intel Nervana™ AI Academy

DEVICES AND TENSORFLOW

TFRECORDS

TFRECORDS FORMAT

TensorFlow has its own file format for storing training data

Allows you to store related data together, instead of having to merge them together at run time, e.g.

- raw image data
- labels
- text
- any other metadata

One file can hold an entire training set (or a sharded chunk)

HOW TO CREATE TFRECORDS

1. Open a `tf.python_io.TFRecordWriter`

```
writer = tf.python_io.TFRecordWriter(path)
```

2. Create a `tf.train.Feature` for each value you want to save

```
# Raw image bytes feature
img = tf.train.BytesList(value=[image_bytes])
image = tf.train.Feature(bytes_list=img)
# Integer label feature
lbl = tf.train.Int64List(value=[label])
label = tf.train.Feature(int64_list=lbl)
```

HOW TO CREATE TFRECORDS

3. Create a `tf.train.Features` object that holds all features:

```
features = tf.train.Features(feature=
    {'image_bytes': image,
     'label': label}
```

4. Wrap Features inside of a `tf.train.Example` class

```
example = tf.train.Example(features=features)
```

5. Write the Example to the records file

```
writer.write(example.SerializeToString())
```

QUEUES

AN ALTERNATIVE TO FEED_DICT

So far, we've passed data to our model via a feed dictionary

Unfortunately, this isn't the fastest mechanism:

- It has to jump back and forth between the Python and C++ layers for each step
- It waits to load data into GPU until after the previous run is finished

What can we do instead?

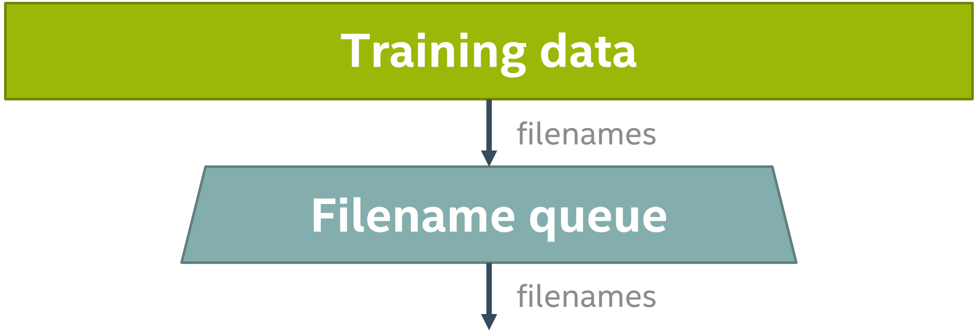
TENSORFLOW QUEUES

- TensorFlow has built-in support for asynchronous queues
- Creates batches of data in the background
- Can immediately start next batch after one batch is done
- Easiest to use with TFRecords!

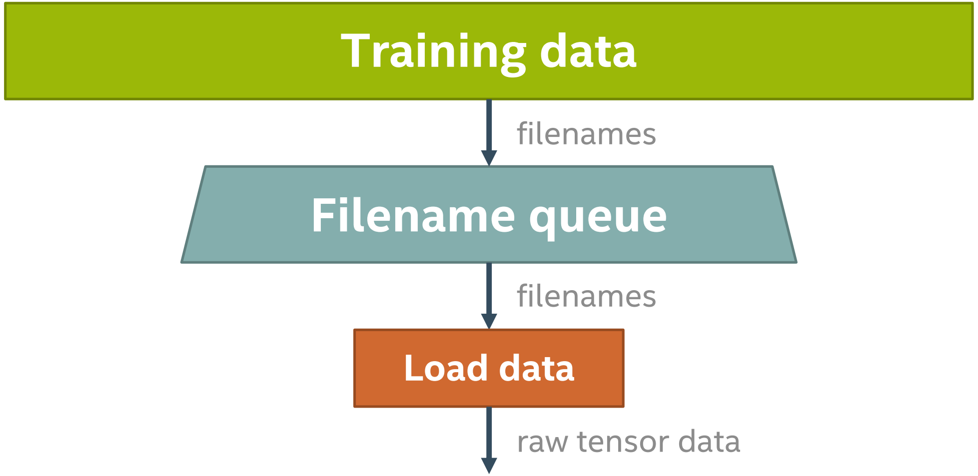
BASIC QUEUE PATTERN

BASIC QUEUE PATTERN

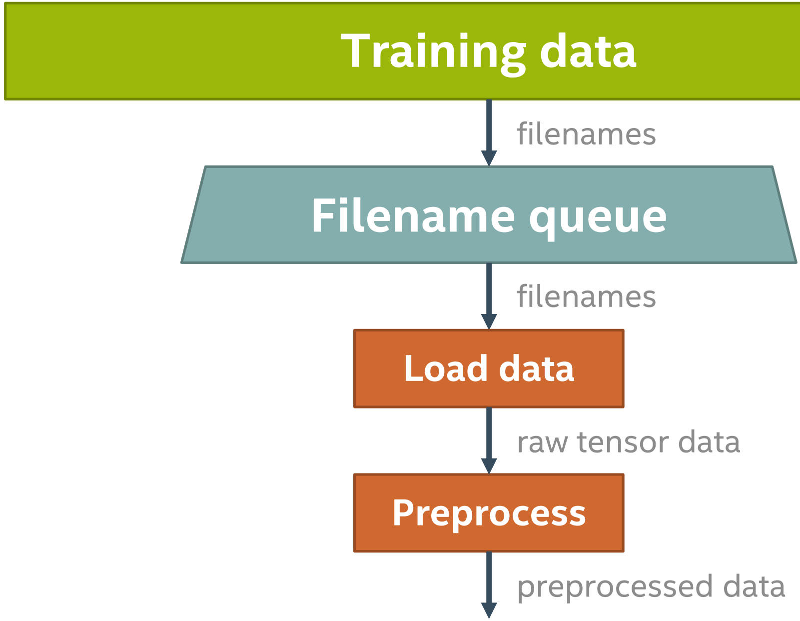
BASIC QUEUE PATTERN



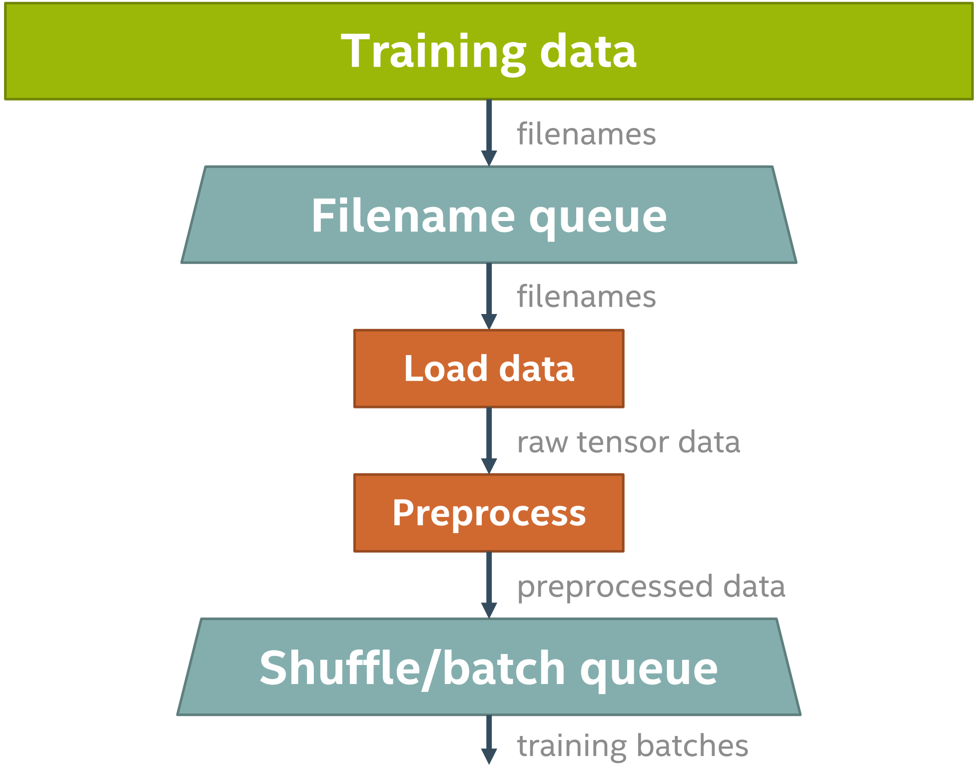
BASIC QUEUE PATTERN



BASIC QUEUE PATTERN



BASIC QUEUE PATTERN



USING THE QUEUE IN A GRAPH

Once you’ve created the queue, you end up with handles to batches of training data:

```
image_batch, label_batch = tf.train.shuffle_batch(…)
```

Use these the same way you would a placeholder!

EXTRA BITS FOR MANAGING THREADS:

Coordinator manages the pool of threads

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess)
try:
    while not coord.should_stop():
        …sess.run(..)
except tf.errors.OutOfRangeError:
    print('done!')
finally:
    coord.request_stop()
    coord.join(threads)
```

EXTRA BITS FOR MANAGING THREADS:

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess)
try:
    while not coord.should_stop():
        …sess.run(..)
except tf.errors.OutOfRangeError:
    print('done!')
finally:
    coord.request_stop()
    coord.join(threads)
```

Starts the threads created in the Graph

EXTRA BITS FOR MANAGING THREADS:

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess)
try:
    while not coord.should_stop():
        …sess.run(..)
except tf.errors.OutOfRangeError:
    print('done!')
finally:
    coord.request_stop()
    coord.join(threads)
```

Coordinator can receive signals to stop training

EXTRA BITS FOR MANAGING THREADS:

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess)
try:
    while not coord.should_stop():
        …sess.run(..)
except tf.errors.OutOfRangeError:
    print('done!')
finally:
    coord.request_stop()
    coord.join(threads)
```

We'll get an OutOfRangeError when we've gone through all data (or when we've gone through the max number of epochs)

EXTRA BITS FOR MANAGING THREADS:

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess)
try:
    while not coord.should_stop():
        …sess.run(..)
except tf.errors.OutOfRangeError:
    print('done!')
finally:
    coord.request_stop()
    coord.join(threads)
```

Requests threads to stop

EXTRA BITS FOR MANAGING THREADS:

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess)
try:
    while not coord.should_stop():
        …sess.run(..)
except tf.errors.OutOfRangeError:
    print('done!')
finally:
    coord.request_stop()
    coord.join(threads)
```

