TensorFlow

Курс “Практическое применение по TensorFlow”

Шигапова Фирюза Зинатуллаевна

1-й семестр, 2019 г.

https://github.com/Firyuza/TensorFlowPractice

Override Backward-Propagation

- Model contains **non-differentiable** operations
- You must define own custom flow in backprop
- You must define own custom forward flow

Override Backward-Propagation. Model

```
stddev = 1e-1
labels = [0, 1]
data = [1, 2]

labels_ph = tf.placeholder(shape=(2), dtype=tf.int32)
data_ph = tf.placeholder(shape=(2), dtype=tf.float32)

c = tf.Variable(tf.truncated_normal(shape=[2], stddev=stddev))
k = tf.Variable(tf.truncated_normal(shape=[2], stddev=stddev))

d = tf.add(data_ph, c)
e = tf.add(c, k)
a = tf.multiply(d, e)

loss = tf.losses.softmax_cross_entropy(logits=a, onehot_labels=labels_ph)
opt = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=0.1)
```

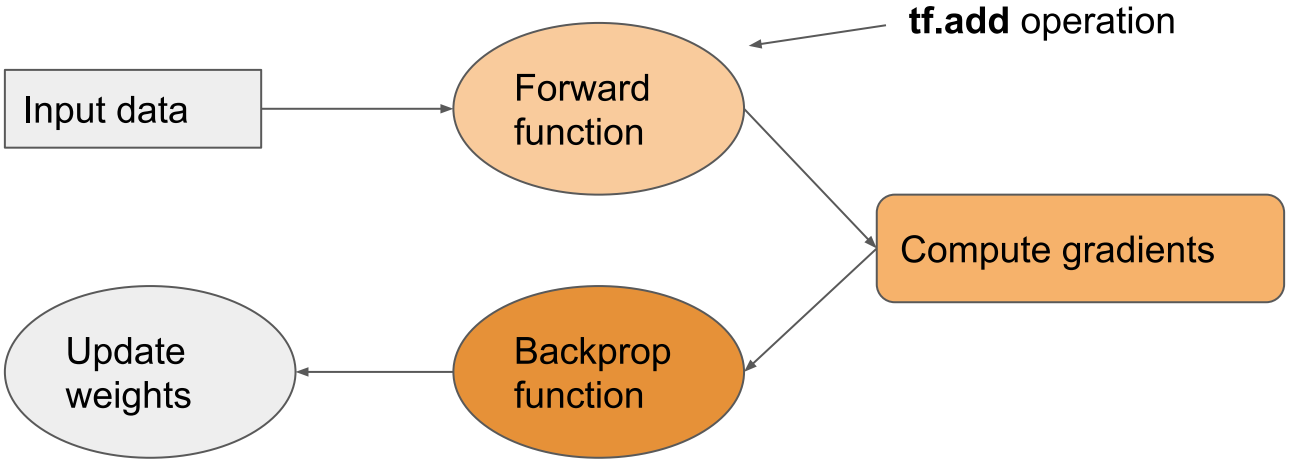
Override Backward-Propagation. Model

```
stddev = 1e-1
labels = [0, 1]
data = [1, 2]

labels_ph = tf.placeholder(shape=(2), dtype=tf.int32)
data_ph = tf.placeholder(shape=(2), dtype=tf.float32)

c = tf.Variable(tf.truncated_normal(shape=[2], stddev=stddev))
k = tf.Variable(tf.truncated_normal(shape=[2], stddev=stddev))

d = tf.add(data_ph, c)
e = tf.add(c, k)
a = tf.multiply(d, e)

loss = tf.losses.softmax_cross_entropy(logits=a, onehot_labels=labels_ph)
opt = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=0.1)
```

Let’s Override backprop for this operation

Override Backward-Propagation

Write function that will replace tf.add operation:



Override Backward-Propagation

Forward function

Say TensorFlow for which operation you want to override Backward-Propagation:

Use **gradient_override_map** that belongs to Graph

```
with graph.gradient_override_map({"PyFunc": rnd_name}):
    return tf.py_func(forward_func, inputs, [np.float32], name=name)
```

It is **tf.add** operation

Type of returned data from forward func

All variables that take part in **tf.add** operation

Override Backward-Propagation

Forward function

Need to pass all variables that take part NOT only in forward but in backprop too

```
def forward_func(c, k, d):
    e = np.add(c, k)

    return e.astype(np.float32)
```

