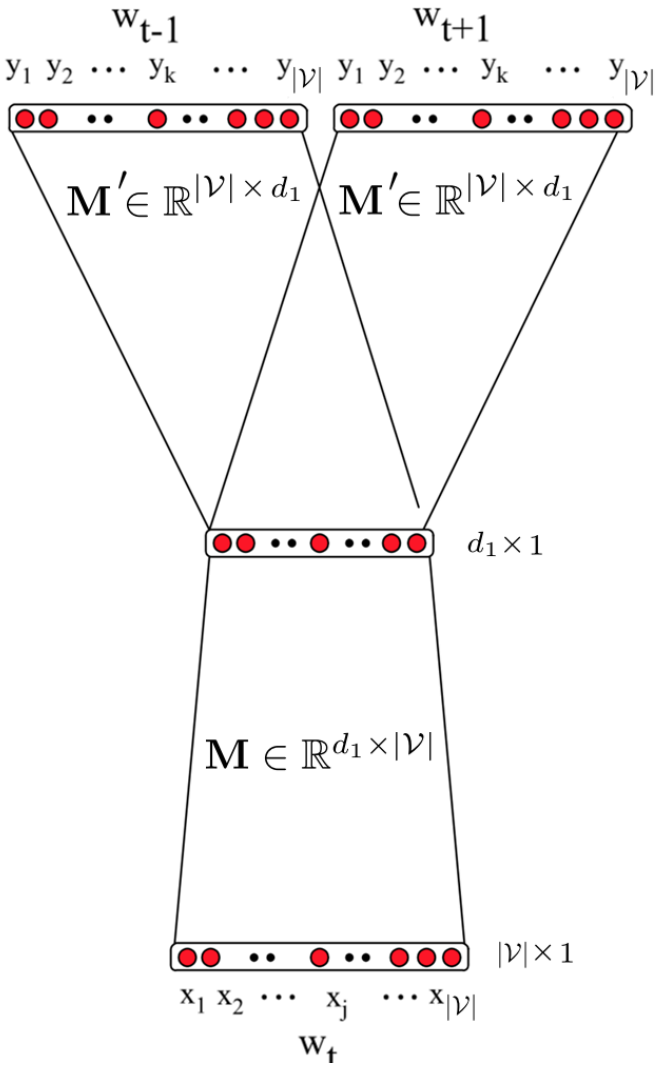
- Word2Vec技术：Skip-Gram



$$P(w_{t+j} \mid w_t) = \text{softmax}\left(\mathbf{v}_{w_t}^{\mathrm{T}} \mathbf{v}'_{w_{t+j}}\right) = \frac{\exp\left(\mathbf{v}_{w_t}^{\mathrm{T}} \mathbf{v}'_{w_{t+j}}\right)}{\sum_{w' \in \mathcal{V}} \exp\left(\mathbf{v}_{w_t}^{\mathrm{T}} \mathbf{v}'_{w'}\right)}$$

$$\mathcal{L}_\theta = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} \log P(w_{t+j} \mid w_t)$$

若两个词的$\boldsymbol{v}_{w_t}$相似，则同样的词作为它们的上下文的概率是相似的。这两个词就对应了相似的上下文



嵌入层与词嵌入学习

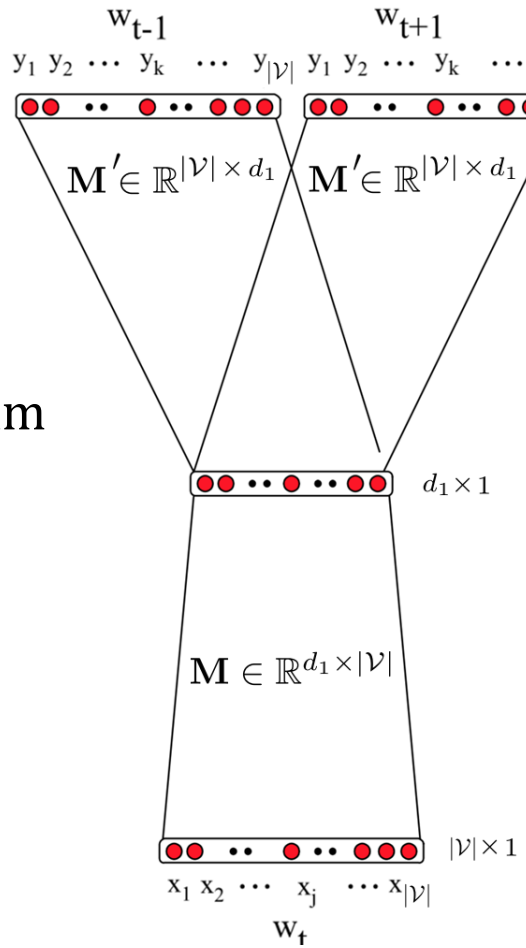
- 词嵌入学习

- Word2Vec技术： *Continuous Bag-of-Words (CBOW)*

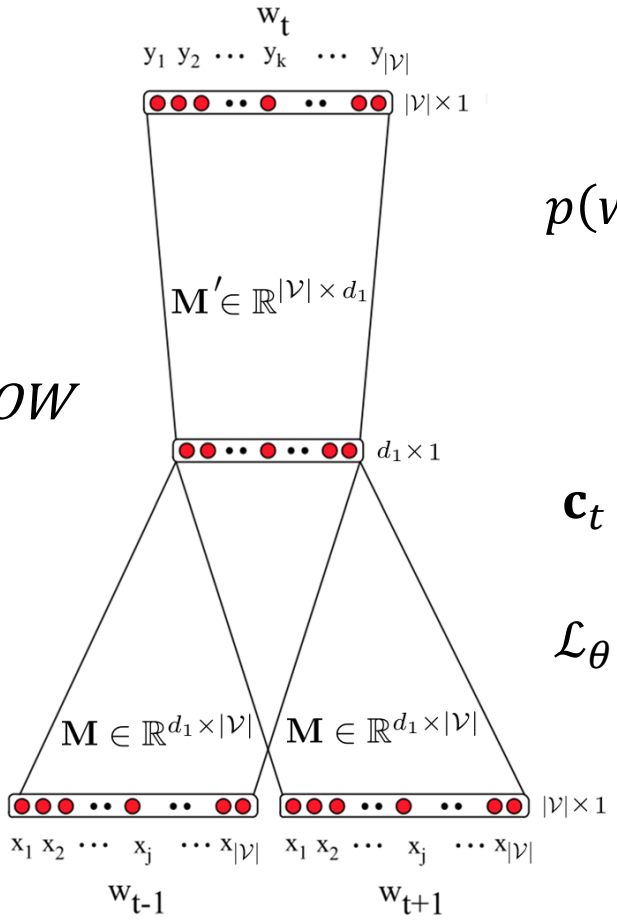
context window

$w_1, w_2, \ldots w_{t-m}, \ldots, w_{t-1}, w_t, w_{t+1} \ldots, w_{t+m}, \ldots, w_{T-1}, w_T$

Skip-Gram



CBOW



$$p(w_t \mid \mathbf{c}_t) = \text{softmax}\left(\mathbf{v}'^{\mathrm{T}}_{w_t} \mathbf{c}_t\right)$$

$$= \frac{\exp\left(\mathbf{v}'^{\mathrm{T}}_{w_t} \mathbf{c}_t\right)}{\sum_{w' \in \mathcal{V}} \exp\left(\mathbf{v}'^{\mathrm{T}}_{w_t} \mathbf{c}_t\right)}$$

$$\mathbf{c}_t = \sum_{-m \le j \le m, j \ne 0} \mathbf{v}_{w_{t+j}}$$

$$\mathcal{L}_\theta = -\frac{1}{T} \sum_{t=1}^{T} \log p(w_t \mid \mathbf{c}_t)$$

预训练技术

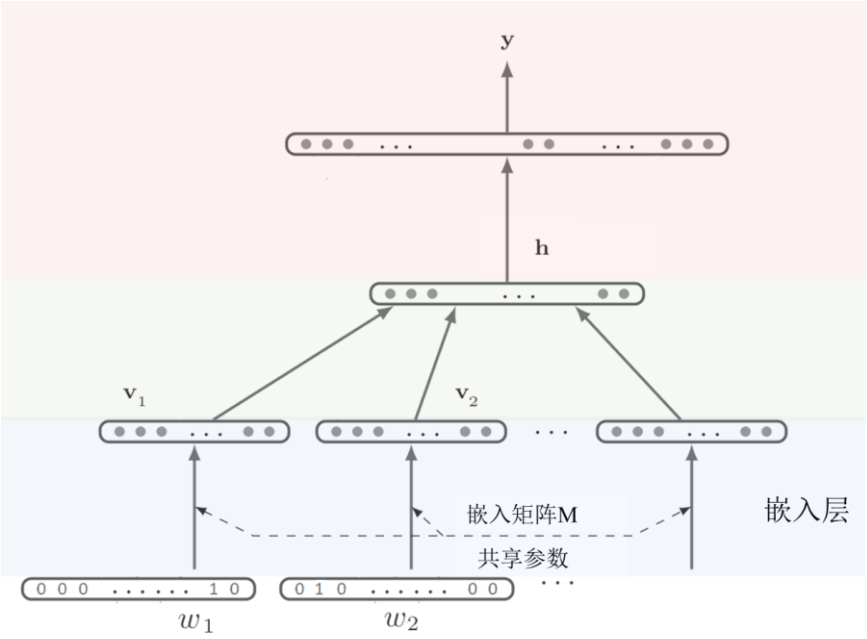
- 预训练初始化

神经网络的参数学习是一个非凸优化问题．当使用梯度下降法来进行优化网络参数时，参数初始值的选取十分关键，关系到网络的优化效率和泛化能力．不同的参数初始值会收敛到不同的局部最优解．虽然这些局部最优解在训练集上的损失比较接近，但是它们的泛化能力差异很大．

