

Deep Learning

Lab11: TensorFlow 101

Bing-Han Chiang & Datalab

Outline

- TensorFlow
- Environment Setup
- Getting Started with TensorFlow
 - Load Dataset
 - Build Model via Sequential API
 - Build Model by Model Subclassing
 - Custom Training
- Autograph
- (Advanced) Gradient Flow Customization
- Assignment: Word2vec
 - Custom Layer
 - Build Model by Functional API

TensorFlow

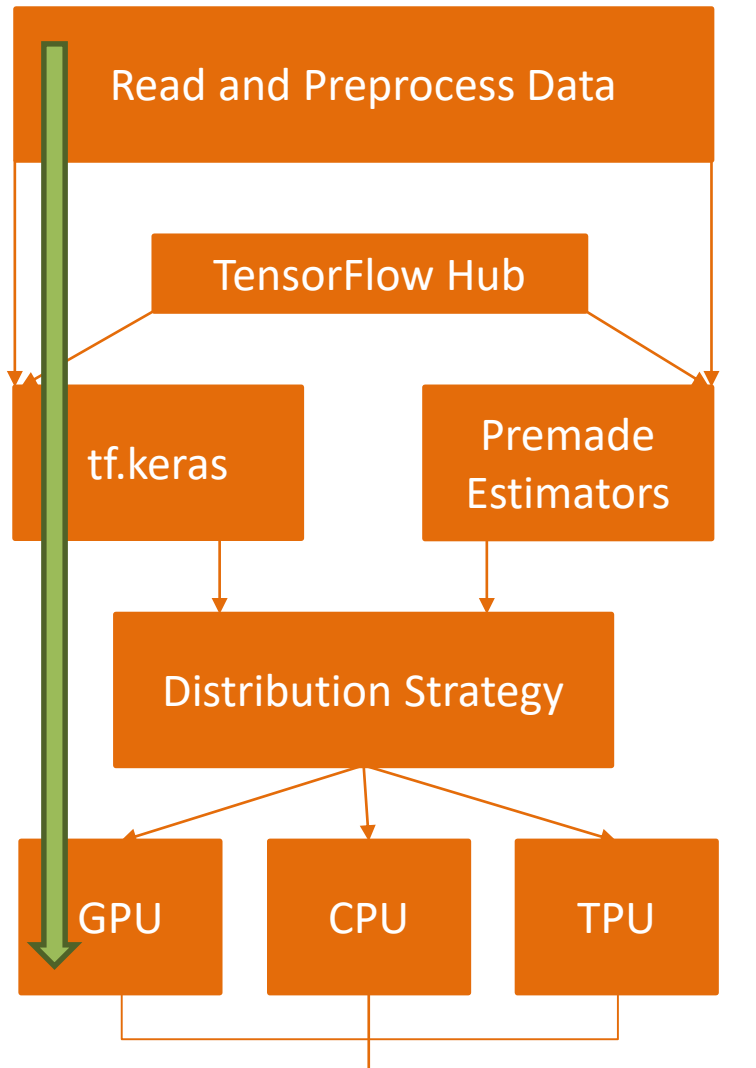
- TensorFlow was originally created by Google as an internal machine learning tool
- For a framework to be useful in production, it needs to be efficient, scalable, and maintainable
- For research, the framework needs to have flexible operations that can be combined in novel ways