Override Backward-Propagation

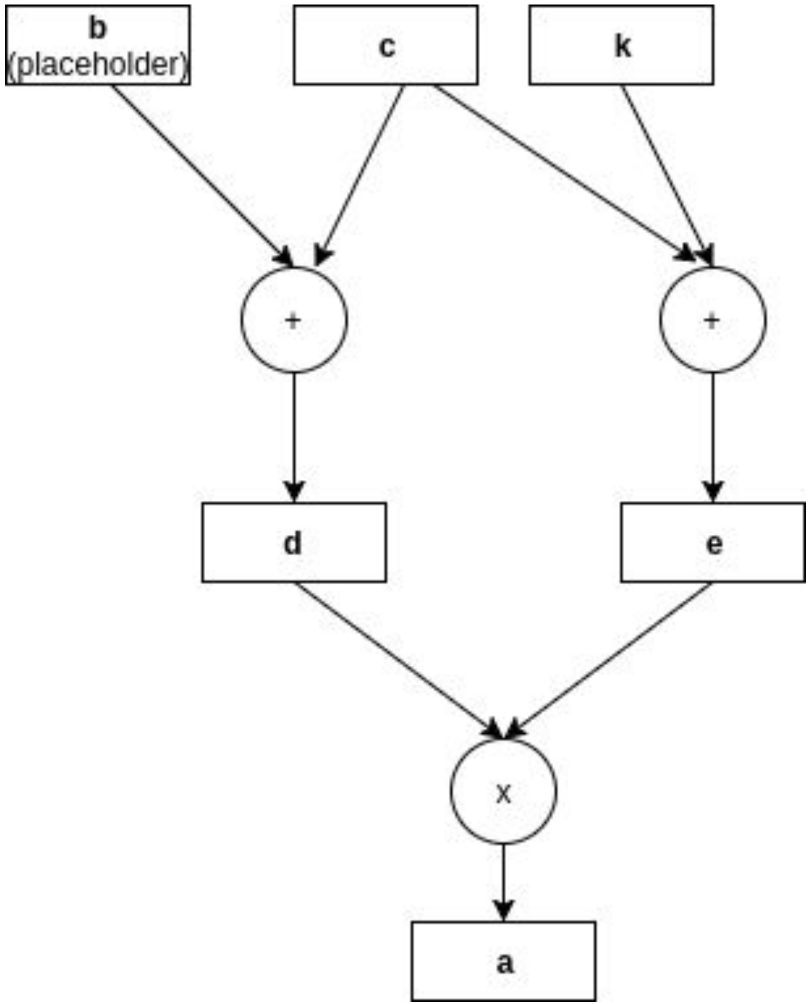
Compute gradients

```
tf.RegisterGradient(rnd_name)(backprop_func)
```

Define gradient function for an operation

Function that computes gradients in backprop flow

Graph



Backprop: chain rule

$$\frac{\partial L}{\partial c} = \frac{\partial d}{\partial c} \cdot \frac{\partial L}{\partial d} + \frac{\partial e}{\partial c} \cdot \frac{\partial L}{\partial e} = e \cdot \frac{\partial L}{\partial a} + d \cdot \frac{\partial L}{\partial a} = (e + d) \cdot \frac{\partial L}{\partial a},$$

where $\frac{\partial L}{\partial d} = \frac{\partial a}{\partial d} \cdot \frac{\partial L}{\partial a} = e \cdot \frac{\partial L}{\partial a}$ and $\frac{\partial L}{\partial e} = \frac{\partial a}{\partial e} \cdot \frac{\partial L}{\partial a} = d \cdot \frac{\partial L}{\partial a}$

$$\frac{\partial L}{\partial k} = \frac{\partial e}{\partial k} \cdot \frac{\partial L}{\partial e} = d \cdot \frac{\partial L}{\partial a}$$

Note: the gradient $\frac{\partial L}{\partial a}$ is already calculated, we get it in backprop, in my example this variable called "`grad`" in `backprop_func`.

Backprop function

Override Backward-Propagation

```
def backprop_func(op, grad):
    c = op.inputs[0]
    k = op.inputs[1]
    d = op.inputs[2]
    e = tf.add(c, k)
    return (e + d) * grad , d * grad, e * grad
```

Compute partial derivative for each passed variable

- **op** contains all passed variables;
- **grad** — the flown gradient from the back propagation.