一个好的初始值会使得网络收敛到一个泛化能力高的局部最优解．

预训练技术指的是通过一个在大规模数据上学习出的模型可以提供一个好的参数初始值，这种初始化方法称为预训练初始化（Pre-trained Initialization）．

预训练初始化通常会提升模型泛化能力的一种解释是预训练任务起到一定的正则化作用．



预训练

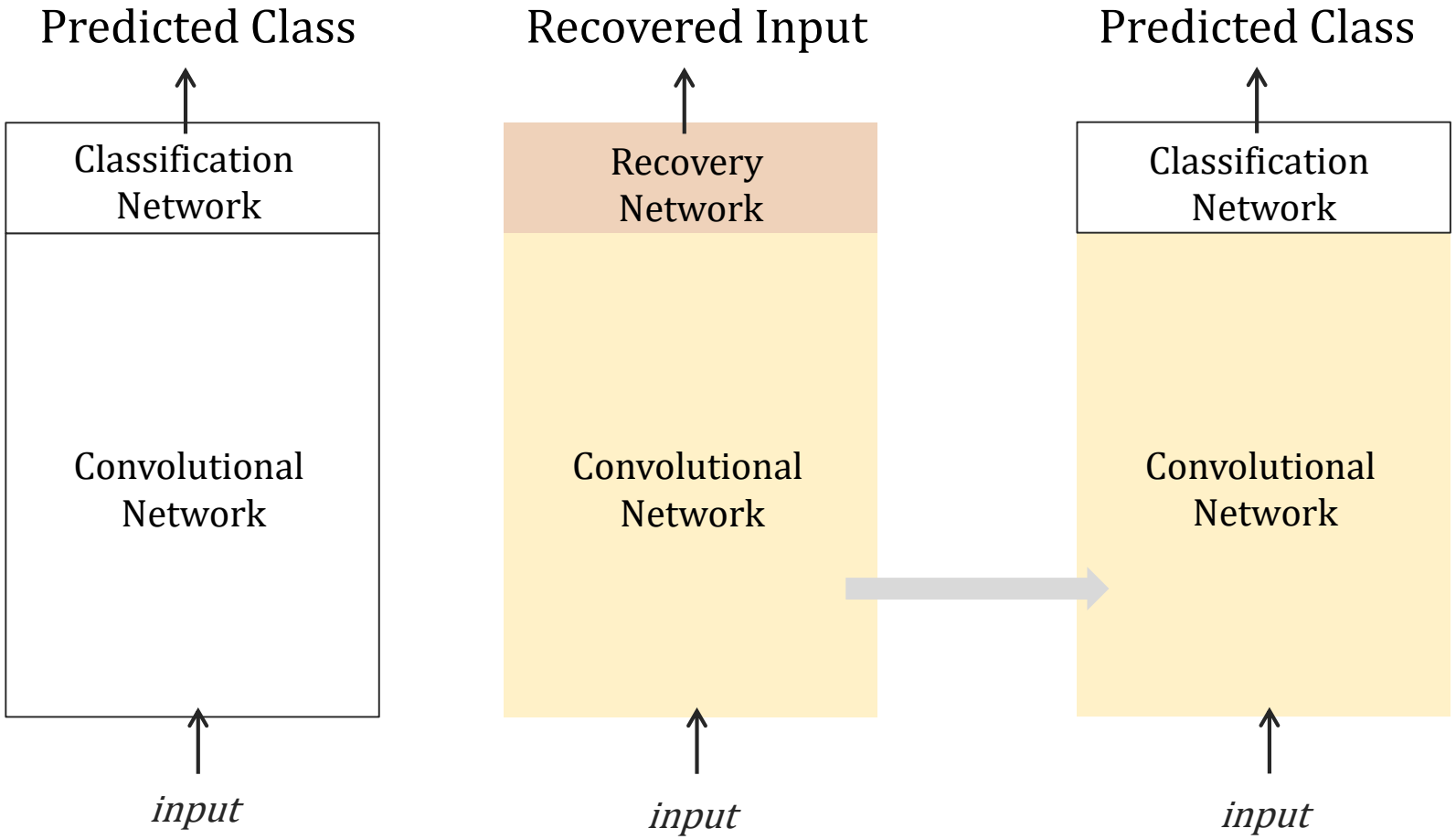
Word2Vec技术

预训练技术

- 参数迁移

对于一个监督学习任务，通常要面临标注样本的数量比较少的问题。

同时，我们常常可以低成本地获取大规模的无标注样本，因此一种自然的参数迁移方式是在大规模无标注样本上利用无监督学习任务学习到一个深度网络。然后将学习到的网络参数迁移到解决监督学习任务的网络上。

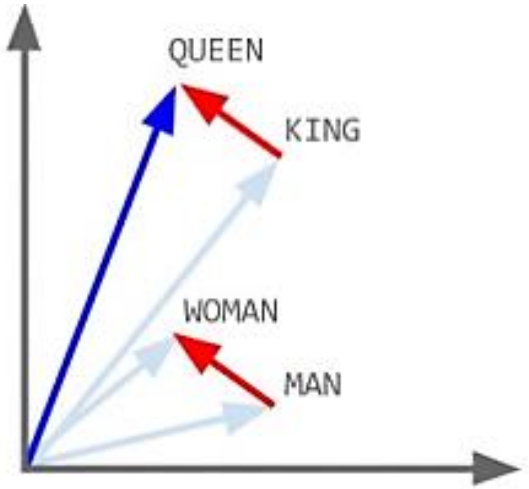


词向量实践

- King – Man + Woman = Queen

```
from gensim.models import word2vec
sentences = word2vec.Text8Corpus('text8’)
model = word2vec.Word2Vec(sentences, size=200)
model.save('text8.model')
model = word2vec.Word2Vec.load('text8.model')
most_similar = model.most_similar(positive=['woman', 'king'], negative=['man'],
topn=1)
print(most_similar)
```

```
[('queen', 0.6217384338378906)]
```



词向量实践

- 用已经训练好的词向量

```
w1 = model['cappuccino']
w2 = model['espresso']
dist = np.linalg.norm(w1-w2)
print('dist(cappuccino,espresso) is ' + str(dist))
```

```
KeyError: "word 'cappuccino' not in vocabulary"
```