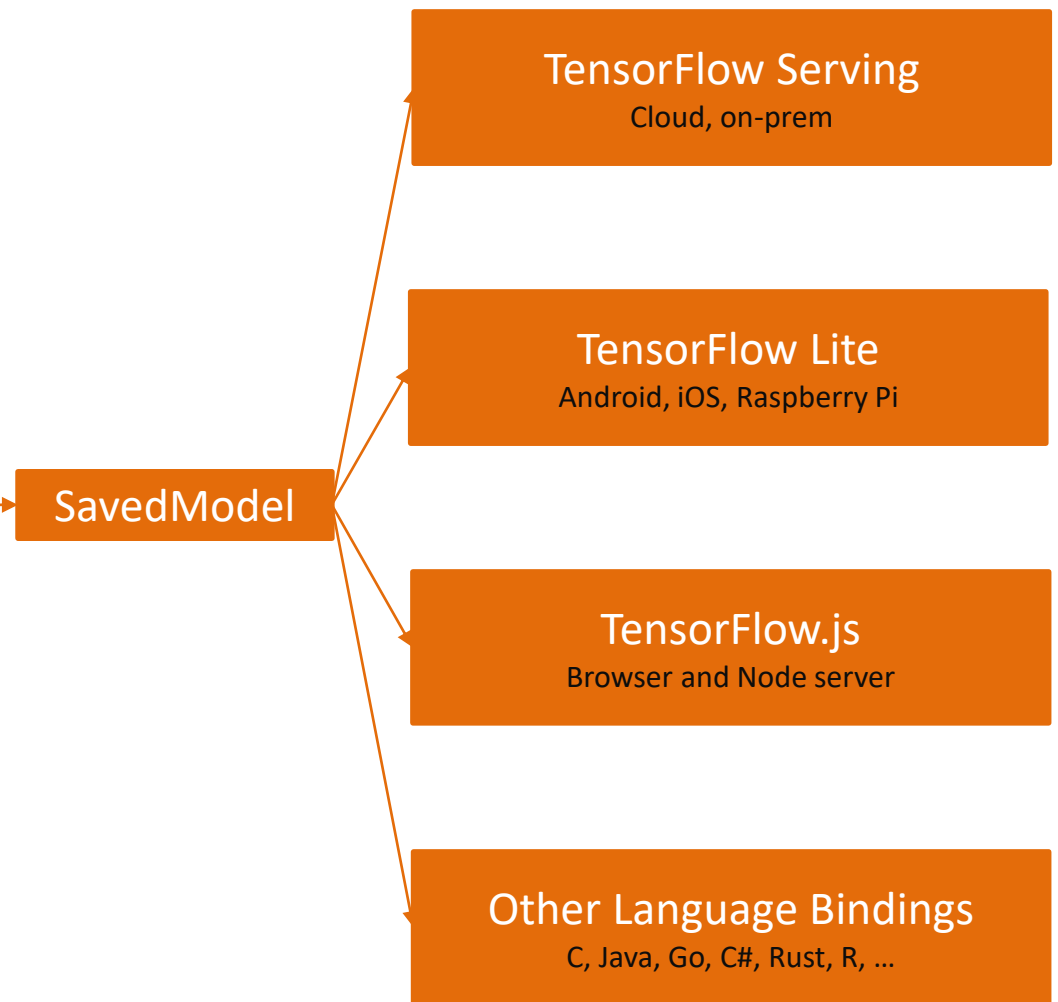
TensorFlow

TensorFlow

TRAINING



DEPLOYMENT



TensorFlow

- In this class, we will use **TensorFlow2.0**, which features
 - Eager Execution By Default
 - Simplified APIs
 - Tight Integration with Keras
 - High flexibility

Environment Setup

1. Install TensorFlow with Python's *pip* package manager

```
> pip install tensorflow-gpu
```

system requirements:

- Python 3.4 or later
- pip 19.0 or later
- ubuntu 16.04 or later (64-bit)
- macOS 10.12.6 (Sierra) or later (64-bit) (*no GPU support*)
- Windows 7 or later (64-bit) (*Python 3 only*)

Environment Setup

2. To enable GPU support, the following NVIDIA® software must be installed:

- [NVIDIA® GPU drivers](#) — CUDA 10.0 requires 410.x or higher.
- [CUDA® Toolkit](#) — TensorFlow supports CUDA 10.0 (TensorFlow >= 1.13.0)
- [cuDNN SDK](#) (>= 7.4.1)
- Please refer to official TensorFlow website([GPU Support](#)) for more detailed and latest information
- (Optional) If you are using Anaconda environment, you can install corresponding *CUDA® Toolkit* and *cuDNN SDK* via

```
> conda install cudnn=7.6.0=cuda10.0_0
```

Notice that you still have to install NVIDIA® GPU drivers on your own.

Environment Setup

3. Google Colab

Colaboratory is a Jupyter notebook environment with free GPU(NVIDIA Tesla K80) that requires no setup and runs entirely in the cloud.

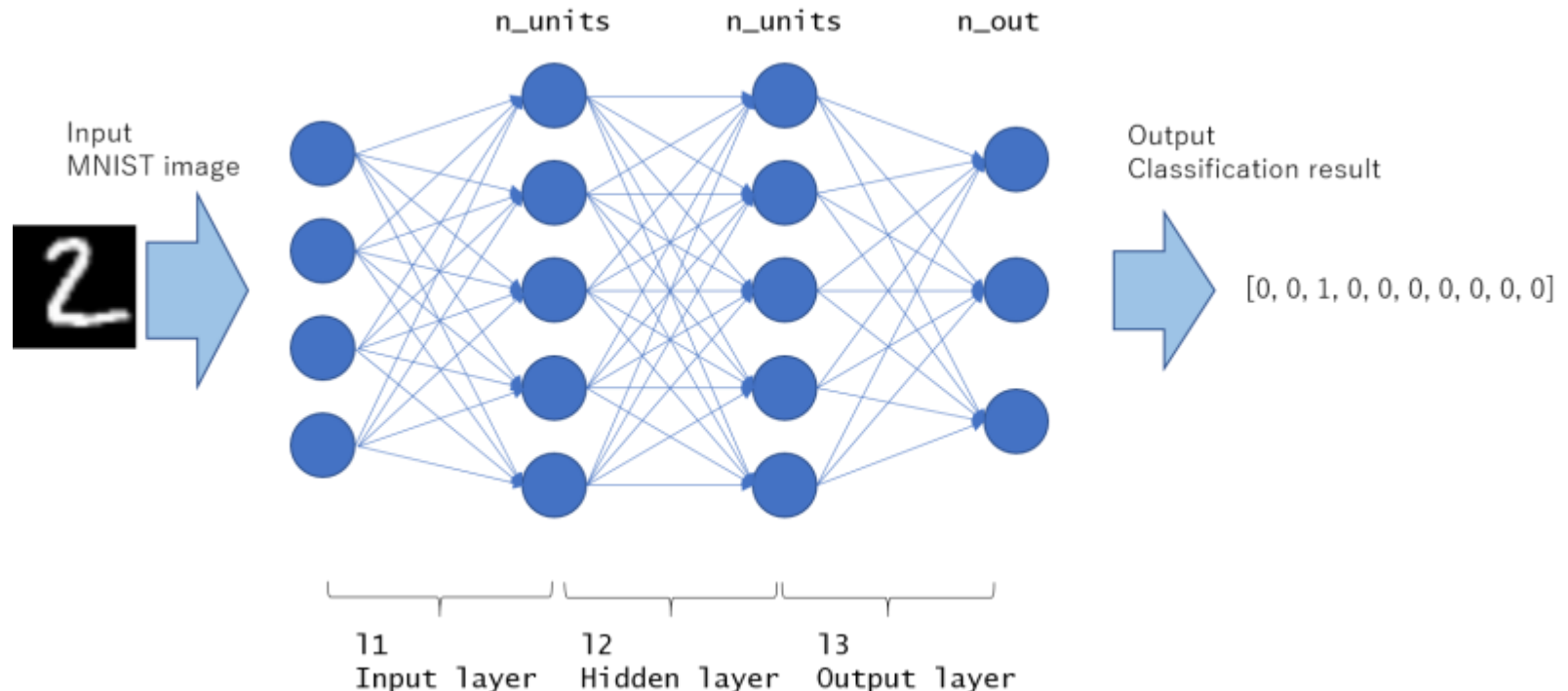
However, colab has a *12-hour* limit for a continuous assignment of VM, which means you can only train a model continuously for 12 hrs