Waits for all threads to finish before continuing

NEWER STYLE WITH CONTEXT MANAGER

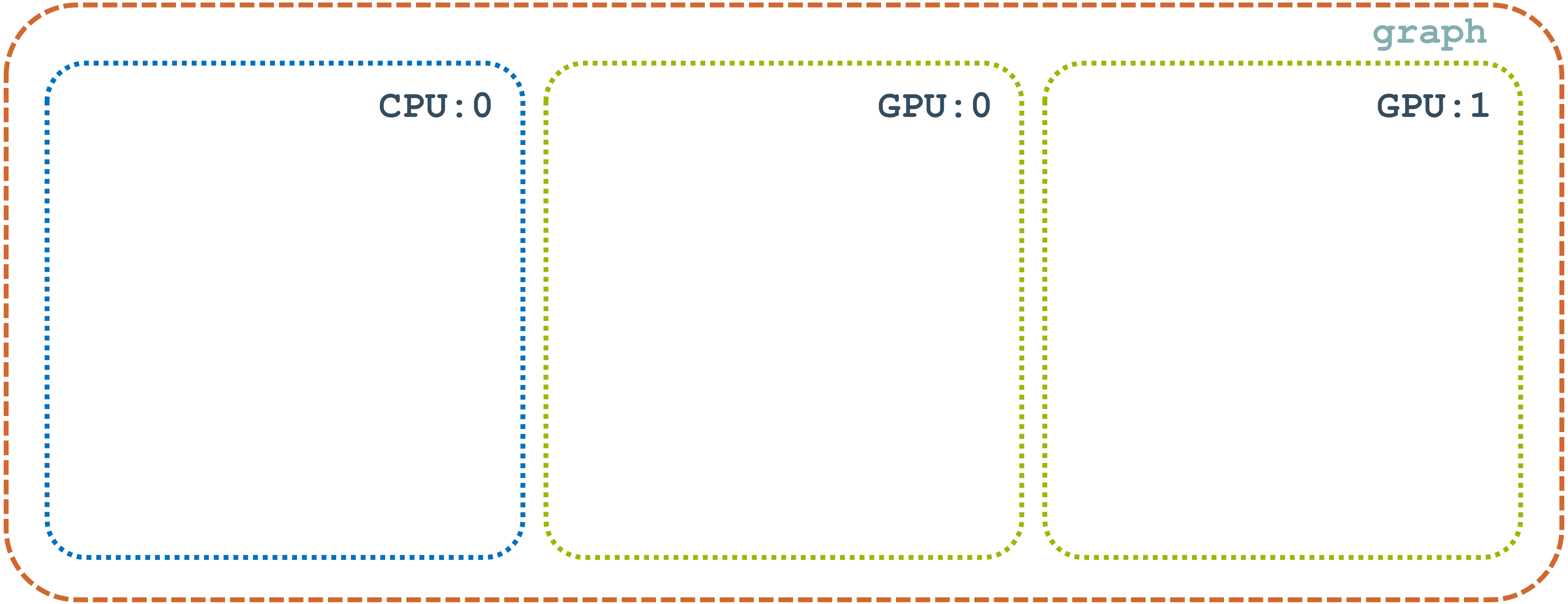
```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess)
with coord.stop_on_exception()
    while not coord.should_stop():
        …sess.run(..)
```

MULTI-GPU

GETTING MORE POWER!

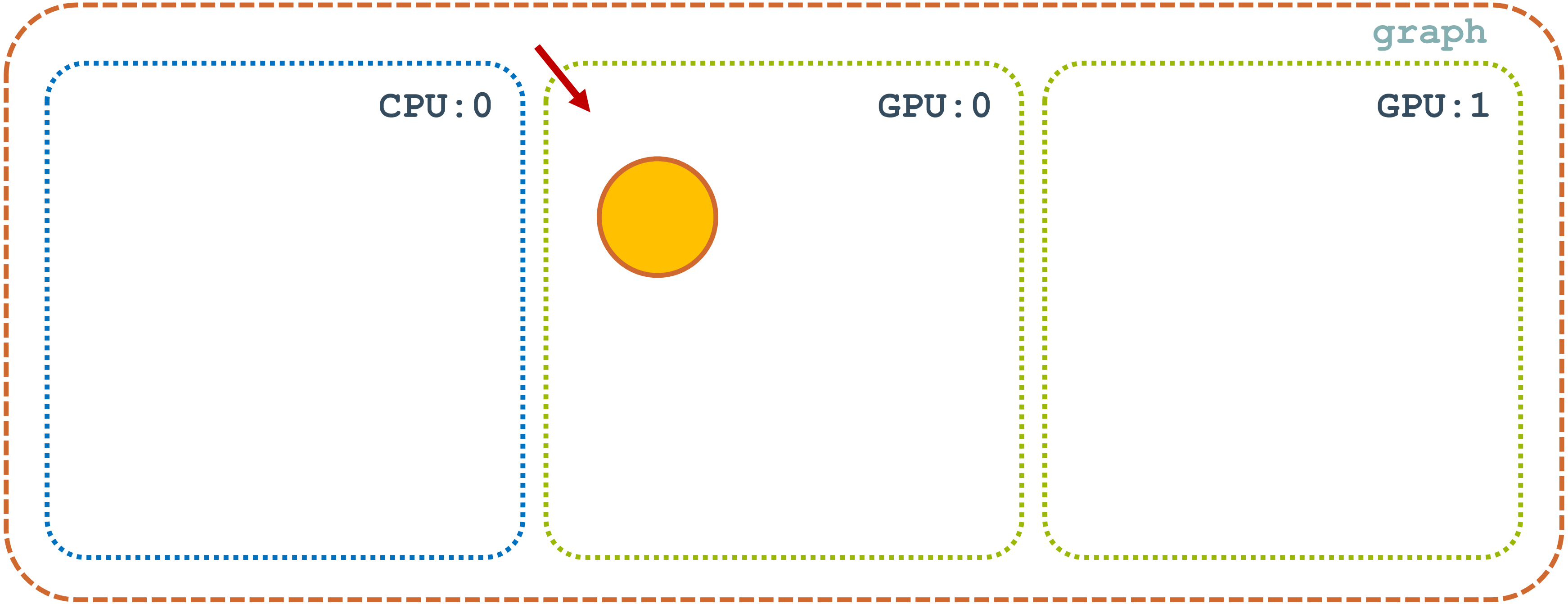
“Device” – A computational device. CPU, GPU, etc.

MULTI-DEVICE GRAPH



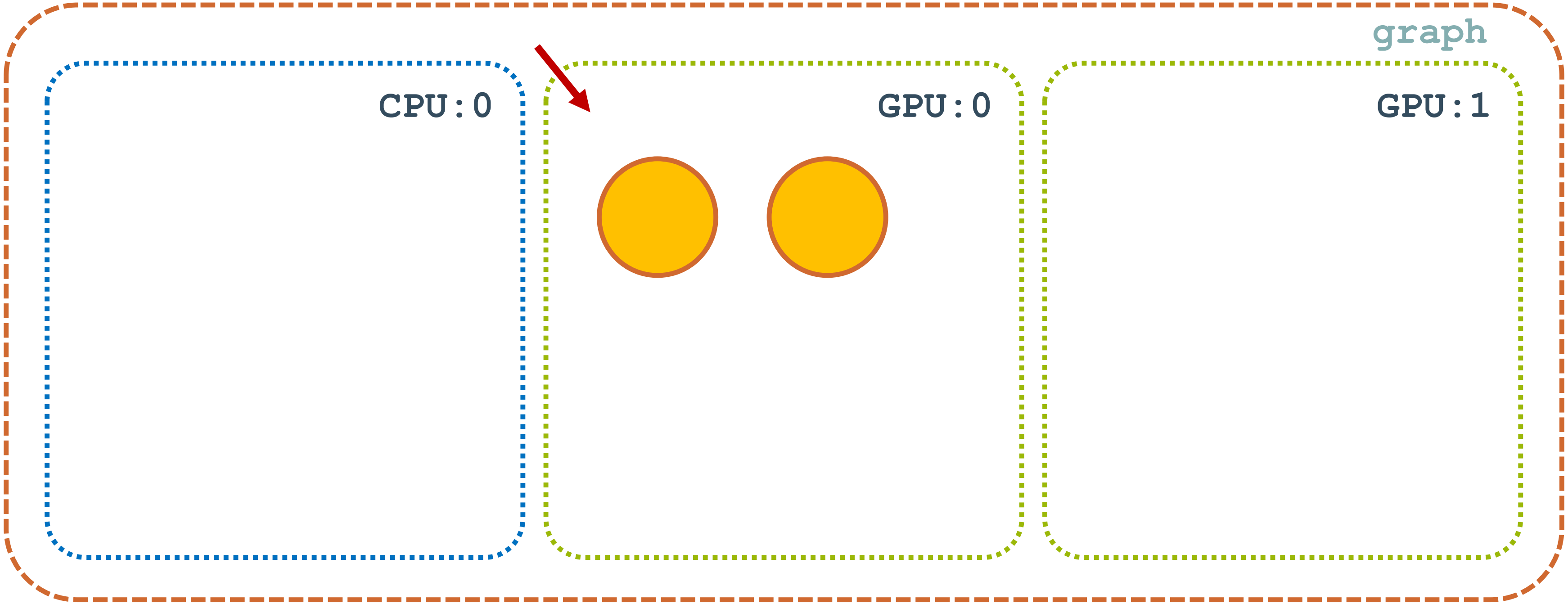
BY DEFAULT, OPERATIONS WILL BE PLACED ON GPU:0

```
>>> a = tf.placeholder(tf.float32)
```