词向量实践

- 用已经训练好的词向量 https://github.com/mmihaltz/word2vec-GoogleNews-vectors

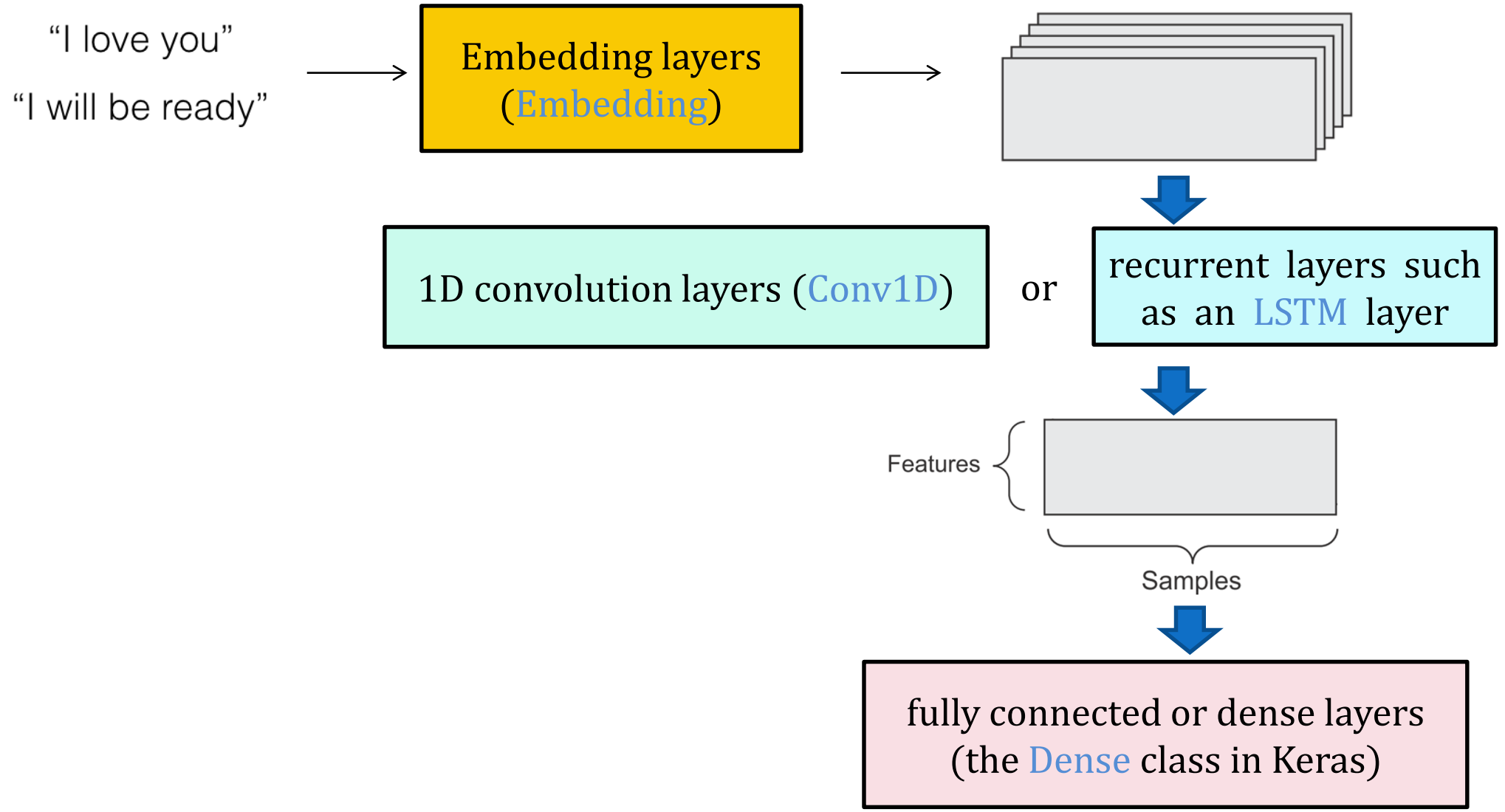
```
from gensim.models import KeyedVectors
import numpy as np
wv_path = './GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(wv_path, binary=True)
w1 = model['cat']
w2 = model['dog']
dist = np.linalg.norm(w1-w2)
print('dist(cat,dog) is ' + str(dist))
w1 = model['cat']
w2 = model['car']
dist = np.linalg.norm(w1-w2)
print('dist(cat,car) is ' + str(dist))
w1 = model['cappuccino']
w2 = model['espresso']
dist = np.linalg.norm(w1-w2)
print('dist(cappuccino,espresso) is ' + str(dist))
```

```
dist(cat,dog) is 2.08153
dist(cat,car) is 3.57391
dist(cappuccino,espresso) is 2.71879
```

word2vec-GoogleNews-vectors: This repository hosts the word2vec pre-trained Google News corpus (3 billion running words) word vector model (3 million 300-dimension English word vectors).

词向量实践

- 基于Keras



词向量实践

- 基于Keras

```
from keras.datasets import imdb
from keras import preprocessing
max_features = 10000
maxlen = 20
batch_size = 32
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
print(x_train[0])
```

```
[  65   16   38 1334   88   12   16  283    5   16 4472  113  103   32
   15   16 5345   19  178   32]
```

词向量实践

- 基于Keras

```
from keras.models import Sequential
from keras.layers import Embedding,Flatten, Dense
emb_dense = Sequential()
emb_dense.add(Embedding(10000, 8, input_length=maxlen))
emb_dense.add(Flatten())
emb_dense.add(Dense(1, activation='sigmoid'))
emb_dense.compile(optimizer='rmsprop', loss='binary_crossentropy',
metrics=['acc'])
emb_dense.summary()
print('training emb_dense')
history = emb_dense.fit(x_train, y_train,
                        epochs=10,
                        batch_size=batch_size,
                        validation_split=0.2)
plot_history(history, 'emb_dense')
```

词向量实践

- 基于Keras

```
from keras.models import Sequential
from keras.layers import Embedding,Flatten, Dense
emb_dense = Sequential()
emb_dense.add(Embedding(10000, 8, input_length=maxlen))
emb_dense.add(Flatten())
emb_dense.add(Dense(1, activation='sigmoid'))
emb_dense.compile(optimizer='rmsprop', loss='binary_crossentropy',
metrics=['acc'])
emb_dense.summary()
print('training emb_dense')
history = emb_dense.fit(x_train, y_train,
                        epochs=10,
                        batch_size=batch_size,
                        validation_split=0.2)
plot_history(history, 'emb_dense')
```

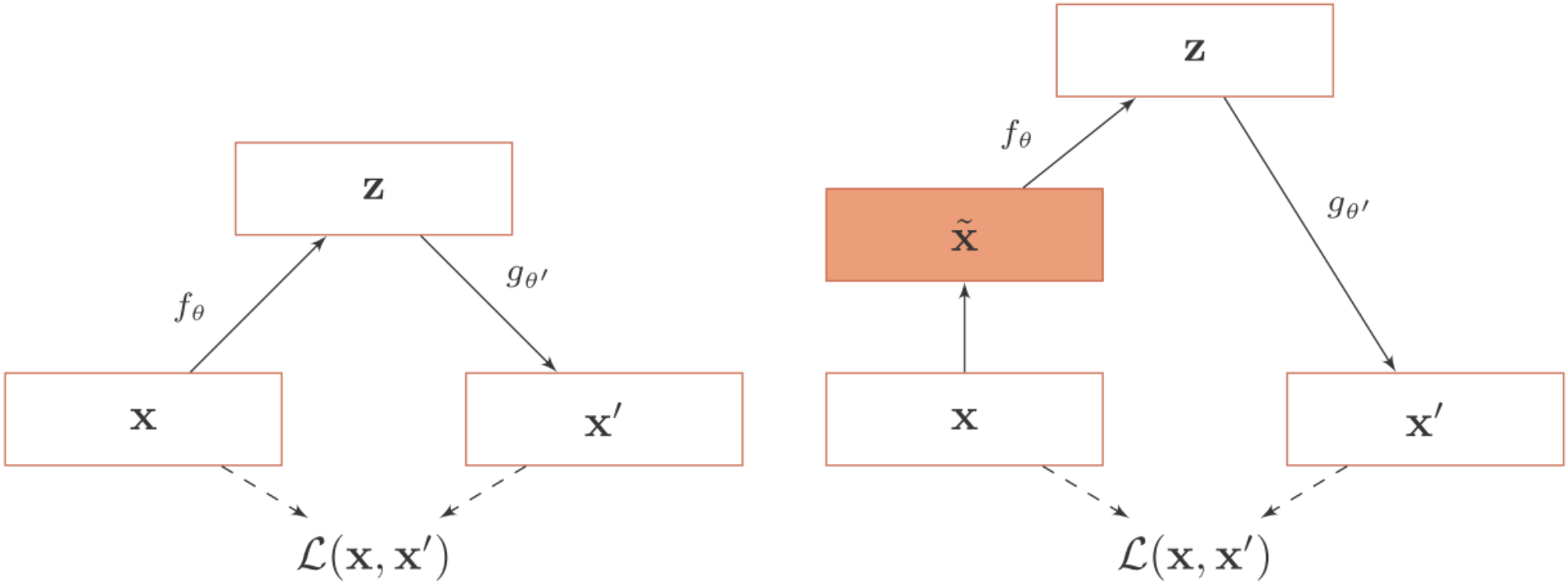
词向量实践

- 基于Keras

```
from keras.layers import SimpleRNN

emb_rnn = Sequential()
emb_rnn.add(Embedding(max_features, 32))
emb_rnn.add(SimpleRNN(32))
emb_rnn.add(Dense(1, activation='sigmoid'))
emb_rnn.compile(optimizer='rmsprop', loss='binary_crossentropy',
metrics=['acc'])
print('training emb_rnn')
history = emb_rnn.fit(x_train, y_train,
                      epochs=10,
                      batch_size=128,
                      validation_split=0.2)
```

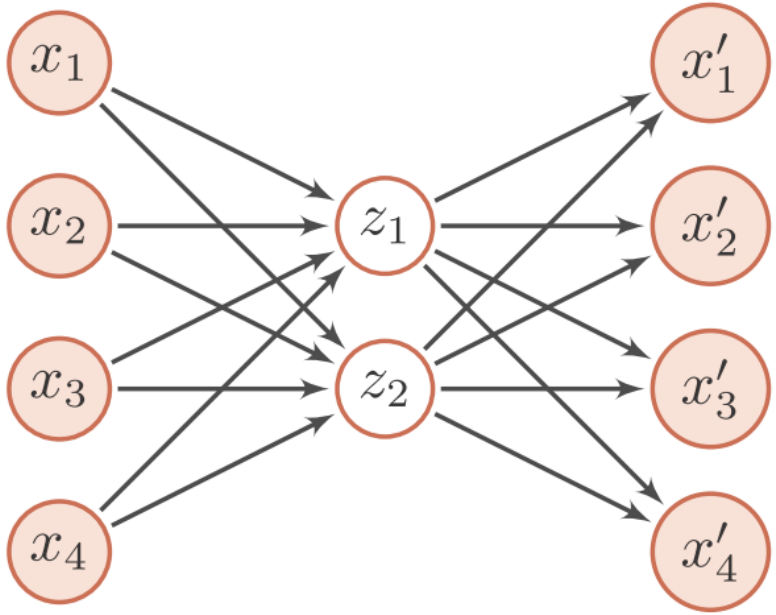
自动编码器

- 为了得到有效的数据表示，我们还可以要求数据表示具备其它性质，比如对数据部分损坏 Partial Destruction 的鲁棒性。这可以通过引入噪声来增加编码鲁棒性。