- To utilize the GPU power, don't forget to change runtime type to GPU
- To check whether Colab is connected to a environment with GPU, type following command in the cell

```
> !nvidia-smi
```


Getting Started with TensorFlow

- Later on we will talk about how to build a simple deep neural network to classify digital numbers
- Dataset: MNIST
 - Hand-written digit ranges from 0 to 9



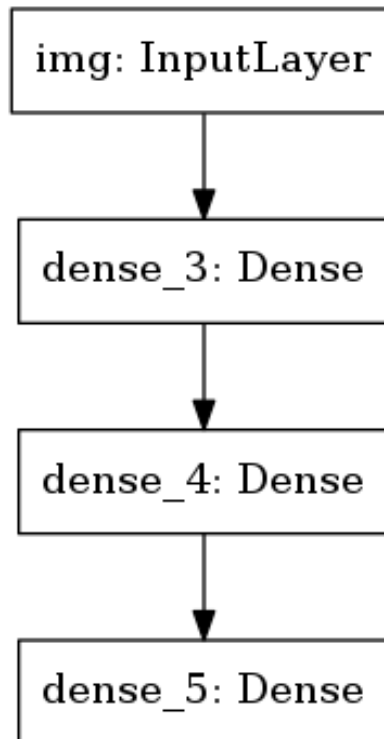
Load dataset

- Currently, `tf.keras.dataset` supports 7 datasets, including
 - **boston_housing** module: Boston housing price regression dataset.
 - **cifar10** module: CIFAR10 small images classification dataset.
 - **cifar100** module: CIFAR100 small images classification dataset.
 - **fashion_mnist** module: Fashion-MNIST dataset.
 - **imdb** module: IMDB sentiment classification dataset.
 - **mnist** module: MNIST handwritten digits dataset.
 - **reuters** module: Reuters topic classification dataset.
- Load mnist module via following command:

```
> (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

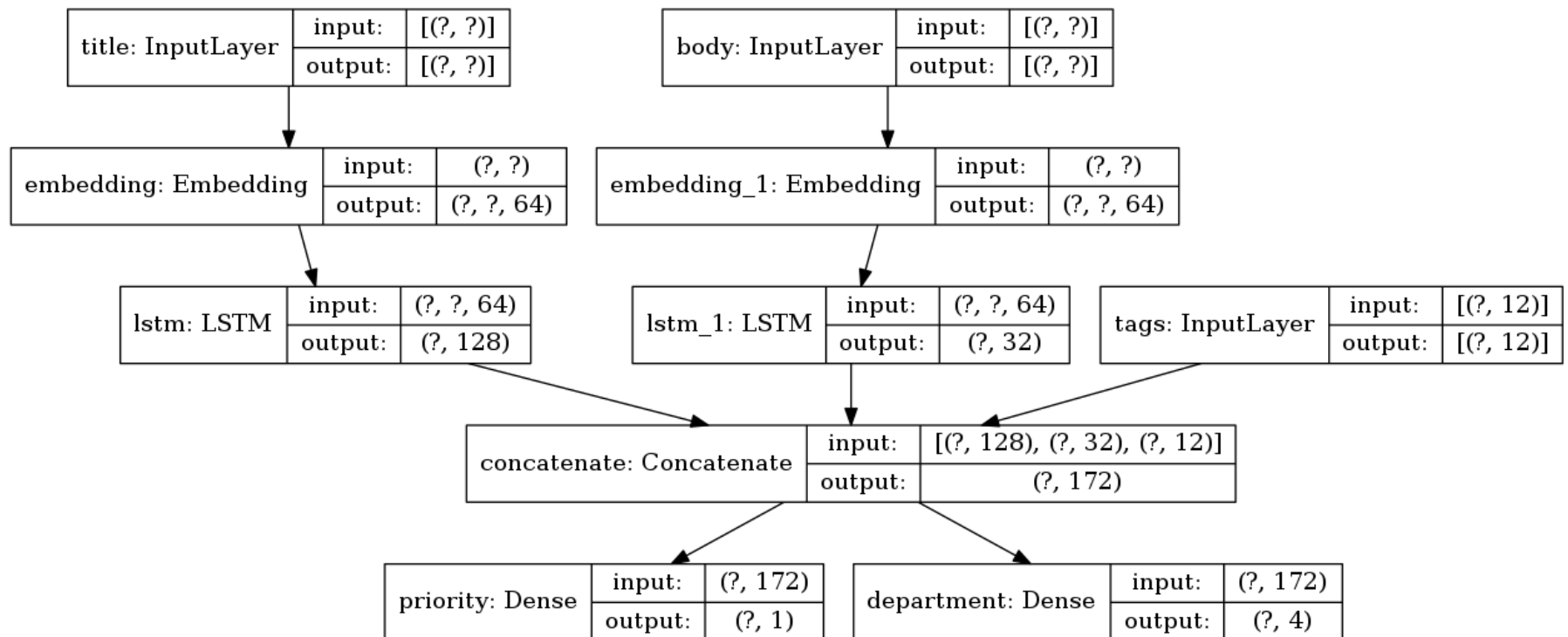
Build Model via Sequential API

- Sequential API is useful for building the model with a single forward path. For example,



Build Model via Sequential API

- However, Sequential API cannot handle models with non-linear topology, models with shared layers, and models with multiple inputs or outputs. For example,



Build Model via Sequential API

- To classify MNIST, let's build a simple neural network with fully connected layers.
- Stack layers by `tf.keras.Sequential`, and choose an optimizer and loss function for training.

First fully connected layer

Flatten input shape from (28, 28) to (784)