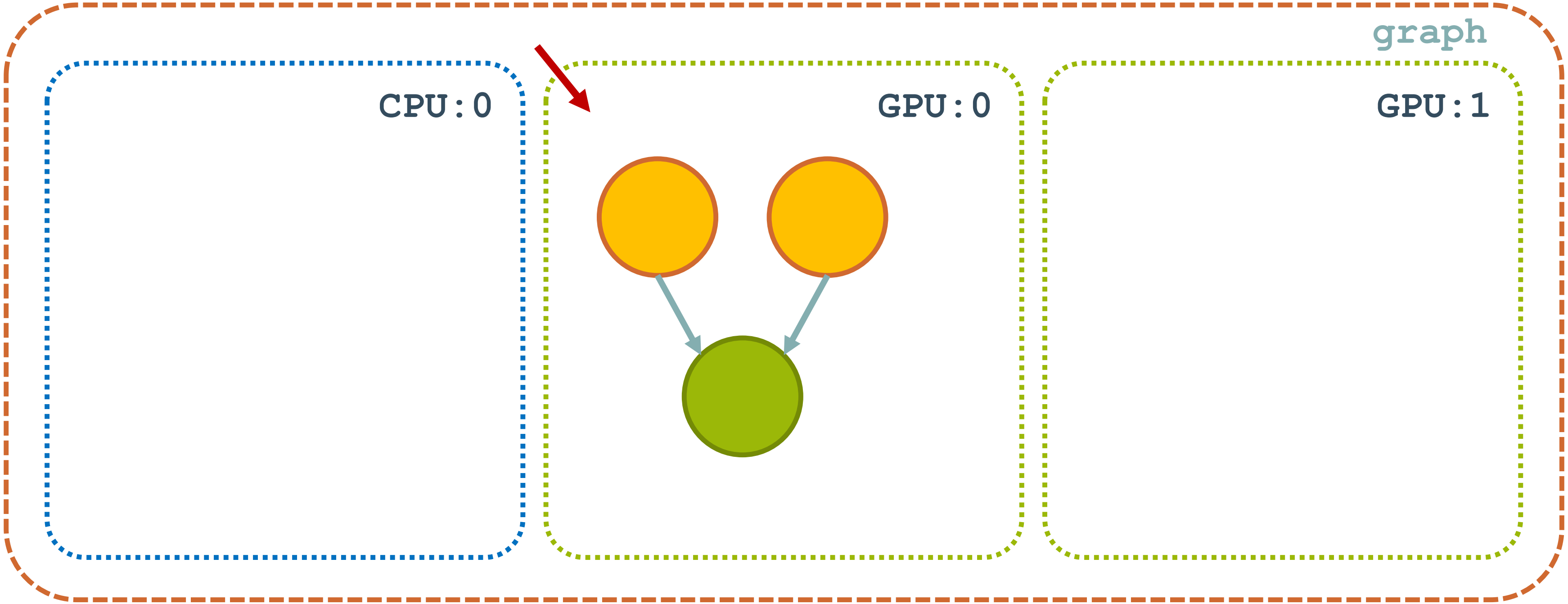
BY DEFAULT, OPERATIONS WILL BE PLACED ON GPU:0

```
>>> b = tf.placeholder(tf.float32)
```



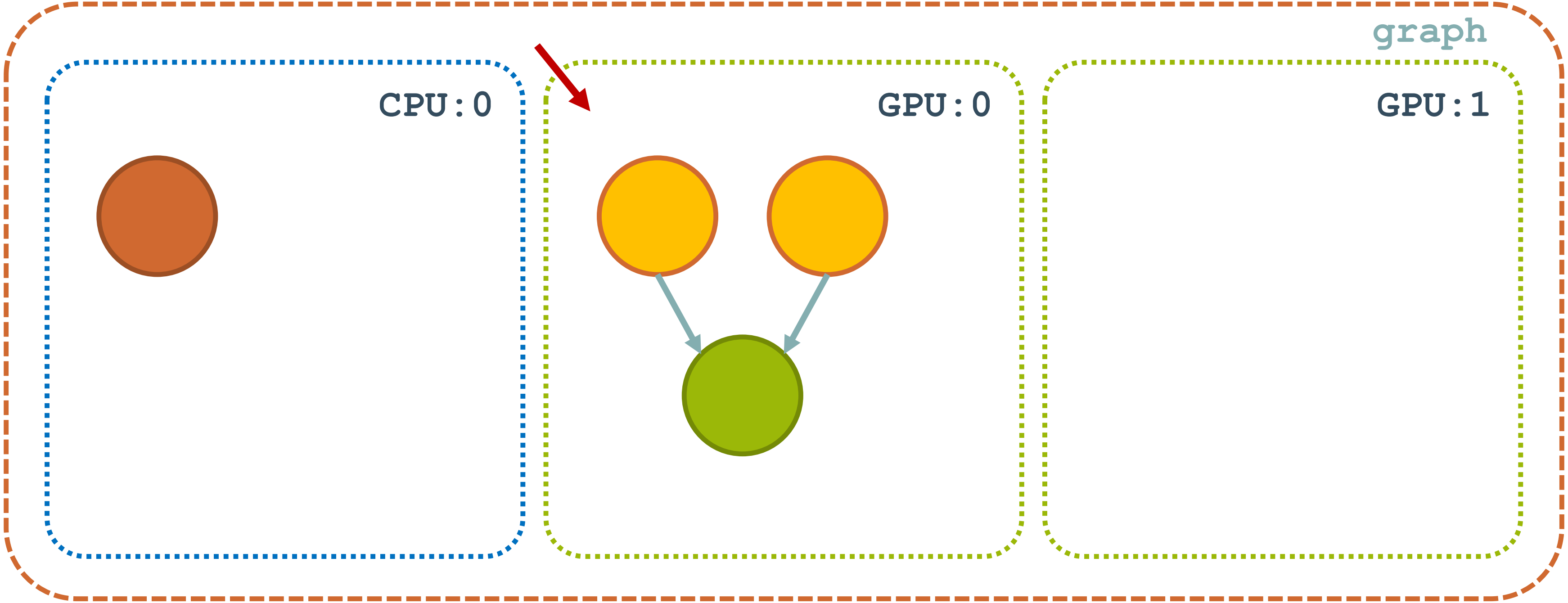
BY DEFAULT, OPERATIONS WILL BE PLACED ON GPU:0

```
>>> c = tf.matmul(a, b)
```



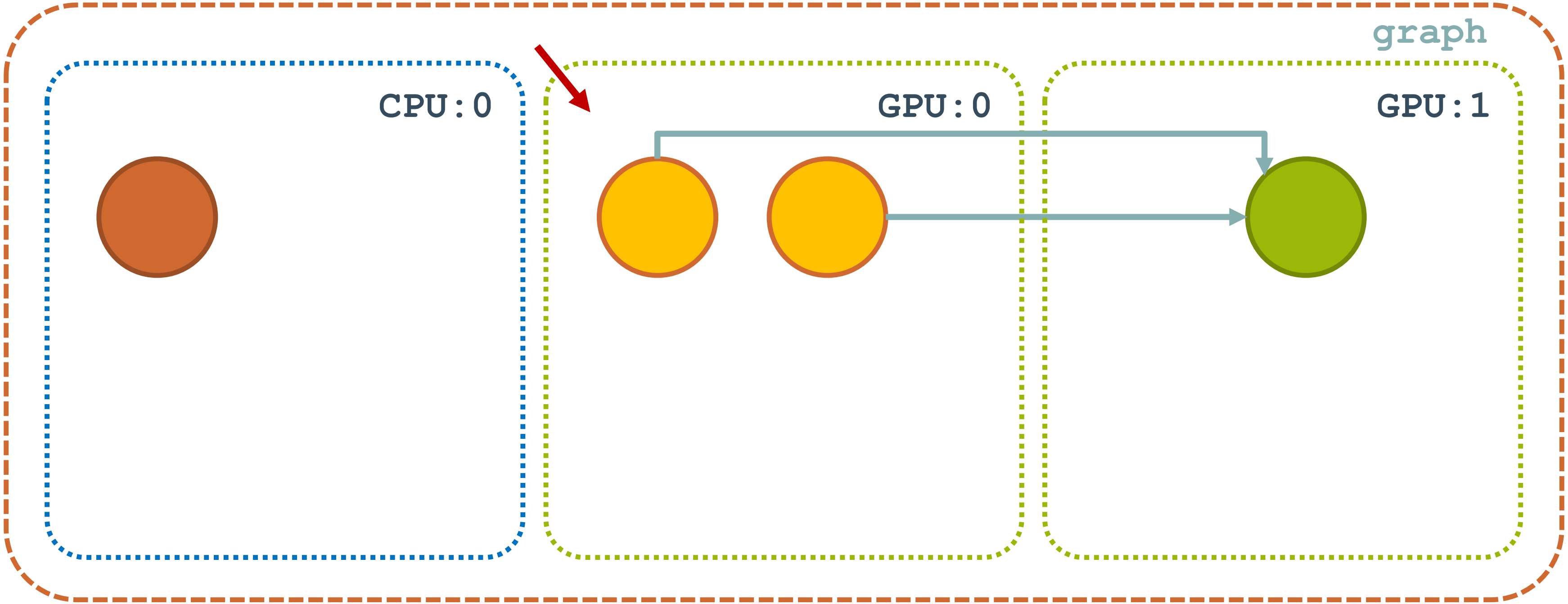
UNLESS THERE ISN'T A GPU IMPLEMENTATION FOR AN OP

```
>>> read = tf.image.decode_jpeg(image)
```



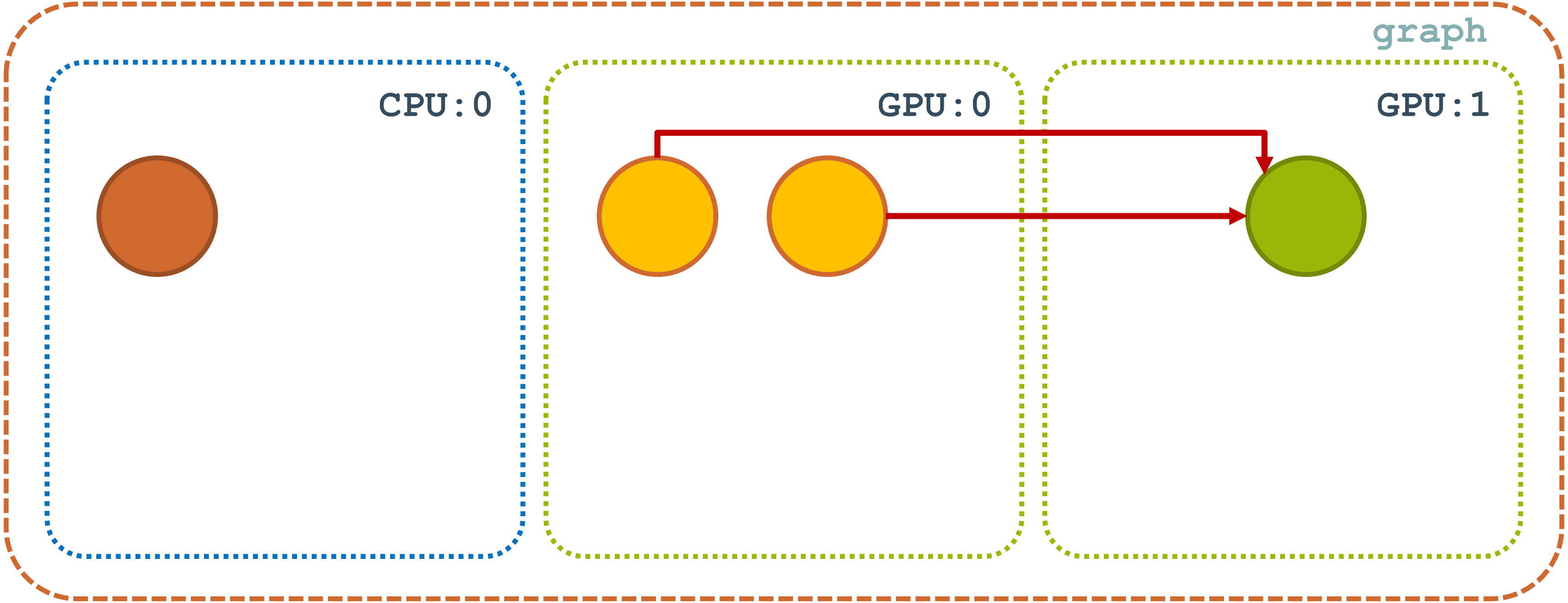
WE CAN EXPLICITLY PLACE OPS ON A DEVICE

```
>>> with tf.device('/gpu:1'):
        c = tf.matmul(a,b)
```