自编码器的实践

```
from keras.layers import Input, Dense
from keras.models import Model
encoding_dim = 32

input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)
autoencoder = Model(input_img, decoded)
print(autoencoder.summary())

autoencoder.compile(optimizer='adadelta',
loss='binary_crossentropy')
```



自编码器的实践

```
import numpy as np
def load_mnist_data(path='mnist.npz'):
    """Loads the MNIST dataset.
    """
    f = np.load(path)
    x_train, y_train = f['x_train'], f['y_train']
    x_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (x_train, y_train), (x_test, y_test)
(x_train, _), (x_test, _) = load_mnist_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train),
np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

自编码器的实践

```
#decoded_imgs = autoencoder.predict(x_test)
ori_input = Input(shape=(784,))
encoder = Model(ori_input,  autoencoder.get_layer('dense_1')(ori_input))
encoded_input = Input(shape=(encoding_dim,))
decoder = Model(encoded_input,
autoencoder.get_layer('dense_2')(encoded_input))
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

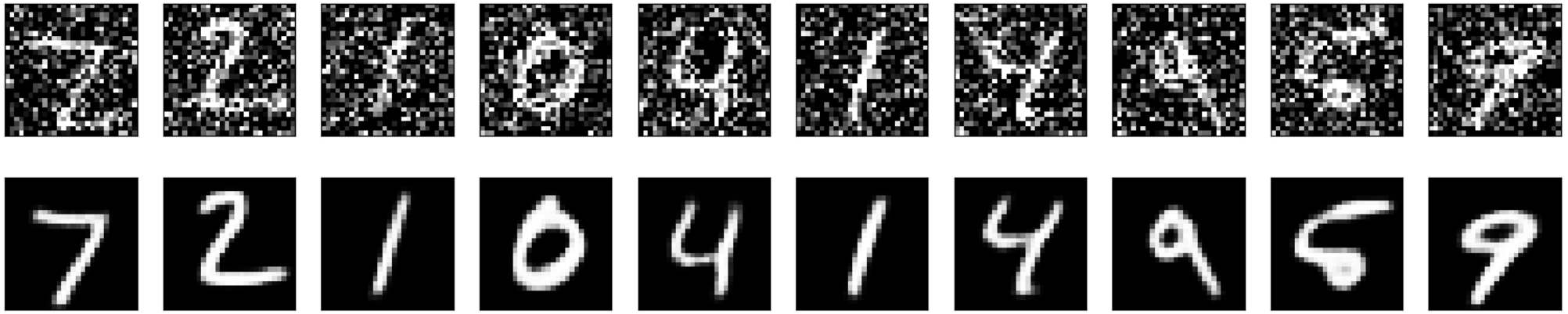
```
import matplotlib.pyplot as plt
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

自编码器的实践

自编码器的实践

```
input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)
```

自编码器的实践



自编码器的实践

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

自编码器的实践

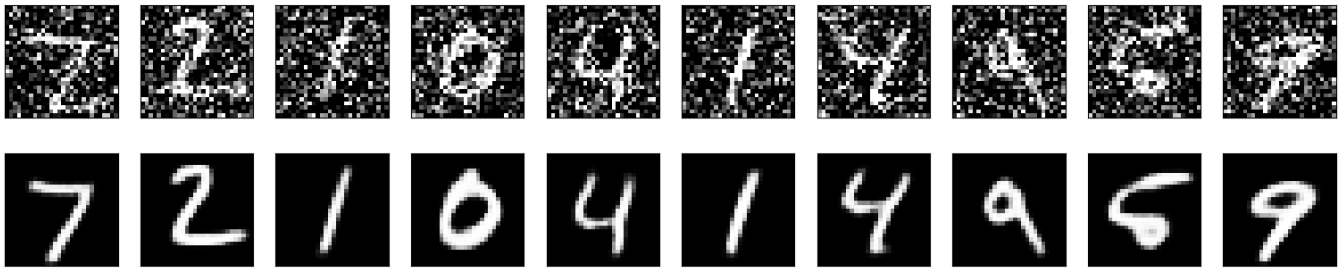
```
input_img = Input(shape=(28, 28, 1))

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (7, 7, 32)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
print(autoencoder.summary())
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

自编码器的实践

```
autoencoder.fit(x_train_noisy, x_train,
                epochs=10,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test_noisy, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/tb', histogram_freq=0,
write_graph=False)])
```