```
model = tf.keras.Sequential([  
    tf.keras.layers.Flatten(input_shape=(28, 28)),  
    tf.keras.layers.Dense(128, activation='relu'),  
    tf.keras.layers.Dropout(0.2),  
    tf.keras.layers.Dense(10, activation='softmax')  
])
```

To prevent overfit

Classification layer

Choose a suitable optimizer
from [tf.keras.optimizers](#)

```
model.compile(optimizer='adam',  
              loss='sparse_categorical_crossentropy',  
              metrics=['accuracy'])
```

Build Model via Sequential API

- Train and evaluate model by simple `fit` and `evaluate` .

feed in training data and label

set up batch size and training epoch

```
model.fit(x_train, y_train, batch_size=32, epochs=5)  
model.evaluate(x_test, y_test, verbose=2)
```

feed in testing data and label

Build Model via Model Subclassing

- Model subclassing gives you the ability to build whatever model structure you want

```
from tensorflow.keras import Model
```

```
class MyModel(Model):
```

```
    def __init__(self):
```

```
        super(MyModel, self).__init__()
```

```
        self.flatten = Flatten()
```

```
        self.dropout = Dropout(0.2)
```

```
        self.d1 = Dense(128, activation='relu')
```

```
        self.d2 = Dense(10, activation='softmax')
```

Define layers when initializing



```
    def call(self, x):
```


```
        x = self.flatten(x)
```

```
        x = self.d1(x)
```

```
        x = self.dropout(x)
```

```
        return self.d2(x)
```

Define feed-forward
path when calling



```
model = MyModel()
```

Custom Training

- Whatever you are using sequential API or model subclassing, you can use `model.fit` and `model.evaluate` to train and evaluate your model.
- However, you can still customize training and evaluation step for more flexible usage.

Custom Training

- Choose the loss function and optimizer you want:

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()  
optimizer = tf.keras.optimizers.Adam()
```



The default learning rate is 1e-3

- Select the metrics to measure the loss and the accuracy in the training process:


```
train_loss = tf.keras.metrics.Mean(name='train_loss')  
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')  
  
test_loss = tf.keras.metrics.Mean(name='test_loss')  
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

Custom Training

- Define your custom training step and use `tf.GradientTape` to compute gradients.
- Operations are recorded if they are executed within the context manager and at least one of their inputs is being "watched".

```
def train_step(images, labels):  
    with tf.GradientTape() as tape:  
        predictions = model(images)  
        loss = loss_object(labels, predictions)  
    gradients = tape.gradient(loss, model.trainable_variables)  
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))  
  
    train_loss(loss)  
    train_accuracy(labels, predictions)
```

Trainable variables are automatically watched



Autograph

- TensorFlow 1.X requires users to build a static graph. The tensor inside it is unknown before calling `session.run()` .

```
with tf.Graph().as_default():  
    a = tf.constant(20.0)  
    print(a)  
    with tf.compat.v1.Session() as sess:  
        print(sess.run(a))
```

```
Tensor("Const:0", shape=(), dtype=float32)  
20.0
```

- In TensorFlow 2.0, eager execution is enabled by default. All the tensors can be evaluated eagerly.