TENSORFLOW AUTOMATICALLY HANDLES CROSS-DEVICE DATA

>>>



TWO MAIN WAYS TO PARALLELIZE MODEL

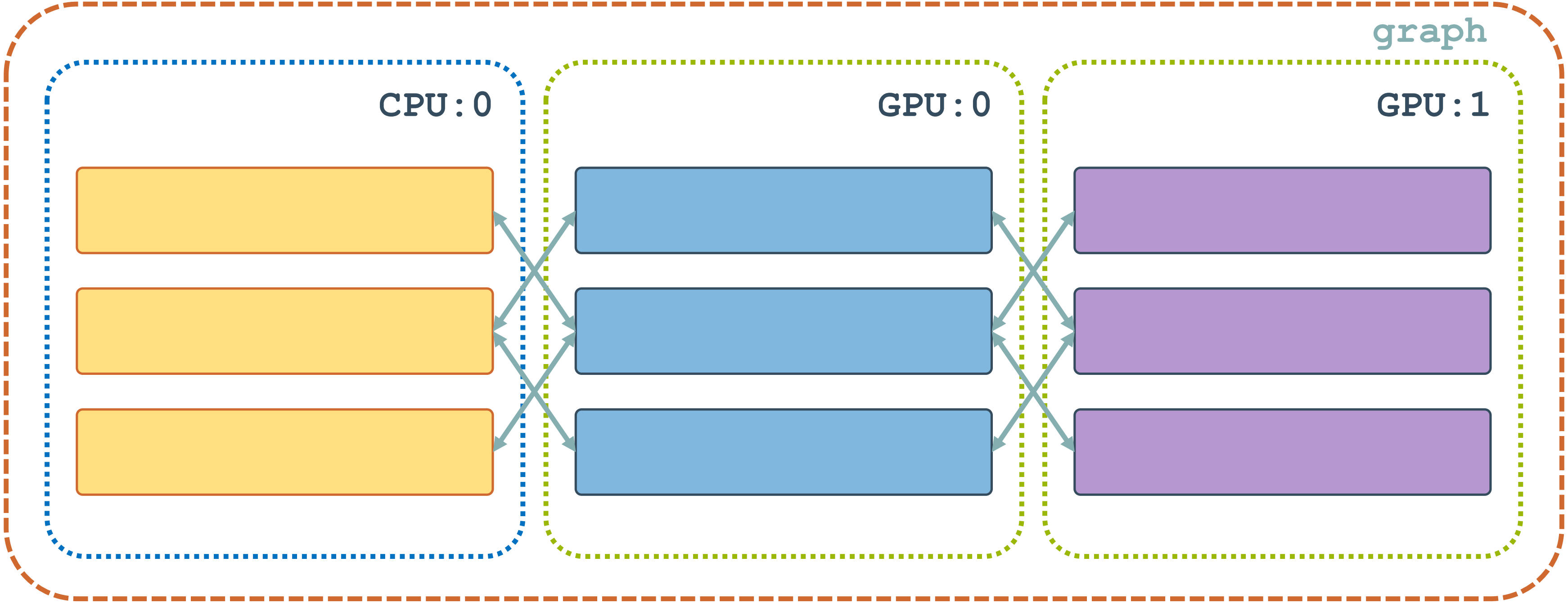
Model Parallel

- Create a huge model that spans multiple GPUs

Data parallel

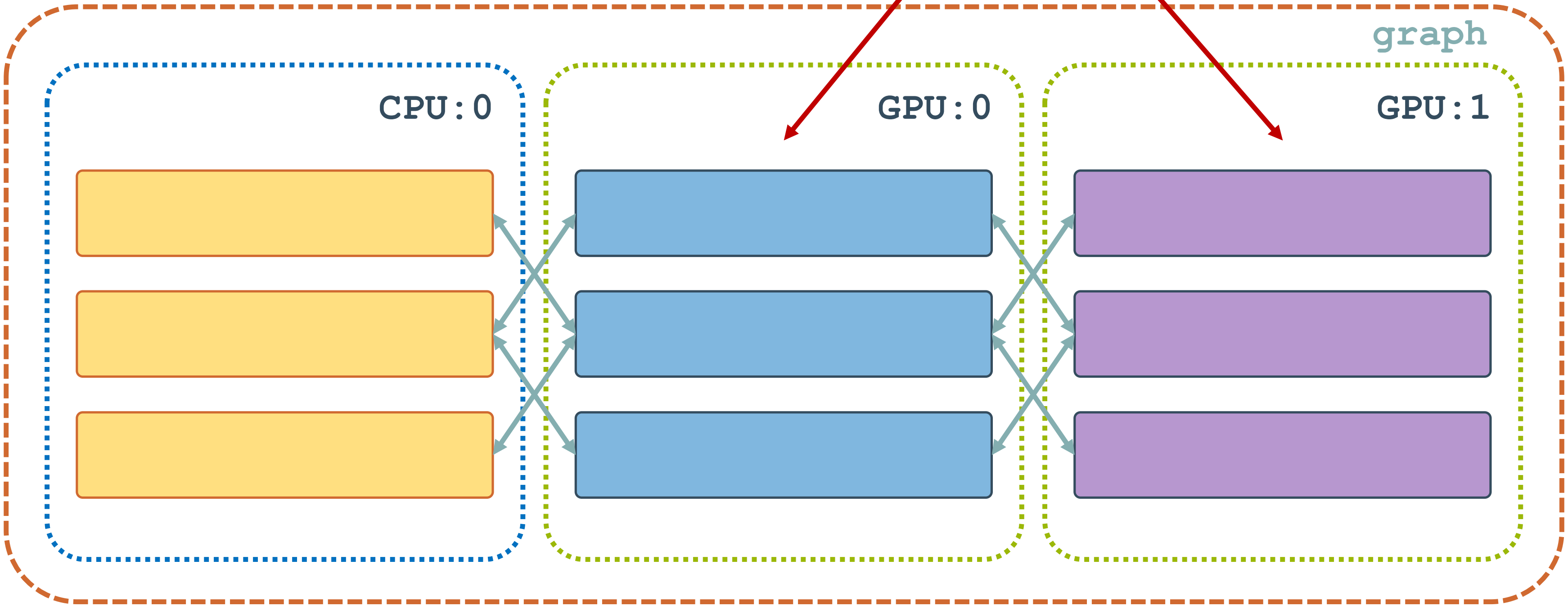
- Same model on each GPU and pass different data to each