自编码器的实践

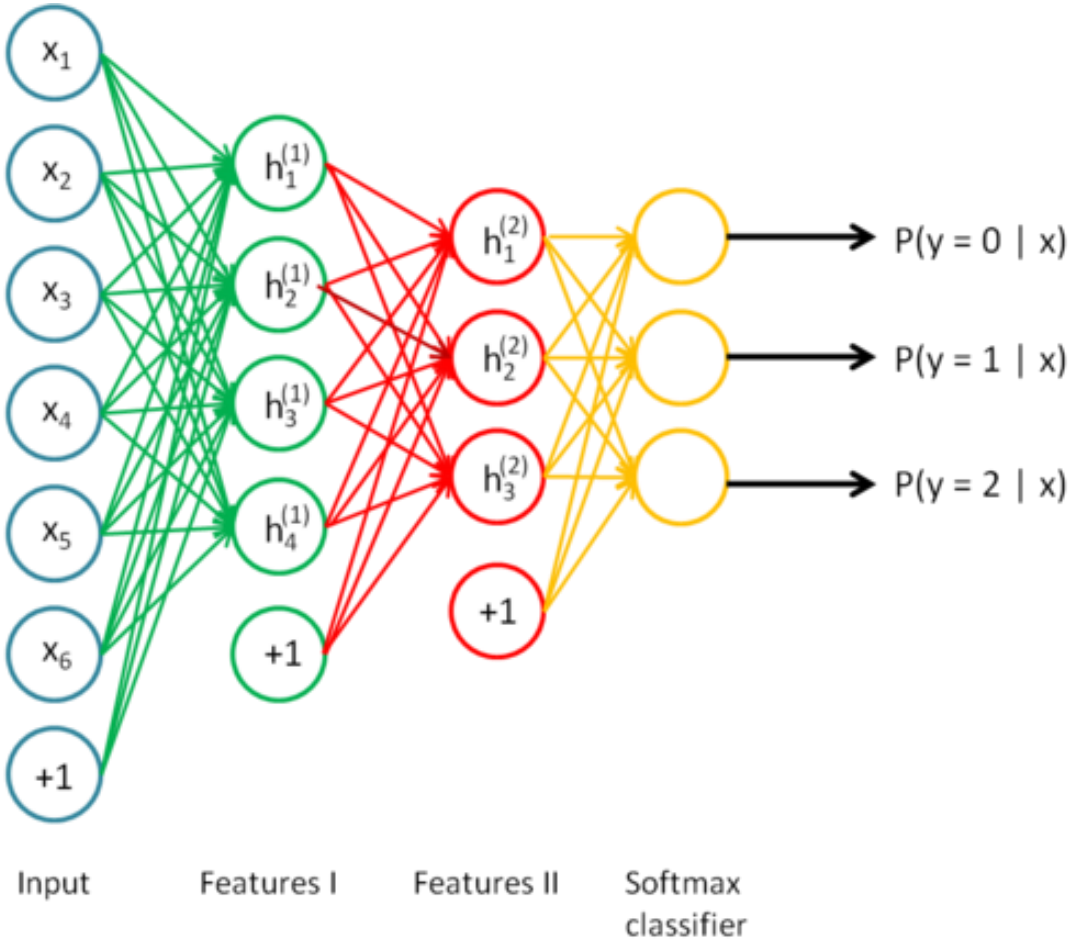
```
decoded_imgs = autoencoder.predict(x_test)
n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



预训练-精调 *pre-training and fine-tuning*

- 思想

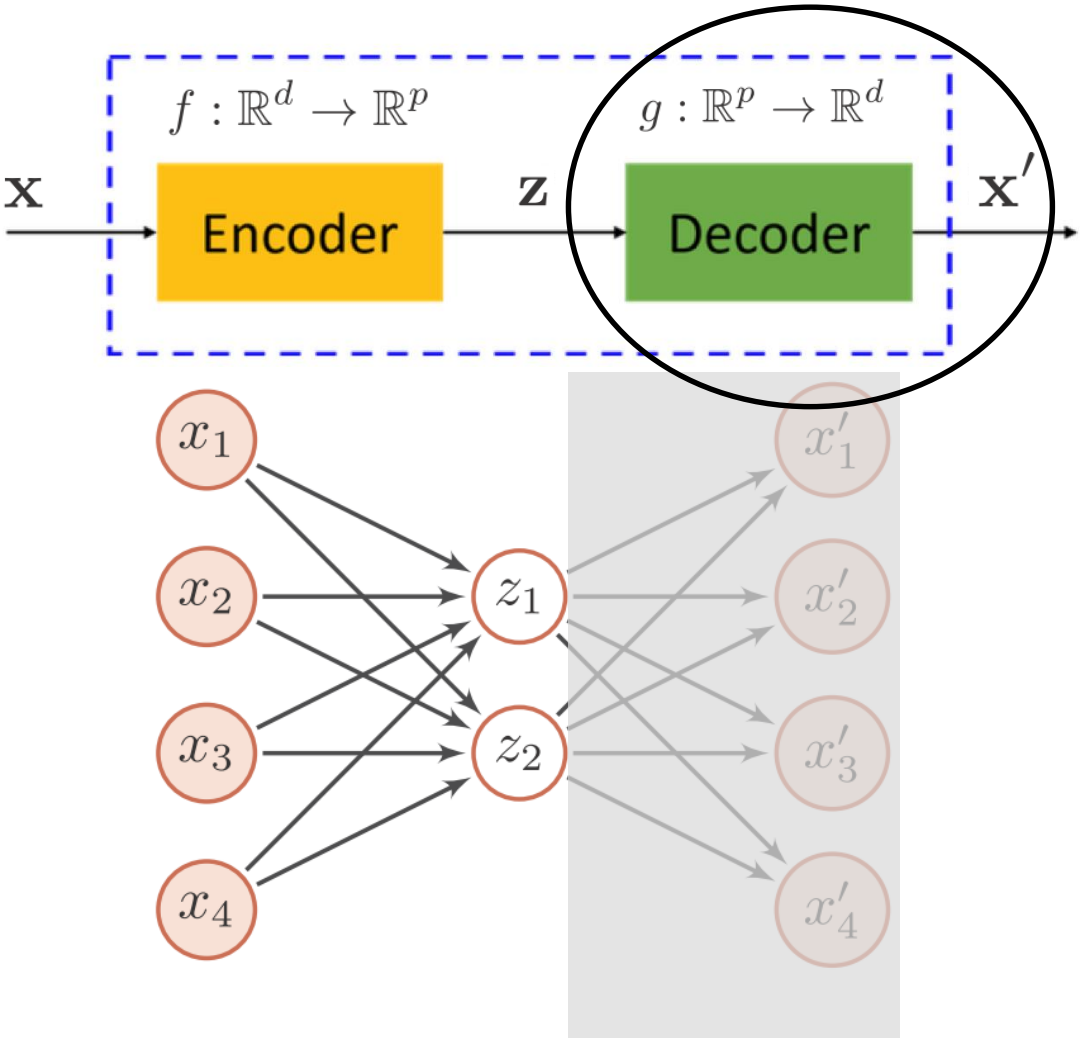
2006 年，Hinton and Salakhutdinov [2006] 发现多层前馈神经网络可以先通过逐层预训练，再用反向传播算法进行精调的方式进行有效学习。深度神经网络的训练过程可以通过先通过逐层预训练将模型的参数初始化为较优的值，再通过传统学习方法对参数进行精调。



预训练-精调 *pre-training and fine-tuning*

- 利用自动编码器做预训练

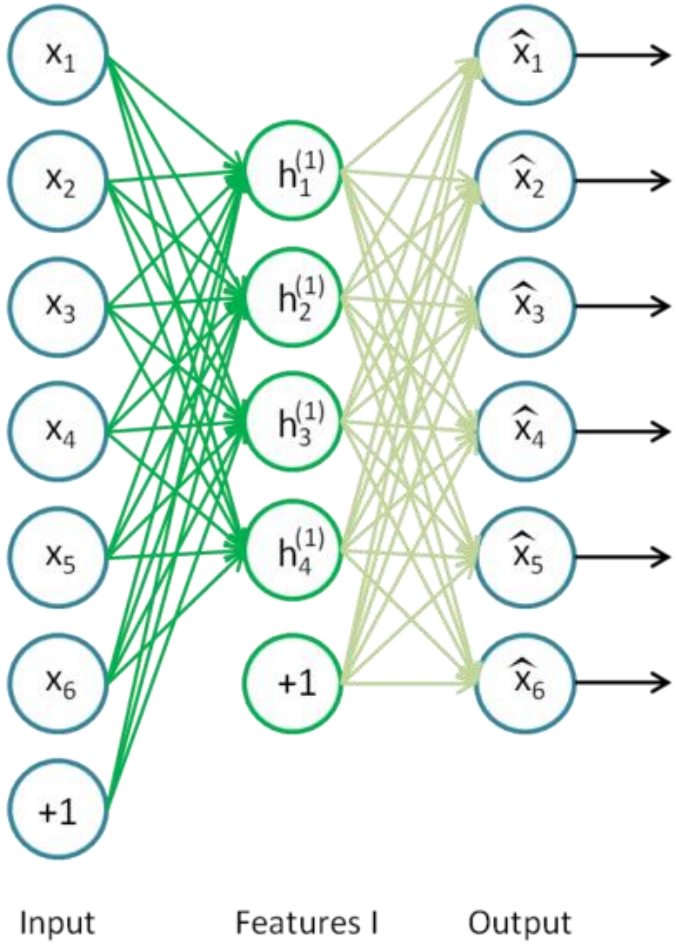
使用自编码器常常是为了得到有效的数据表示，因此在训练结束后，我们一般去掉解码器，只保留编码器。编码器的输出可以直接作为后续机器学习模型的输入。



预训练-精调 *pre-training and fine-tuning*

- 利用自动编码器做预训练

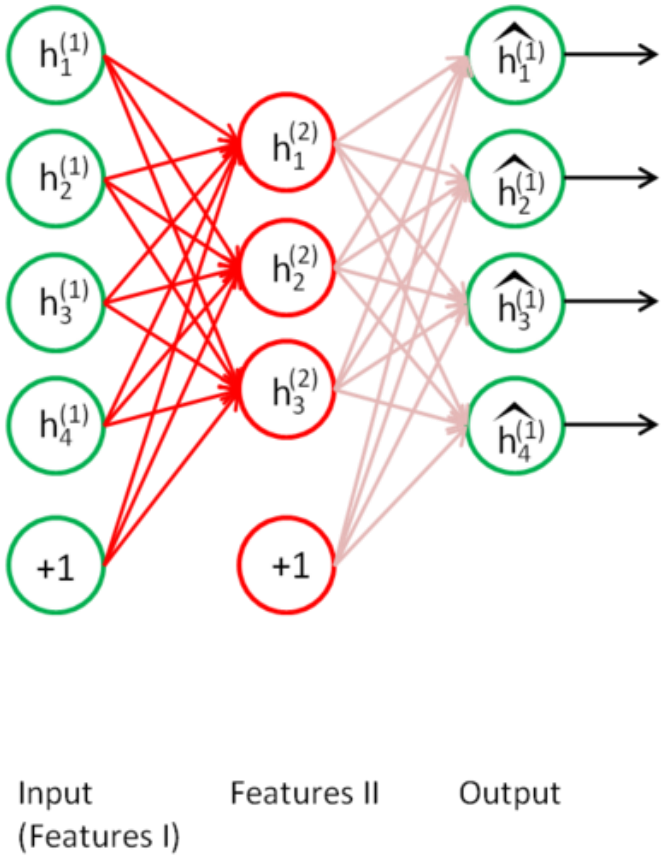
首先，用原始输入训练第一个自编码器，它能够学习得到原始输入的一阶特征表示



预训练-精调 *pre-training and fine-tuning*

- 利用自动编码器做预训练

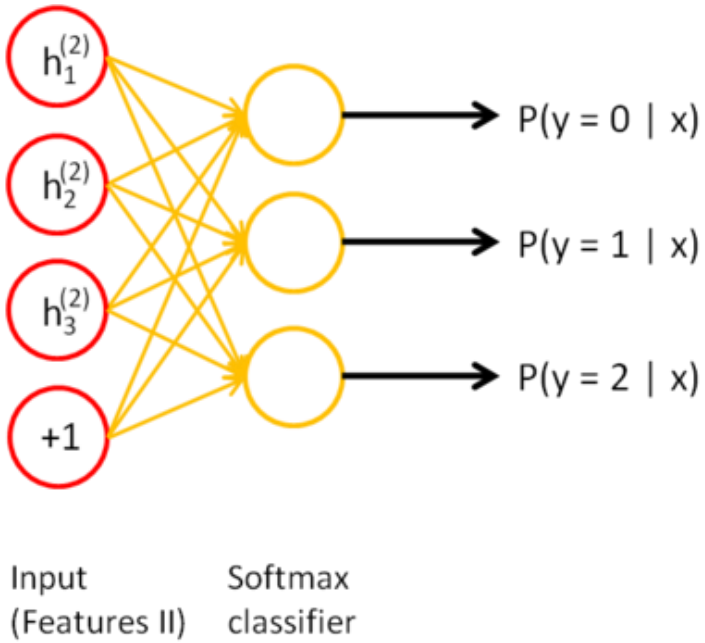
接着，需要把原始数据输入到上述训练好的稀疏自编码器中，对于每一个输入，都可以得到它对应的一阶特征表示，然后再用这些一阶特征作为另一个稀疏自编码器的输入，使用它们来学习二阶特征



预训练-精调 *pre-training and fine-tuning*

- 利用自动编码器做预训练

同样，再把一阶特征输入到刚训练好的第二层稀疏自编码器中，得到每个对应的二阶特征激活值接下来，把这些二阶特征作为softmax分类器的输入，训练得到一个能将二阶特征映射到类别标签的模型。

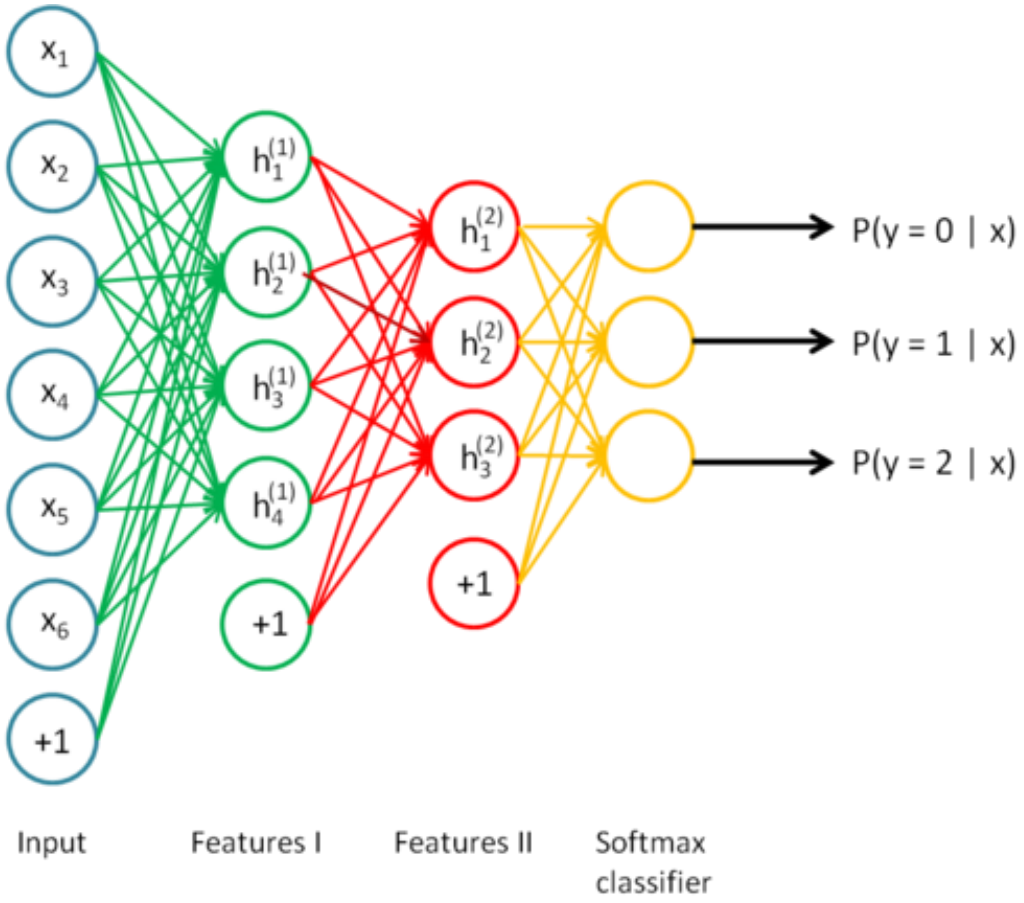


预训练-精调 *pre-training and fine-tuning*

- 利用自动编码器做预训练

之后，将上面三层结合起来构建一个包含两个隐藏层和一个最终softmax分类器层的栈式自编码网络。

上述步骤称为逐层预训练。完成之后通常还会精调，将所有层视为一个模型，这样在每次迭代中，网络中所有的权重值都可以得到优化。



利用训练好的词向量做预训练