```
a = tf.constant(20.0)  
print(a)  
print(a.numpy())
```

```
tf.Tensor(20.0, shape=(), dtype=float32)  
20.0
```

Autograph

- Although eager execution is convenient, it sacrifices the speed of static graph.
- However, in TensorFlow 2.0, you can still use `tf.function` to transform a subset of Python syntax into portable, high-performance TensorFlow graphs.

Autograph

- Let's create two functions with same content, except one of them is annotated by `tf.function`.

```
def f(x, y):  
    return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)  
  
@tf.function  
def g(x, y):  
    return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
```

- You can see that the function annotated by `tf.function` is a bit faster.

```
%time for i in range(10000): f(x, y)
```

```
CPU times: user 5.6 s, sys: 418 ms, total: 6.02 s  
Wall time: 3.64 s
```

```
%time for i in range(10000): g(x, y)
```

```
CPU times: user 4.82 s, sys: 550 ms, total: 5.37 s  
Wall time: 2.62 s
```

Gradient Flow Customization

- Consider the following function,

$$y = \log_e(1 + e^x)$$

- The derivative of it is,

$$\frac{dy}{dx} = \frac{e^x}{1+e^x} = 1 - \frac{1}{1+e^x}$$

- Due to numeric instability, the gradient evaluated at $x = 100$ will be *NaN*.

```
x = tf.constant(100.)
with tf.GradientTape() as g:
    g.watch(x)
    y = log1pexp(x)
dy_dx = g.gradient(y, x) # Will be evaluated as NaN.
dy_dx.numpy()
```

nan


Gradient Flow Customization

- The gradient expression can be analytically simplified to provide numerical stability by decorating the function with

`tf.custom_gradient` :

```
@tf.custom_gradient
def log1pexp(x):
    e = tf.exp(x)
    def grad(dy):
        return dy * (1 - 1 / (1 + e))
    return tf.math.log(1 + e), grad
```