Override Backward-Propagation

```python
def custom_add(c, k, d, graph, name=None):
    with tf.name_scope(name, "AddGrad", [c, k, d]) as name:
        # Need to generate a unique name to avoid duplicates
        # if you have more than one custom gradients:
        rnd_name = 'PyFuncGrad' + str(np.random.randint(0, 1e+8))

        inputs = [c, k, d]

        tf.RegisterGradient(rnd_name)(backprop_func)
        with graph.gradient_override_map({"PyFunc": rnd_name}):
            return tf.py_func(forward_func, inputs, [np.float32], name=name)
```

Override Backward-Propagation

Explicitly call **compute_gradients** and apply to optimizer!!!

```
grads = opt.compute_gradients(loss, tf.trainable_variables())
grads = list(grads)
train_op = opt.apply_gradients(grads_and_vars=grads)
```

Override Backward-Propagation

```
stddev = 1e-1
labels = [0, 1]
data = [1, 2]

labels_ph = tf.placeholder(shape=(2), dtype=tf.int32)
data_ph = tf.placeholder(shape=(2), dtype=tf.float32)

c = tf.Variable(tf.truncated_normal(shape=[2], stddev=stddev))
k = tf.Variable(tf.truncated_normal(shape=[2], stddev=stddev))

d = tf.add(data_ph, c)
e = custom_add(c, k, d, tf.get_default_graph()) # tf.add(c, k)
a = tf.multiply(d, e)

loss = tf.losses.softmax_cross_entropy(logits=a, onehot_labels=labels_ph)
opt = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=0.1)

grads = opt.compute_gradients(loss, tf.trainable_variables())
grads = list(grads)
train_op = opt.apply_gradients(grads_and_vars=grads)
```

tf.stop_gradient

Stop to flow backprop flow further for the given operation

```
tf.stop_gradient(
    input,
    name=None
)
```

Clothing Retrieval with Visual Attention Model

Zhonghao Wang [1], Yujun Gu [1], Ya Zhang [1✉], Jun Zhou [2], Xiao Gu [2]

[1] *Cooperative Medianet Innovation Center, Shanghai Jiao Tong University, Shanghai, China*
[2] *Institute of Image Communication and Network Engineering, Shanghai Jiao Tong University, Shanghai, China*
{kindredcain,yjgu,ya_zhang,zhoujun,gugu97}@sjtu.edu.cn

They describe attention network that generates **Bernoulli** series has to be multiplied with another feature map.

Bernoulli is **not differentiable** => custom backpropagation

https://arxiv.org/pdf/1710.11446.pdf

Control dependencies

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Source: https://arxiv.org/pdf/1502.03167.pdf

Control dependencies

These ops **will not** be evaluated during one train step

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Source: https://arxiv.org/pdf/1502.03167.pdf

Control dependencies

```
assign_mean = self.mean.assign(mean)
assign_variance = self.variance.assign(variance)
```

- Need to say evaluate it **First** and then train step.
- These ops are not explicitly called during train step evaluation.
- These nodes are independent in Graph.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Source: https://arxiv.org/pdf/1502.03167.pdf

Control dependencies

contains these UPDATE operations:
assign_mean, assign_variance

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(extra_update_ops):
    train_op = optimizer.minimize(loss)
```

OR

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
session.run([train_op, extra_update_ops], ...)
```

Control dependencies. What will be printed?

Only zeros.
Operation is ***not*** created within Control Dependency context!

```
x = tf.Variable(0.0)
x_plus_1 = tf.assign_add(x, 1)

with tf.control_dependencies([x_plus_1]):
    y = x
init = tf.initialize_all_variables()

with tf.Session() as session:
    init.run()
    for i in range(5):
        print(session.run(y))
```

Control dependencies. Solution

```
x = tf.Variable(0.0)
x_plus_1 = tf.assign_add(x, 1)

with tf.control_dependencies([x_plus_1]):
    y = tf.identity(x)
init = tf.initialize_all_variables()

with tf.Session() as session:
    init.run()
    for i in range(5):
        print(session.run(y))
```

OR

```
x = tf.Variable(0.0)
x_plus_1 = tf.assign_add(x, 1)

with tf.control_dependencies([x_plus_1]):
    y = x_plus_1
init = tf.initialize_all_variables()

with tf.Session() as session:
    init.run()
    for i in range(5):
        print(session.run(y))
```

Operations by themselves

For example Center Loss: “A Discriminative Feature Learning Approach for Deep Face Recognition”