```
import os

imdb_dir = './aclImdb'
train_dir = os.path.join(imdb_dir, 'train')
labels = []
texts = []
for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding='utf-8')
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

利用训练好的词向量做预训练

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100  # We will cut reviews after 100 words
training_samples = 200  # We will be training on 200 samples
validation_samples = 10000  # We will be validating on 10000 samples
max_words = 10000  # We will only consider the top 10,000 words in the dataset

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

利用训练好的词向量做预训练

```
data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

利用训练好的词向量做预训练

```
embeddings_index = {}
f = open('./glove.6B.100d.txt', encoding='utf-8')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Found %s word vectors.' % len(embeddings_index))

embedding_dim = 100
embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector
```

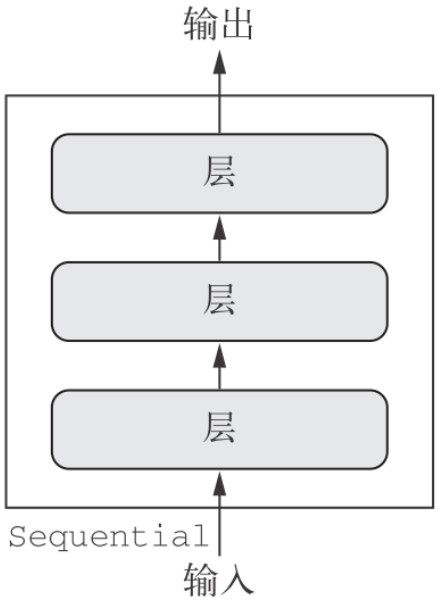
利用训练好的词向量做预训练

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

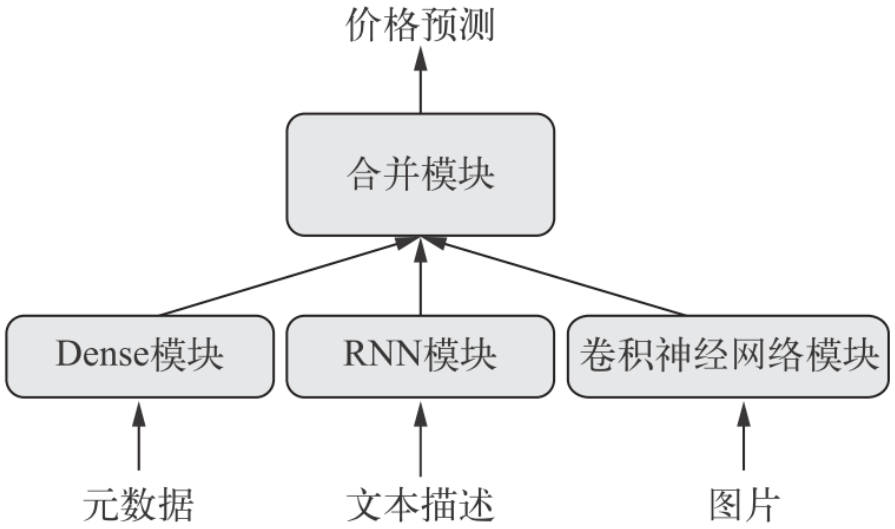
利用训练好的词向量做预训练

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = True # or False

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=30,
                    batch_size=32,
                    validation_data=(x_val, y_val))
```