```
x = tf.constant(100.)
with tf.GradientTape() as g:
    g.watch(x)
    y = log1pexp(x)
dy_dx = g.gradient(y, x) # Will be 1.0 when evaluated.
dy_dx.numpy()
```



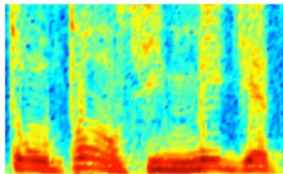
Manually watch input x

1.0

Word2vec

- Why learn word embeddings?
 - For tasks like object or speech recognition we know that all the information required to successfully perform the task is encoded in the data.

AUDIO



Audio Spectrogram

DENSE

IMAGES

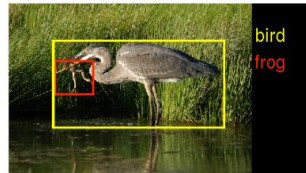
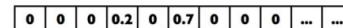


Image pixels

DENSE

TEXT



Word, context, or document vectors

SPARSE

- However, natural language processing system traditionally treat words as discrete atomic symbols, and therefore 'cat' may be represented as '2' and 'dog' as '1.'

Word2vec

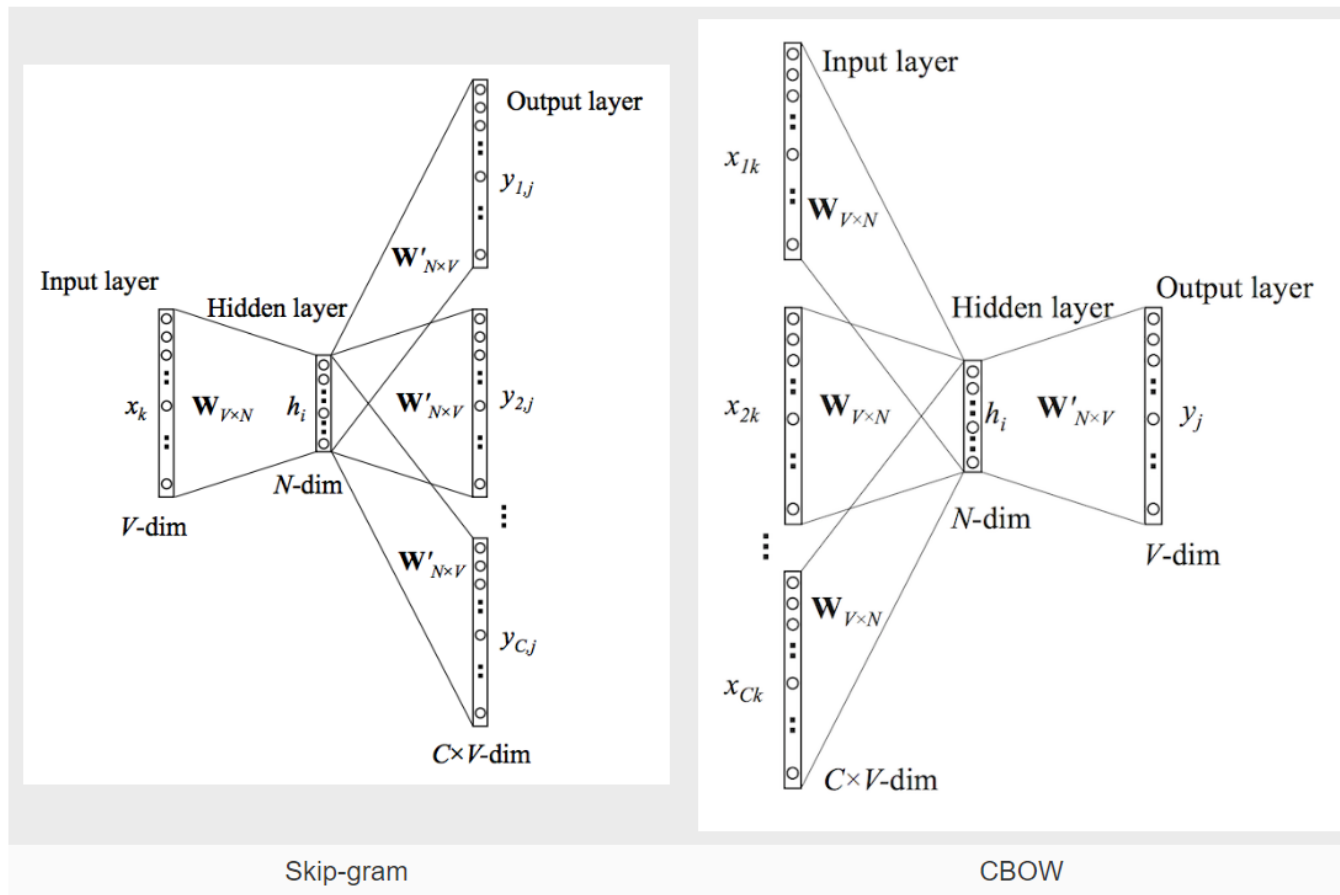
- Why learn word embeddings?
 - These encodings are arbitrary, and provide no useful information to the system regarding the relationships that may exist between the individual symbols.
 - **Vector space models (VSMs)**, which represent words as vectors can help overcome these obstacles. This is based on a key observation that **semantically similar words are often used interchangeably in different contexts**.
 - For example, the words `cat` and `dog` may both appear in a context “___ is my favorite pet.”

Word2vec

- Skip-Gram and CBOW
 - Word2vec comes in two flavors, the Continuous Bag-of-Words model (CBOW) and the Skip-Gram model.
 - CBOW predicts the target words using its neighborhood(context) whereas Skip-Gram does the inverse, which is to predict context words from the target words.
 - For example, given the sentence **the quick brown fox jumped over the lazy dog**.
 - CBOW will be trained on the dataset:
([the, brown], quick), ([quick, fox], brown), ...
 - Skip-Gram will be trained on the dataset:
(quick, [the, brown]), (brown, [quick, fox]), ...

Word2vec

- Skip-Gram and CBOW
 - We will focus on building the skip-gram model in the rest of the slides.



Word2vec

- Scaling up with noise-contrastive training
 - Neural probabilistic language models are traditionally trained using the maximum likelihood (ML) principle to maximize the probability of the next word w_t given the previous words h in terms of a softmax function,

$$P(w_t|t) = \text{softmax}(\text{score}(w_t, h)) = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word } w \text{ in Vocab}} \exp\{\text{score}(w, h)\}}$$

Word2vec

- Scaling up with noise-contrastive training
 - $\text{score}(w_t, h)$ computes the compatibility of word w_t with the context h .