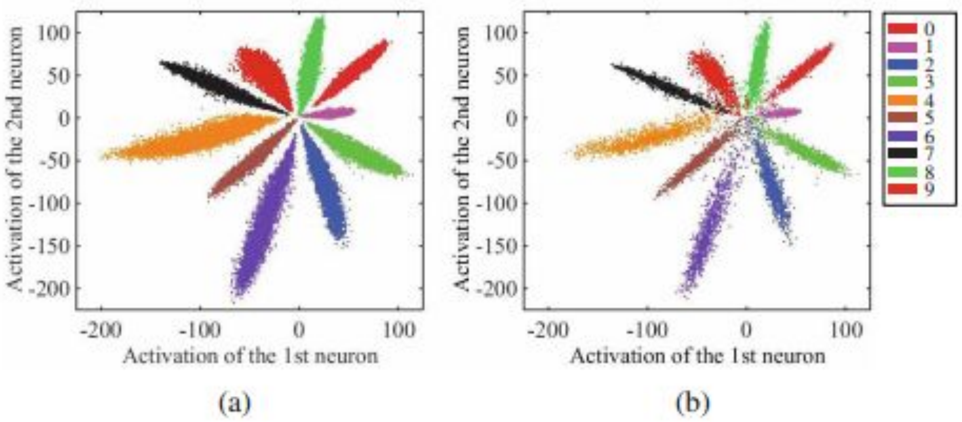


Fig. 2. The distribution of deeply learned features in (a) training set (b) testing set, both under the supervision of softmax loss, where we use 50K/10K train/test splits. The points with different colors denote features from different classes. **Best viewed in color.** (Color figure online)

Operations by themselves

For example, Center Loss: “A Discriminative Feature Learning Approach for Deep Face Recognition”

$$\mathcal{L}_C = \frac{1}{2}\sum_{i=1}^{m}\|\boldsymbol{x}_i - \boldsymbol{c}_{y_i}\|_2^2$$

The $\boldsymbol{c}_{y_i} \in \mathbb{R}^d$ denotes the y_ith class center of deep features.

$$\Delta \boldsymbol{c}_j = \frac{\sum_{i=1}^{m}\delta(y_i = j)\cdot(\boldsymbol{c}_j - \boldsymbol{x}_i)}{1+\sum_{i=1}^{m}\delta(y_i = j)} \qquad \boldsymbol{c}_j^{t+1} = \boldsymbol{c}_j^t - \alpha\cdot\Delta\boldsymbol{c}_j^t$$

Centers update is **independent** operation.

For updating centers values need to **explicitly** evaluate this op!

Source: https://ydwen.github.io/papers/WenECCV16.pdf

Operations by themselves

For example Center Loss: “A Discriminative Feature Learning Approach for Deep Face Recognition”

```
centers = tf.get_variable(name_centers, [nrof_classes, nrof_features], dtype=tf.float32,
    initializer=tf.constant_initializer(0), trainable=False)
```

```
self.update_centers_op = tf.scatter_sub(centers, label, diff)
```

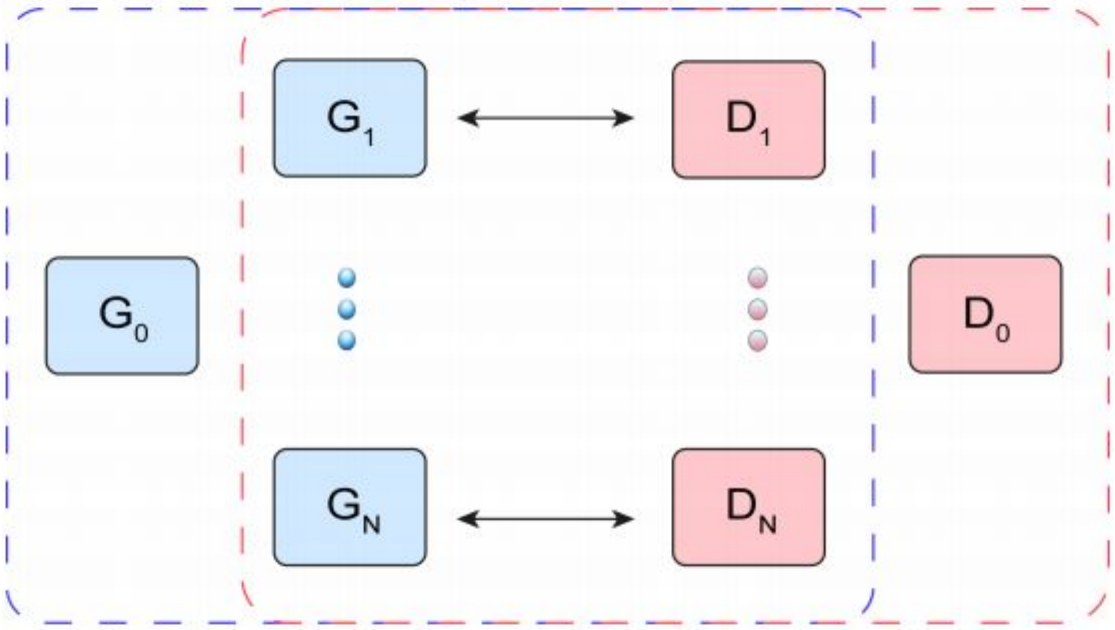
```
self.sess.run(
    [self.train_op, self.update_centers_op],
    feed_dict=feed_dict)
```