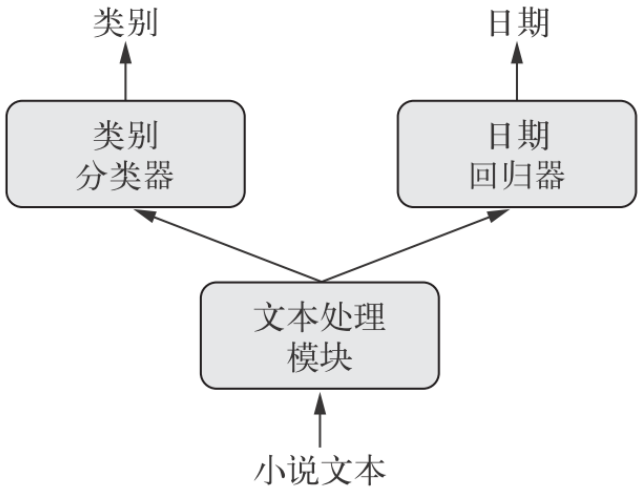
Keras for flexible networks



A sequential model

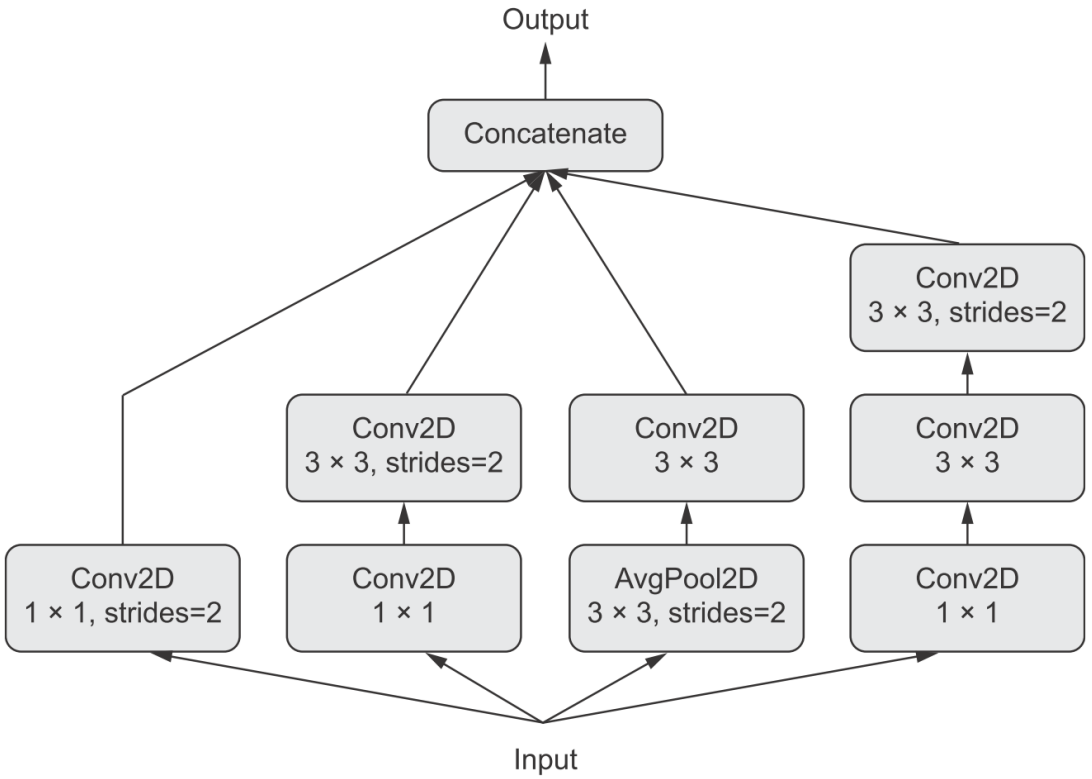


A multi-input model



A multi-output (or multihead) model

Keras for flexible networks



An graph-like model: a subgraph of layers with several parallel convolutional branches

Keras for flexible networks

- 函数式API *functional API*

Keras里定义模型有两种方法：一种是使用 Sequential 类，其仅用于层的线性堆叠，另一种是函数式 API functional API，用于层组成的有向无环图，让你可以构建任意形式的架构。

使用函数式 API，你可以直接操作张量，也可以把层当作函数来使用，接收张量并返回张量（因此得名函数式 API）

```
from keras import Input, layers
input_tensor = Input(shape=(32,))
dense = layers.Dense(32, activation='relu')
output_tensor = dense(input_tensor)
```


“层”作为一个函数被使用

Keras for flexible networks

- 函数式API *functional API*

Keras里定义模型有两种方法：一种是使用 Sequential 类，其仅用于层的线性堆叠，另一种是函数式 API functional API，用于层组成的有向无环图，让你可以构建任意形式的架构。

使用函数式 API，你可以直接操作张量，也可以把层当作函数来使用，接收张量并返回张量（因此得名函数式 API）

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu',
input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Sequential 模型

对应的函数式 API 实现

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

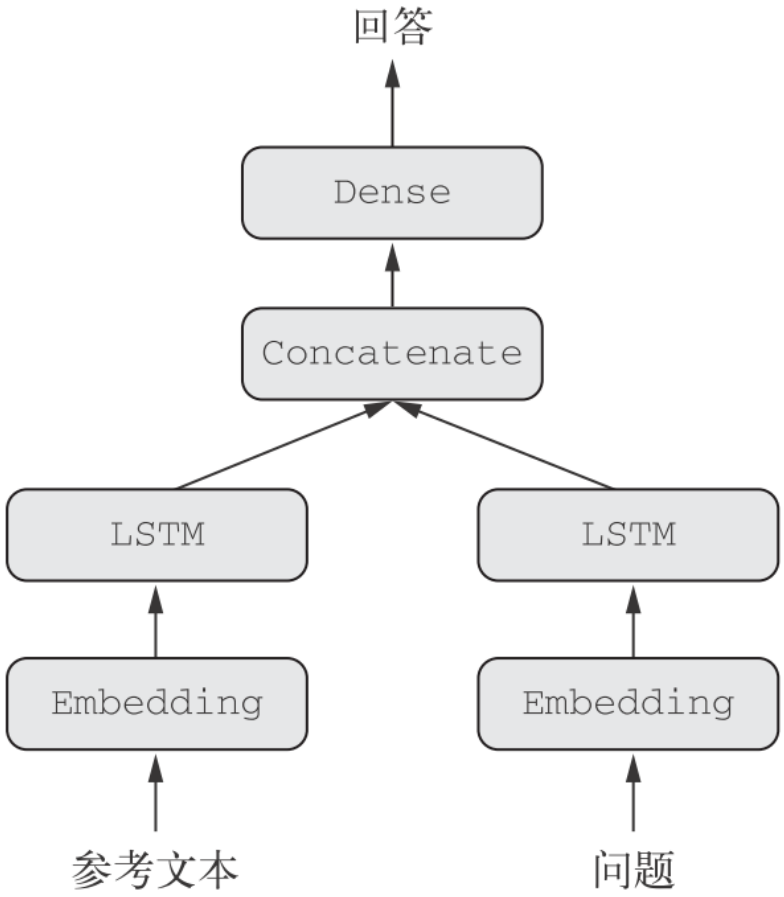
Model 类将输入张量和输出张量转换为一个模型

Keras for flexible networks

- 函数式API *functional API*

- 多输入模型

Step 0. 设定text_vocabulary_size, question_vocabulary_size, 和 answer_vocabulary_size



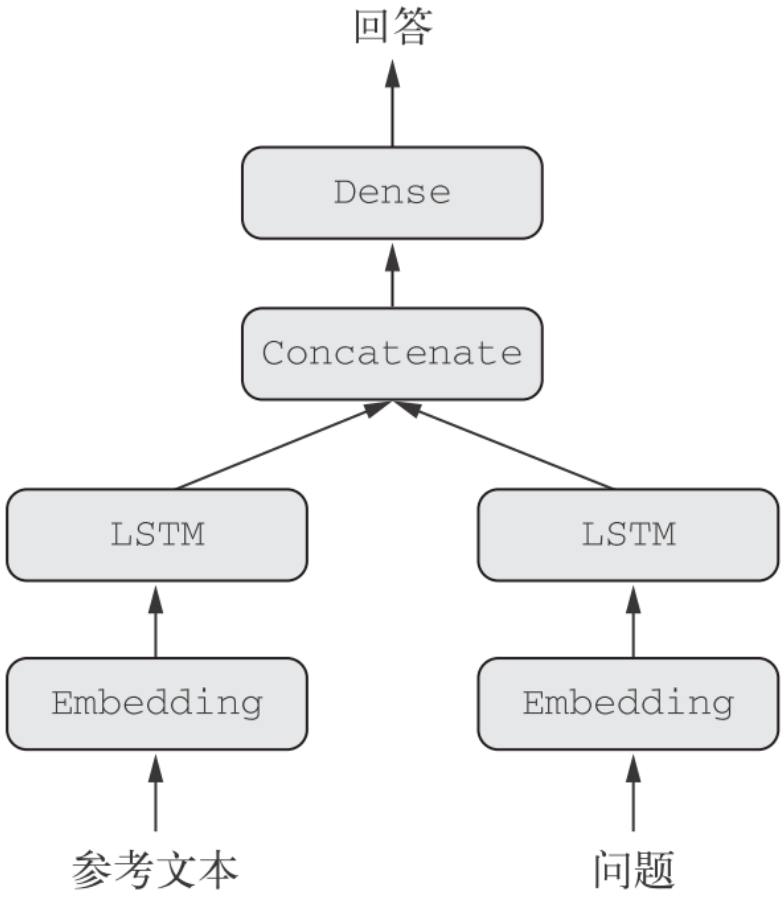

```
text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500
```

Keras for flexible networks

- 函数式API *functional API*

- 多输入模型

Step 1. 设定两个独立的分支，分别对应于将参考文本和问题进行编码



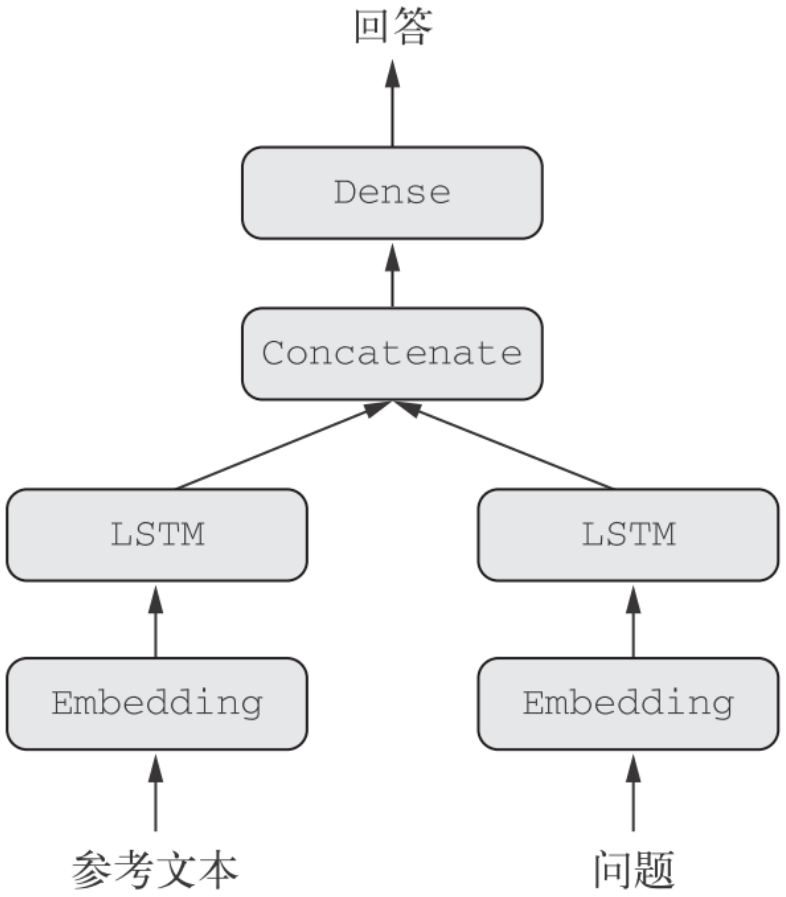

```
text_input = Input(shape=(None,),
dtype='int32', name='text')
embedded_text = layers.Embedding(64,
text_vocabulary_size)(text_input)
encoded_text = layers.LSTM(32)(embedded_text)
question_input = Input(shape=(None,),
dtype='int32', name='question')
embedded_question = layers.Embedding(32,
question_vocabulary_size)(question_input)
encoded_question =
layers.LSTM(16)(embedded_question)
```