 - Usually, we train the language model by maximizing its log-likelihood on the training set, i.e. by maximizing:

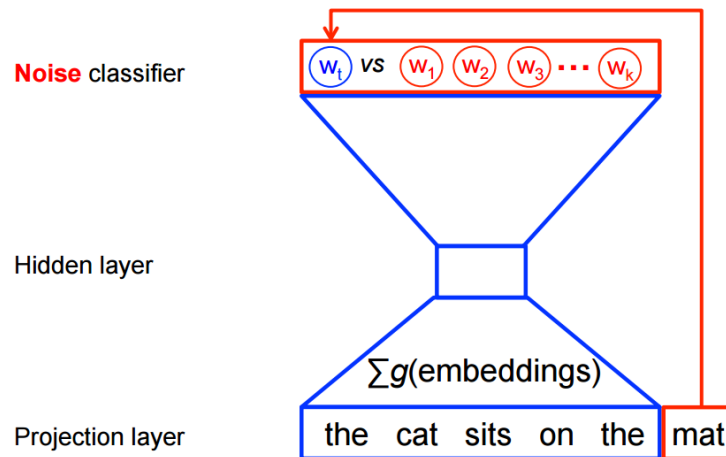
$$J_{ML} = \log(P(w_t|h)) = \text{score}(w_t, h) - \log\left(\sum_{\text{Word } w \text{ in Vocab}} \exp\{\text{score}(w, h)\}\right)$$

Word2vec

- Scaling up with noise-contrastive training
 - This yields a properly normalized probabilistic model for language modeling.
 - However, this is very expensive, because we need to compute and normalize each probability using the score for all other V words w in the current context h , *at every training step*.
 - Consider a language model with 50000 words, the outputs of it will be 5000 times larger compared to a classification model trained on MNIST, which has 10 labels only.

Word2vec

- Scaling up with noise-contrastive training
 - On the other hand, for feature learning in word2vec we do not need a full probabilistic model.
 - The **CBOW** and **skip-gram** models are instead trained using a binary classification objective to discriminate the real target words w_t from k imaginary (noise) words \tilde{w} , in the same context.
 - We illustrate this below for a CBOW model.



*Noise-contrastive training for
CBOW model*

Word2vec

- Scaling up with noise-contrastive training
 - Mathematically, the objective (for each example) is to maximize

$$J_{NEG} = \log Q_{\theta}(D = 1|w_t, h) + k \cdot \mathbb{E}[\log Q_{\theta}(D = 0|\tilde{w}, h)]$$

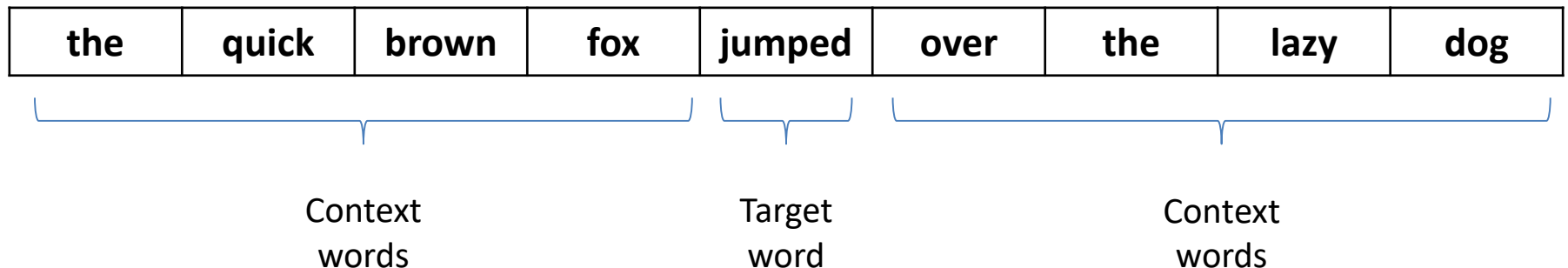
- $Q_{\theta}(D = 1|w_t, h)$ is the binary logistic regression probability under the model of seeing the word w_t in the context h in the dataset D , calculated in terms of the learned embedding vectors θ .
- In practice we approximate the expectation by drawing k contrastive words from the noise distribution.

Word2vec

- Model training
 - Step 1: Prepare dataset
 - Step 2: Compute word embedding by learned embedding matrix
 - Step 3: Compute loss by nce-loss
 - Step 4: Train the word2vec model by gradient-descent

Word2vec

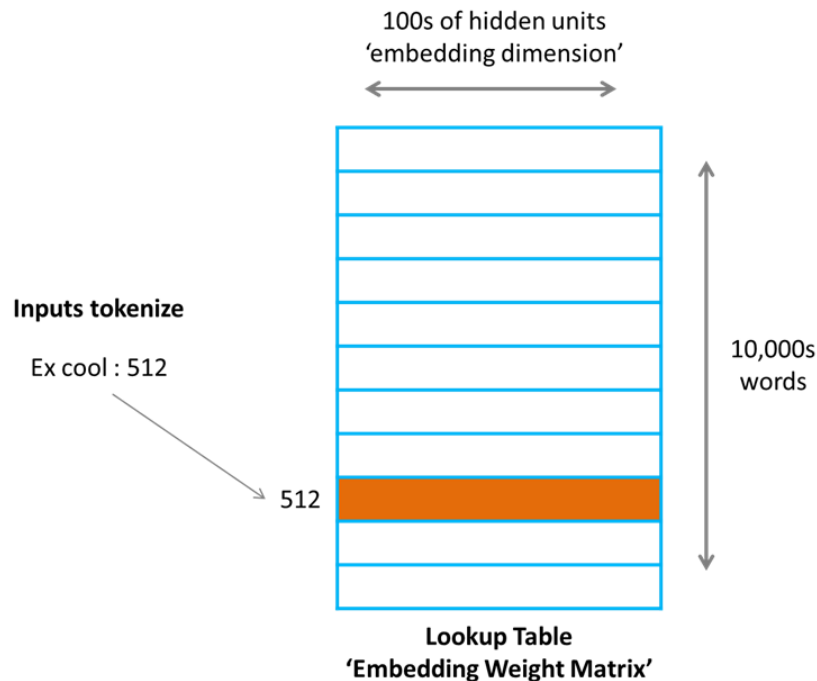
- Model training
 - Step 1: Prepare dataset
 - Let's prepare the dataset for skip-gram model
 - Given a sentence,



- Pair target word with context words randomly. The dataset becomes, (jumped, fox), (jumped, the), (jumped, fox)...

Word2vec

- Model training
 - Step 2: Compute word embedding by learned embedding matrix
 - Turn tokenize inputs to embeddings
 - The shape of embedding matrix should be [vocabulary size, embedding size]



Word2vec

- Model training
 - Step 2: Compute word embedding by learned embedding matrix
 - Define embedding_lookup layer by Layer subclassing.