Evaluate **update_centers_op** with **train_op**

Source: https://arxiv.org/pdf/1707.07391.pdf

Train several networks independently

For example, Several GAN: “SGAN: An Alternative Training of Generative Adversarial Networks”.



Source: http://openaccess.thecvf.com/content_cvpr_2018/papers/Chavdarova_SGAN_An_Alternative_CVPR_2018_paper.pdf

Train several networks independently

1. Create Graph for each network separately.

Train several networks that are defined in the **same** Graph is **impossible**.

2. Copy values of one network to another:

```
vars_1 = [var for var in train_vars1
          if '[needed scope name]' in var.name]


with self.session_2.as_default():
    vars_2 = [var for var in train_vars2
              if '[needed scope name]' in var.name]
    for var_idx, var in enumerate(vars_1):
       vars_2[var_idx].load(
                      vars_1[var_idx].eval(session=self.session_1),
                      self.session_2)
```

Train several networks independently

- assign
- tf.Variable
- tf.contrib.copy_graph.copy_variable_to_graph

They make **reference**, not a **deep copy.**

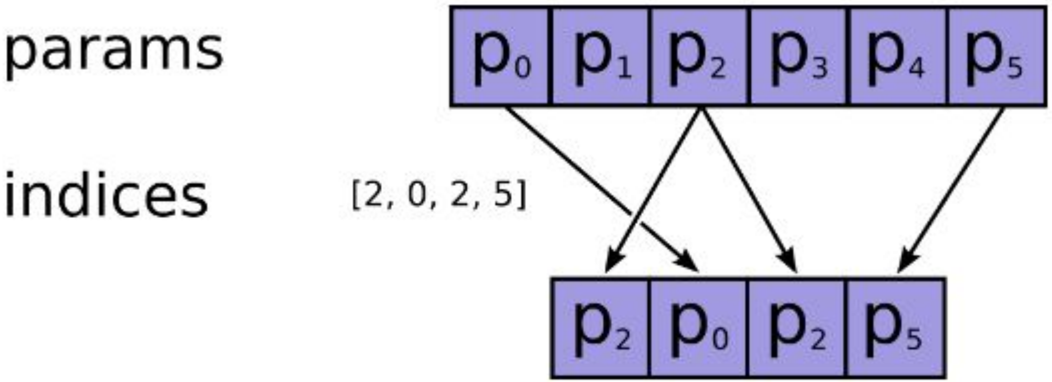
For making a deep copy of trainable variables into another one use **load.**

TensorFlow Queue

1. Always control that **queue is completely filled**.
2. Always be sure that **validation data** is **not** used as **training data** and vice versa.
3. During **dequeue** in **multithreading** way care about transformation: all random transformation has to be **RANDOM.**

tf.gather

Take data from the given params according to indices.



tf.gather

If code is running on **GPU:** indices can be out of range, **zeros** will be returned.

```
a = tf.constant([1, 2, 3, 4, 5])
b = tf.gather(a, [0, 1, 6]) # [1 2 0]
```

If code is running on **CPU**: indices **cannot** be out of range, otherwise throws **error**.

```
a = tf.constant([1, 2, 3, 4, 5])
b = tf.gather(a, [0, 1, 6])
# InvalidArgumentError: indices[2] = 6 is not in [0, 5) [Op:GatherV2]
```