MODEL-PARALLEL

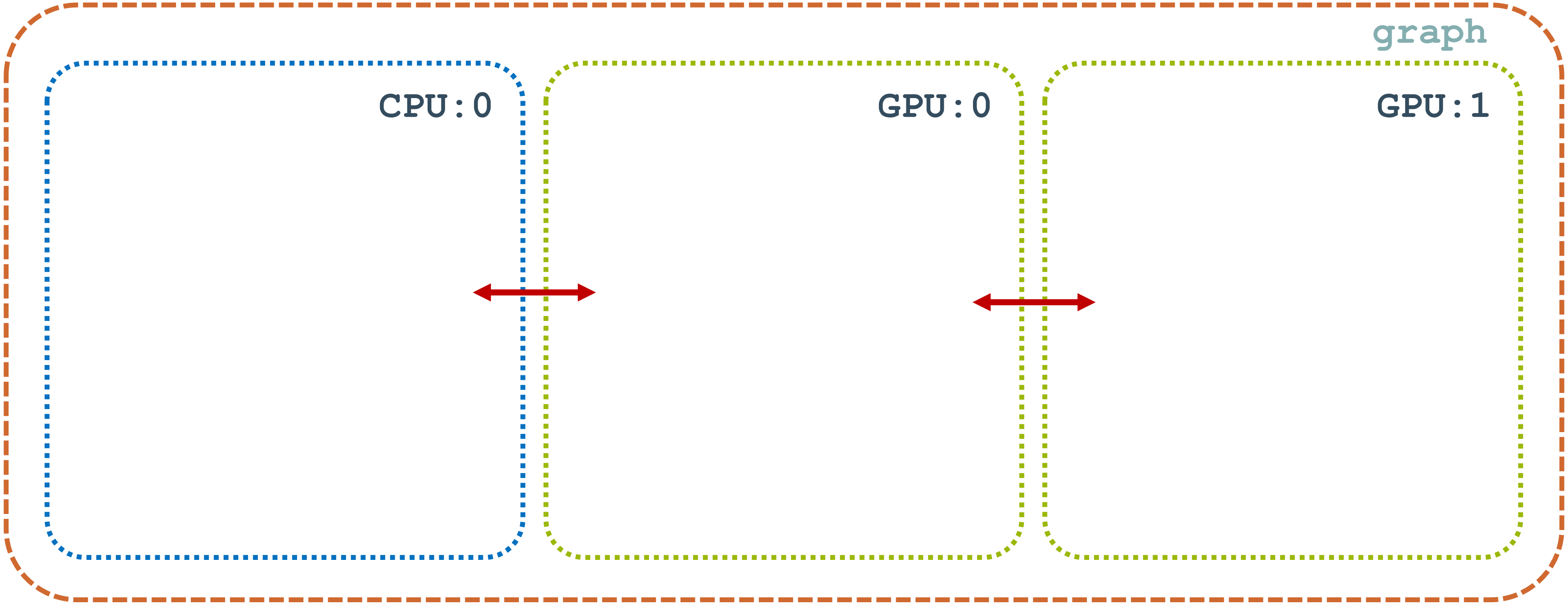


MODEL-PARALLEL

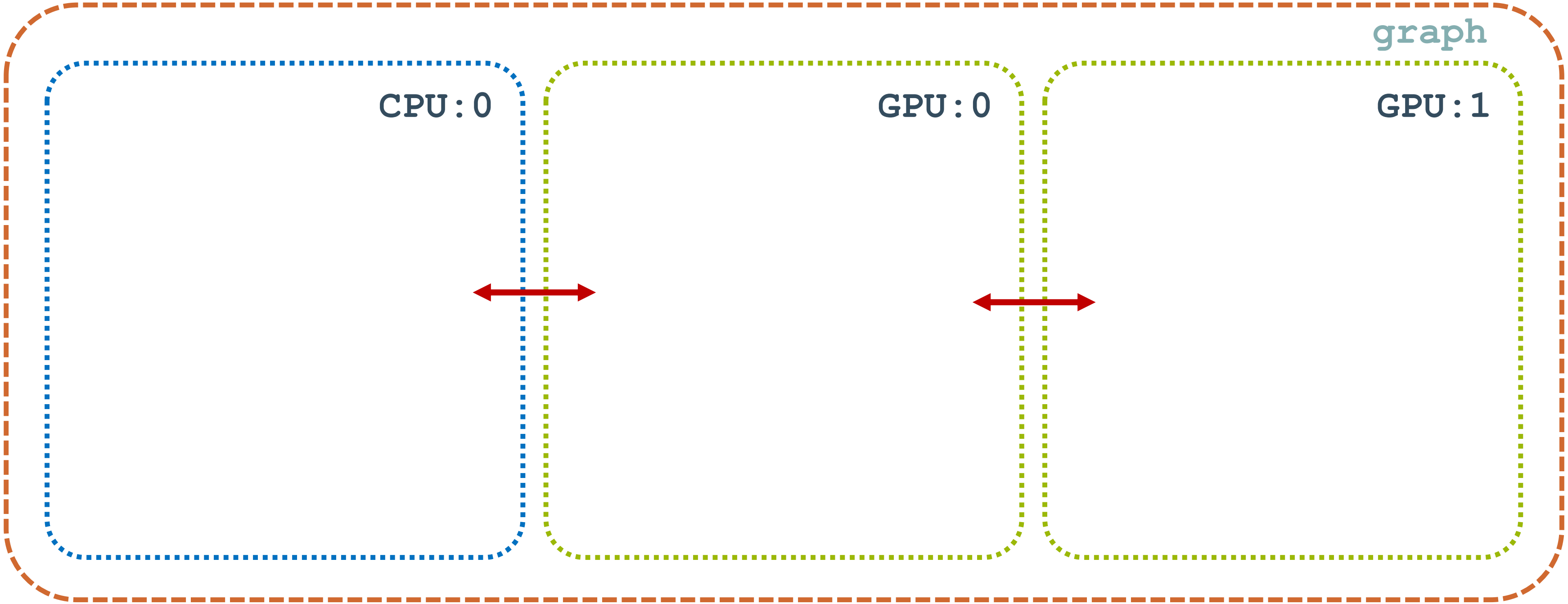
Different computations occur on different devices



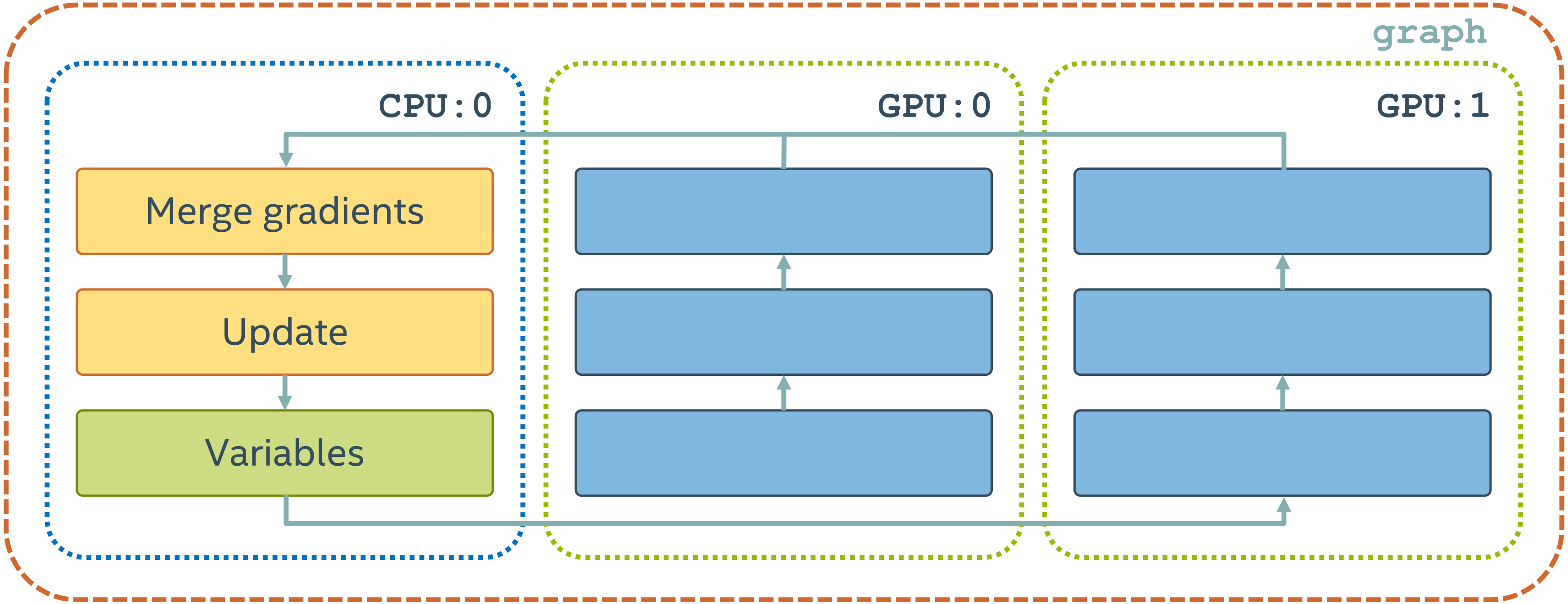
PROBLEM: COMMUNICATION ACROSS DEVICES IS SLOW



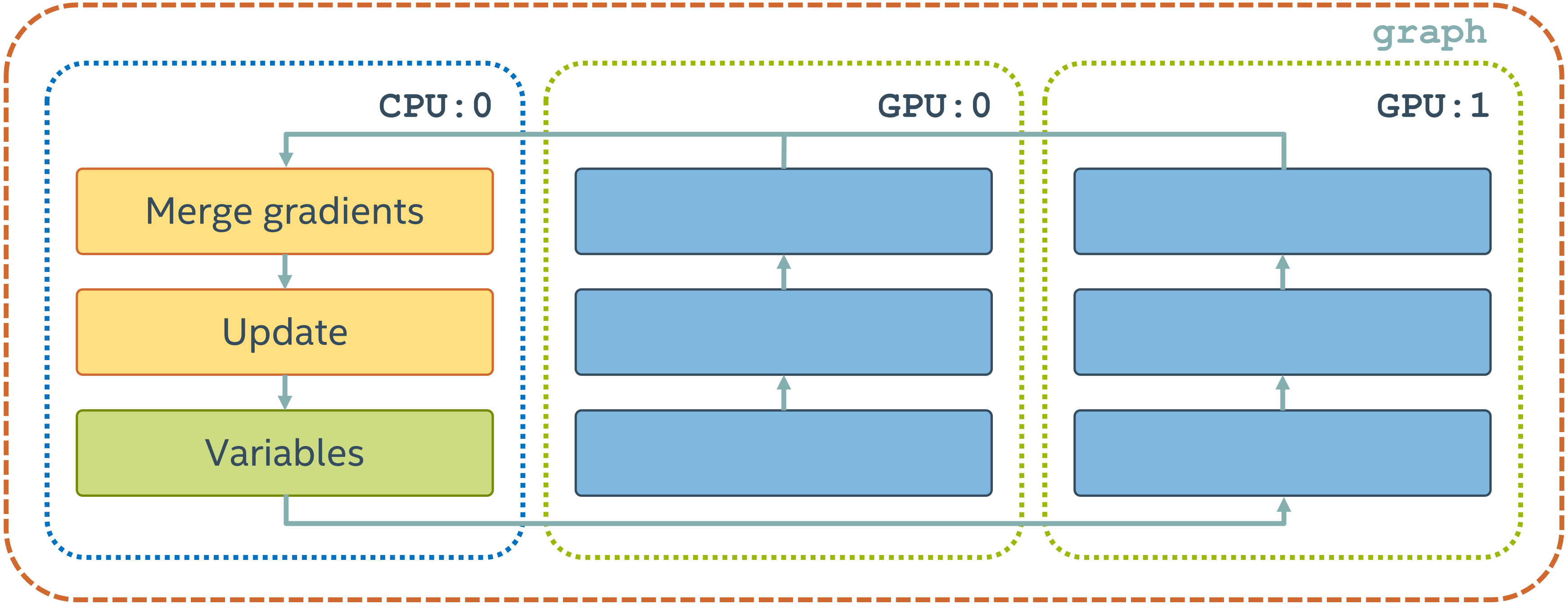
WANT TO MINIMIZE AS MUCH AS POSSIBLE!