```
from tensorflow.keras.layers import Layer

class embedding_lookup(Layer): ← Inherits from Keras layer
    def __init__(self):
        super(embedding_lookup, self).__init__()
        embedding_init = tf.keras.initializers.GlorotUniform()
        self.embedding_matrix = self.add_weight(name="embedding_matrix",
                                                trainable=True,
                                                shape=[vocabulary_size, embed_size],
                                                initializer=embedding_init)

    def call(self, inputs): ← Tokenized inputs
        center_words = inputs
        embedding = tf.nn.embedding_lookup(self.embedding_matrix,
                                           center_words,
                                           name='embedding')

    return embedding ← Return word embeddings
```

Word2vec

- Model training
 - Step 3: Compute loss by nce-loss
 - Define nce_loss layer by Layer subclassing.

```
class nce_loss(Layer):
    def __init__(self):
        super(nce_loss, self).__init__()
        nce_w_init = tf.keras.initializers.TruncatedNormal(stddev=1.0/(embed_size ** 0.5))
        self.nce_weight = self.add_weight(name='nce_weight',
                                          trainable=True,
                                          shape=[vocabulary_size, embed_size],
                                          initializer=nce_w_init)

        self.nce_bias = self.add_weight(name='nce_bias',
                                         trainable=True,
                                         shape=[vocabulary_size],
                                         initializer=tf.keras.initializers.Zeros)

    def call(self, inputs):
        embedding, target_words = inputs[0], inputs[1]
        loss = tf.reduce_mean(tf.nn.nce_loss(weights=self.nce_weight,
                                             biases=self.nce_bias,
                                             labels=target_words,
                                             inputs=embedding,
                                             num_sampled=num_sampled,
                                             num_classes=vocabulary_size),
                              name='loss')

        return loss
```

Compute nce-loss by
built-in function 😊



Word2vec

- Model training
 - Step 4: Train model by gradient-descent
 - Define model by functional API.
 - The Functional API is a way to create models that is as flexible as model subclassing.

```
from tensorflow.keras import Model, Input

center_words = Input(shape=(), name='center_words', dtype='int32')
target_words = Input(shape=(1), name='target_words', dtype='int32')

embedding = embedding_lookup()(center_words)
loss = nce_loss()(embedding, target_words)

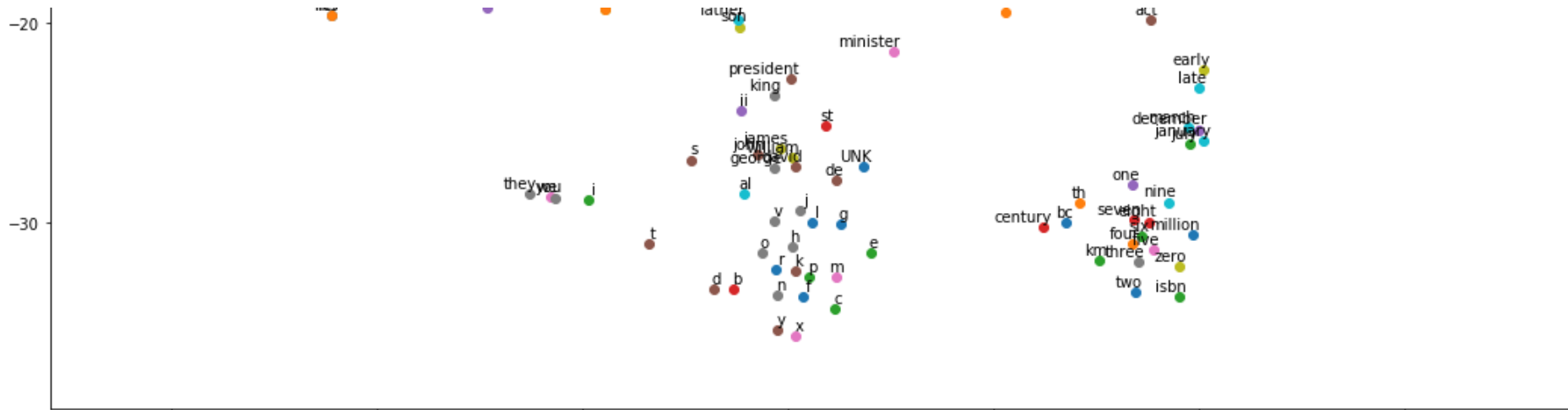
word2vec = Model(name='word2vec',
                  inputs=[center_words, target_words],
                  outputs=[loss])
```

Define the logics of inputs and outputs between layers

Use inputs and outputs to build model

Word2vec

- Visualize the learned embeddings by t-SNE
 - [t-SNE](#) is a machine learning algorithm which is often used to visualize high-level representations learned by artificial neural network.



Word2vec

- Cosine similarity

- Cosine similarity is a metrics for evaluating the similarity between two vectors.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

- The resulting similarity ranges from -1 meaning exactly opposite, to 1 meaning exactly the same, with 0 indicating orthogonality or decorrelation.

Word2vec

- Assignment requirements:
 1. Devise a word2vec model by model subclassing.
 - Layer subclassing is not allowed.
 2. Train your word2Vec model and plot your learning curve.
 3. Visualize your embedding matrix by t-SNE.
 4. Show top-5 nearest neighbors of "beautiful" and "people."
 5. Submit to iLMS with your **ipynb** (Lab11_{student_id}.ipynb).
- Due time:
 - 2019-10-31(Thur) 23:59