Keras for flexible networks

- 函数式API *functional API*

- 多输入模型

Step 2. 串联 *concatenate* 编码后的向量



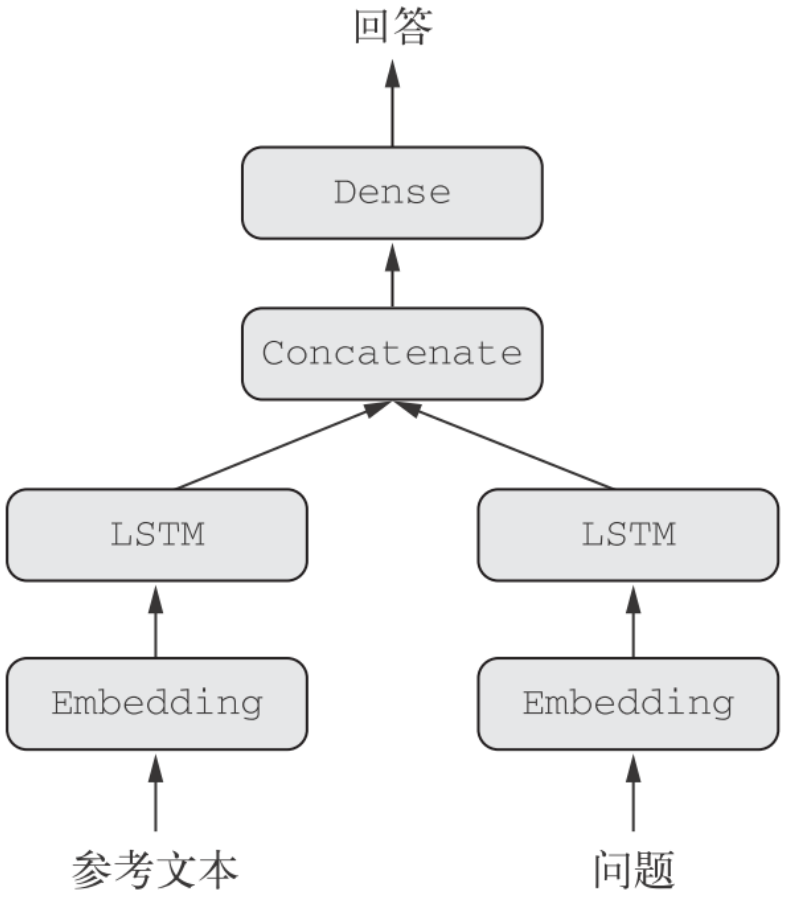

```
concatenated =
layers.concatenate([encoded_text,
encoded_question], axis=-1)
```

Keras for flexible networks

- 函数式API *functional API*

- 多输入模型

Step 3. 在串联之后的表达上添加softmax分类器

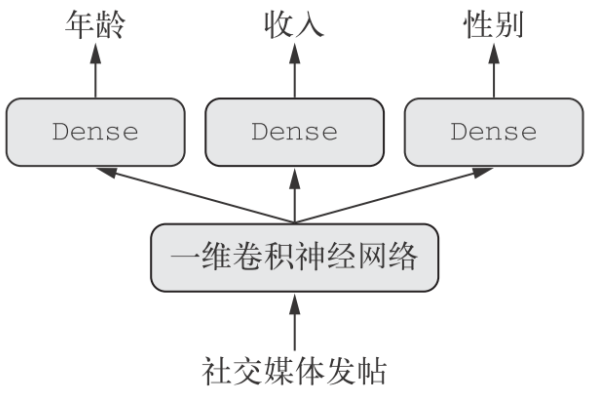



```
answer = layers.Dense(answer_vocabulary_size,
activation='softmax')(concatenated)
```

```
from keras.models import Model
model = Model([text_input, question_input], answer)
```

Keras for flexible networks

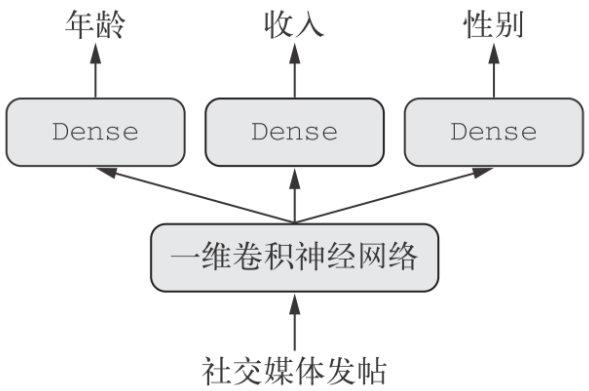
- 函数式API
  - 多输出模型




```
from keras import layers
from keras import Input
from keras.models import Model
vocabulary_size = 50000
num_income_groups = 10
posts_input = Input(shape=(None,), dtype='int32', name='posts')
embedded_posts = layers.Embedding(256,
vocabulary_size)(posts_input)
x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation='relu')(x)
age_prediction = layers.Dense(1, name='age')(x)
income_prediction = layers.Dense(num_income_groups,
activation='softmax', name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid',
name='gender')(x)
model = Model(posts_input, [age_prediction, income_prediction,
gender_prediction])
```

Keras for flexible networks

- 函数式API
  - 多输出模型



训练这种模型需要能够对网络的各个头指定不同的损失函数。可以在编译时使用损失组成的列表或字典来为不同输出指定不同损失，然后将得到的损失值相加得到一个全局损失，并在训练过程中将这个损失最小化。

```
model.compile(optimizer='rmsprop', loss=['mse',
'categorical_crossentropy', 'binary_crossentropy'])
```

也可以在给定输出层名字下设定

```
model.compile(optimizer='rmsprop', loss={'age': 'mse',
'income': 'categorical_crossentropy', 'gender':
'binary_crossentropy'})
```

不平衡的损失贡献会导致模型表示针对单个损失值最大的任务优先进行优化

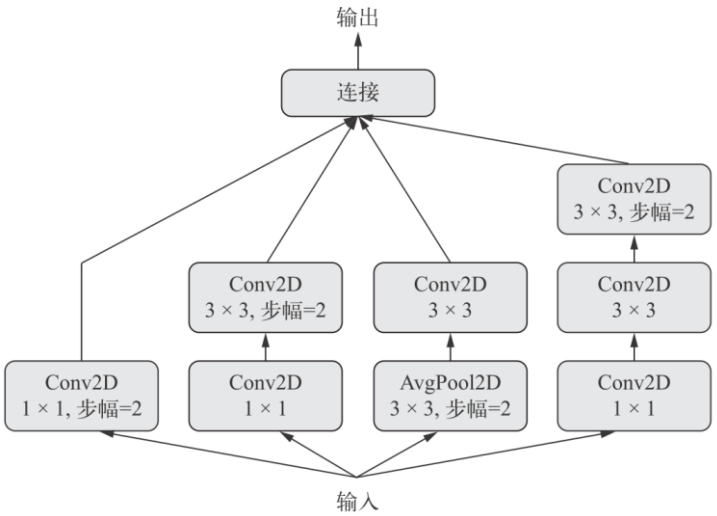
```
model.compile(optimizer='rmsprop', loss=['mse',
'categorical_crossentropy', 'binary_crossentropy'],
loss_weights=[0.25, 1., 10.])
```

```
model.compile(optimizer='rmsprop', loss={'age': 'mse',
'income': 'categorical_crossentropy', 'gender':
'binary_crossentropy'}, loss_weights={'age': 0.25,
'income': 1., 'gender': 10.})
```

Keras for flexible networks

- 函数式API

- Graph like




```
from keras import layers
branch_a = layers.Conv2D(128, 1,
activation='relu', strides=2)(x)
branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu',
strides=2)(branch_b)
branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3,
activation='relu')(branch_c)
branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3,
activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu',
strides=2)(branch_d)
output = layers.concatenate([branch_a, branch_b,
branch_c, branch_d], axis=-1)
```