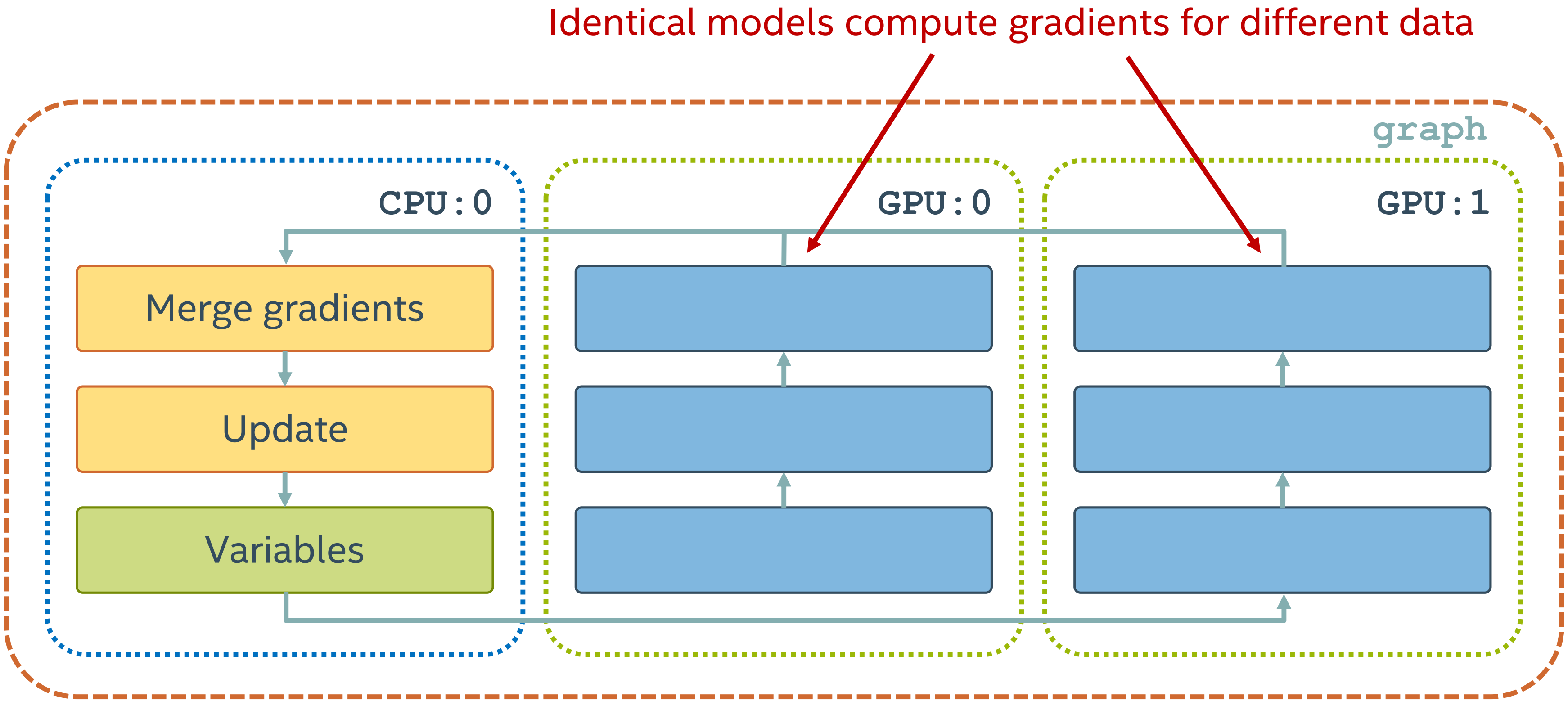
DATA PARALLEL



MORE COMMON PATTERN FOR MODELS

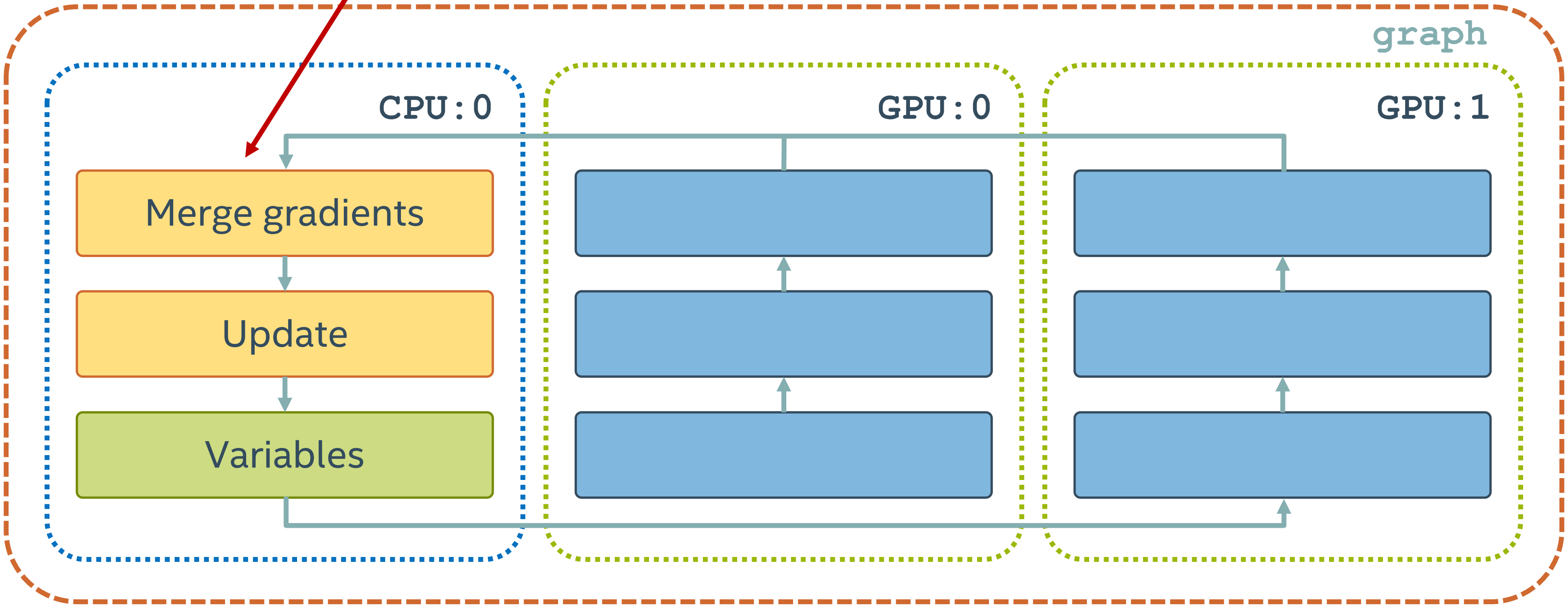


DATA PARALLEL



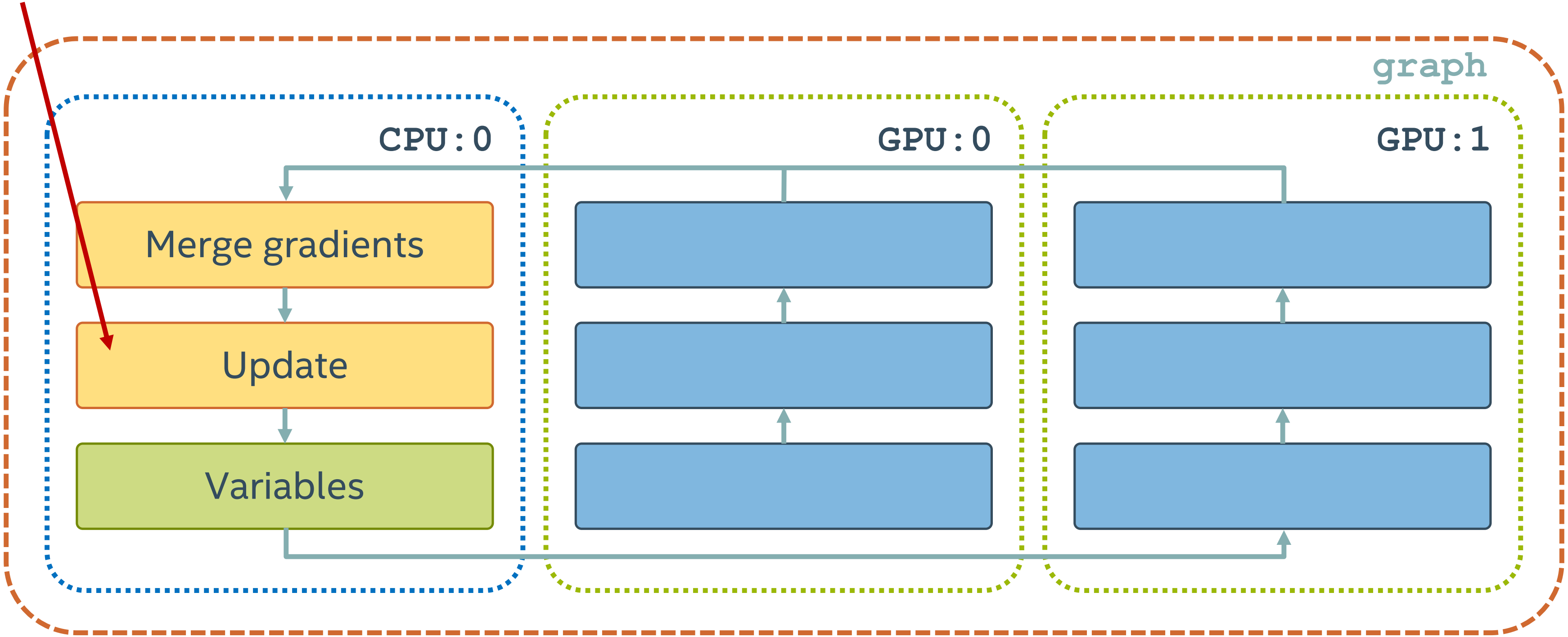
DATA PARALLEL

Gradients Variables can then be merged/averaged across all GPUs



DATA PARALLEL

Gradients for Variables can then be merged/averaged across all GPUs



BASIC IDEA FOR DATA PARALLEL MODELS

Placed on CPU:

- Variables (including global step), shared updates
- Queues, preprocessing steps
- Initializers, merged summaries, Savers

Placed on GPU:

- Replicated model computation
- Individual loss functions

MORE GENERALLY

Placed on CPU:

- Anything that needs to be shared between model replicas

Placed on GPU:

- Anything that doesn't need to be shared

HOW WE'RE GOING TO ACCOMPLISH THIS:

- Modify our layer functions to create Variables on CPU
- Reuse Variables via variable_scope() for each GPU
- Loop over each GPU in our system, placing models on each
- Average the calculated gradient from each GPU for updates

RNNS AND SEQ2SEQ (BRIEFLY)

MODELLING SEQUENCES OF DATA

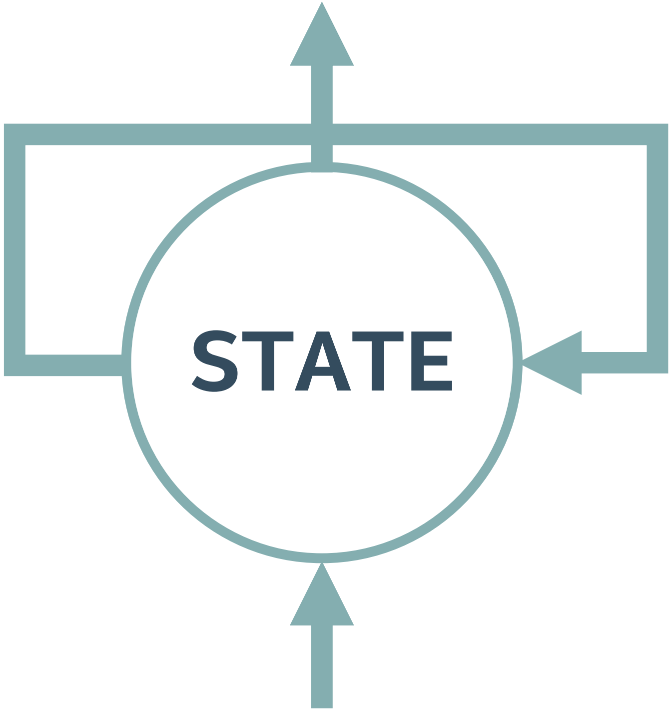
Thus far, our models required data be set to a specific size

What if we want to be able to model text?

Sentences can be of any length!

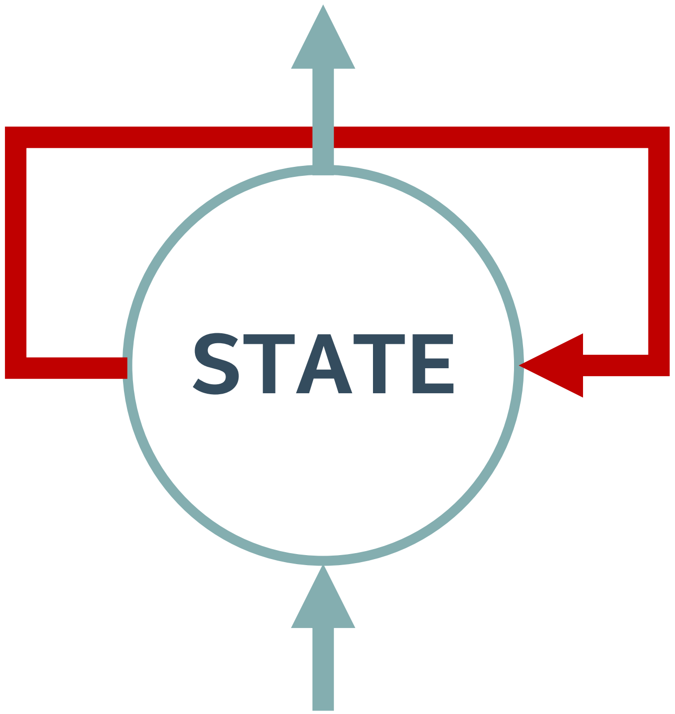
What would our model look like?

IDEA: RECURRENCE



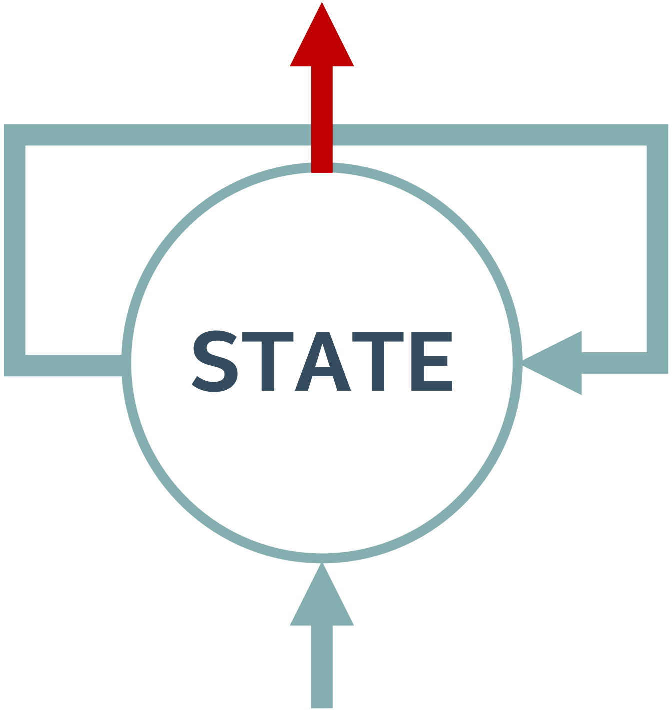
IDEA: RECURRENCE

Have some sort of “state” that gets passed back into itself



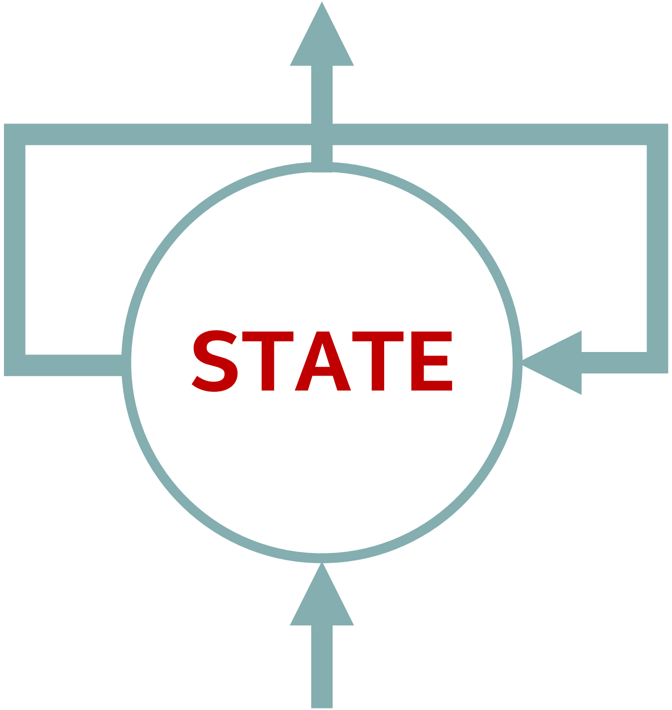
IDEA: RECURRENCE

At each “step”, output a prediction from the current state

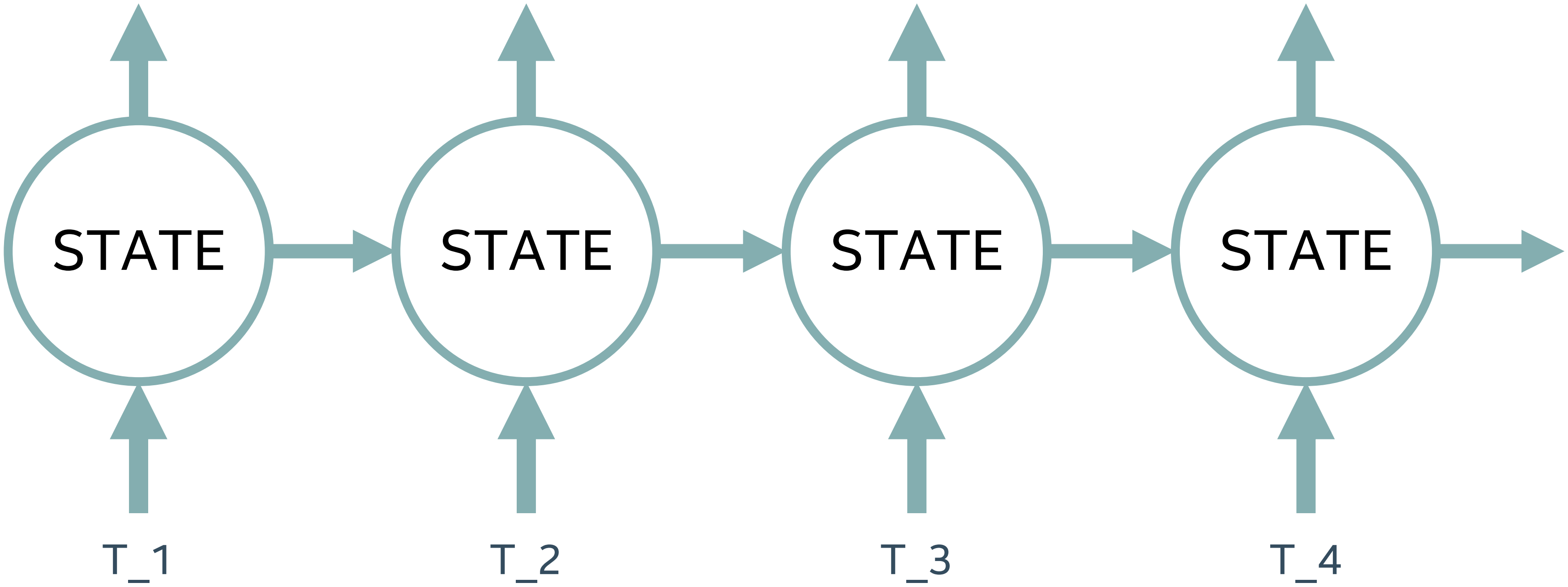


IDEA: RECURRENCE

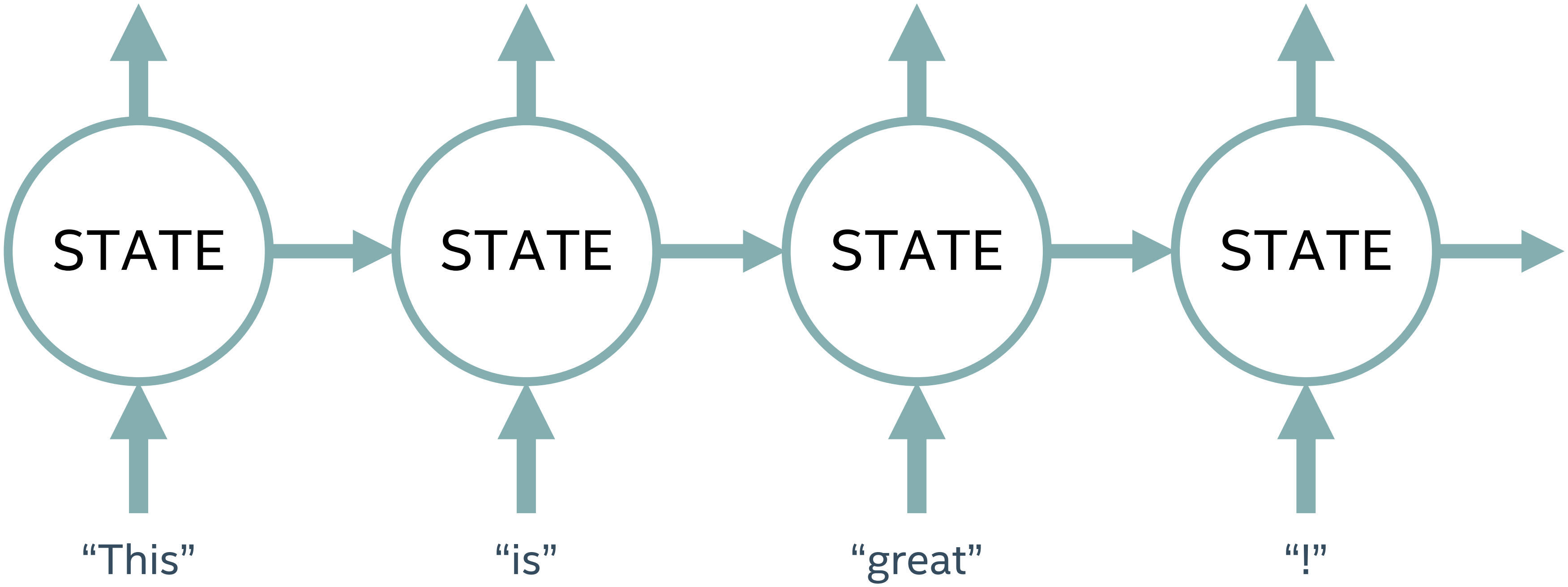
The ”state” is just a vector of neurons, similar to a FF network



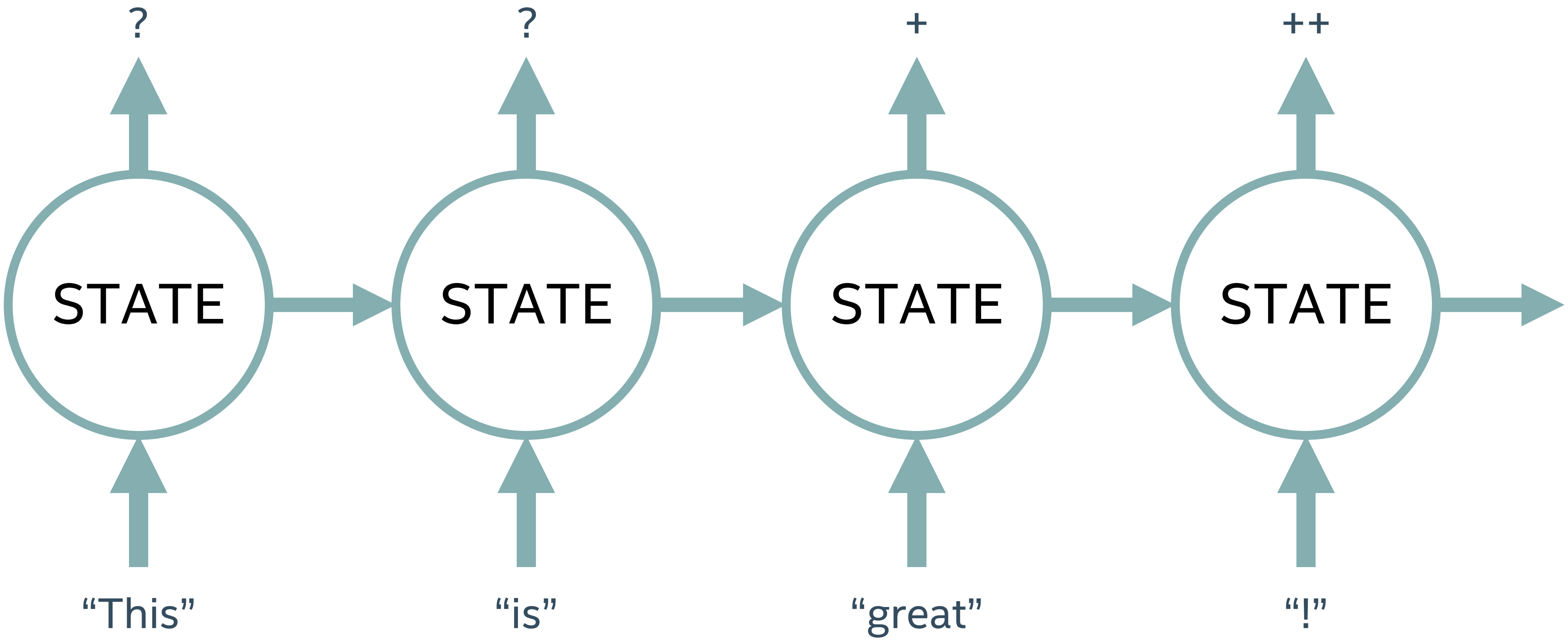
UNROLLED VIEW OF AN RNN



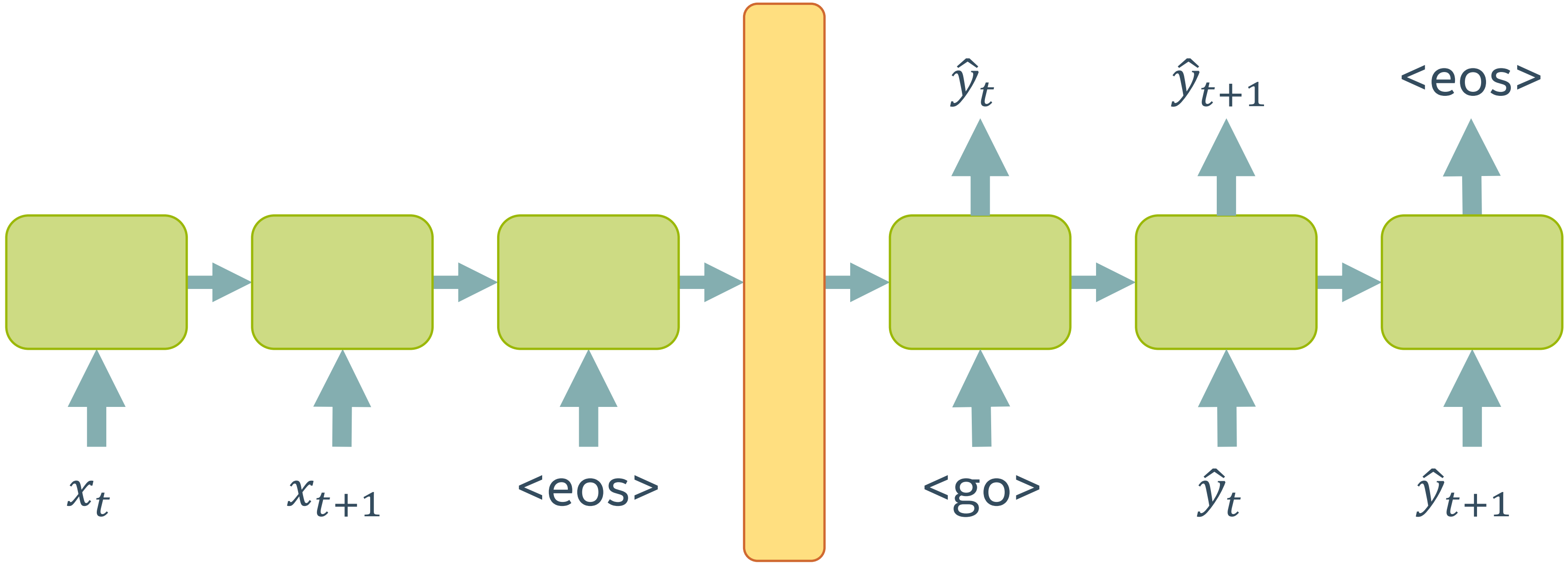
AT EACH STEP, WE PASS IN AN INPUT



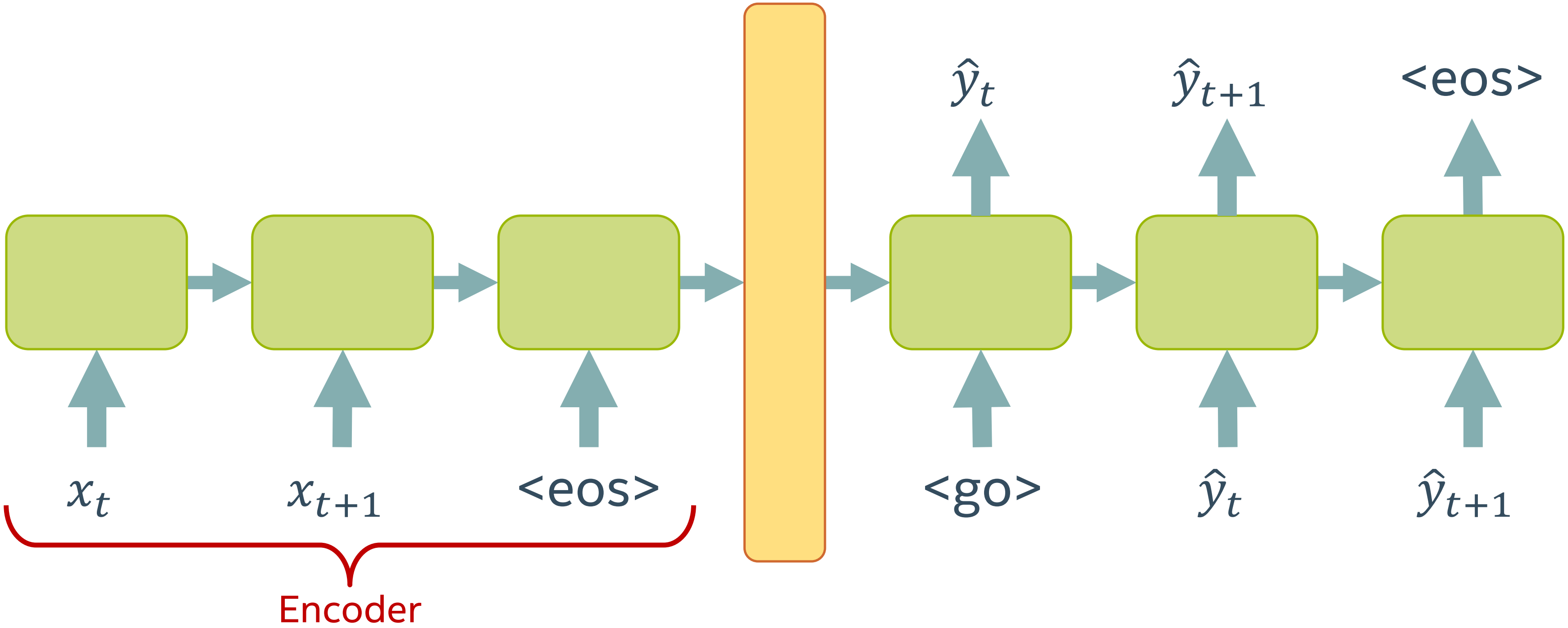
AS WE GO ALONG, THE NETWORK MAKES A PREDICTION, WHICH CHANGES AS IT GETS MORE INFO



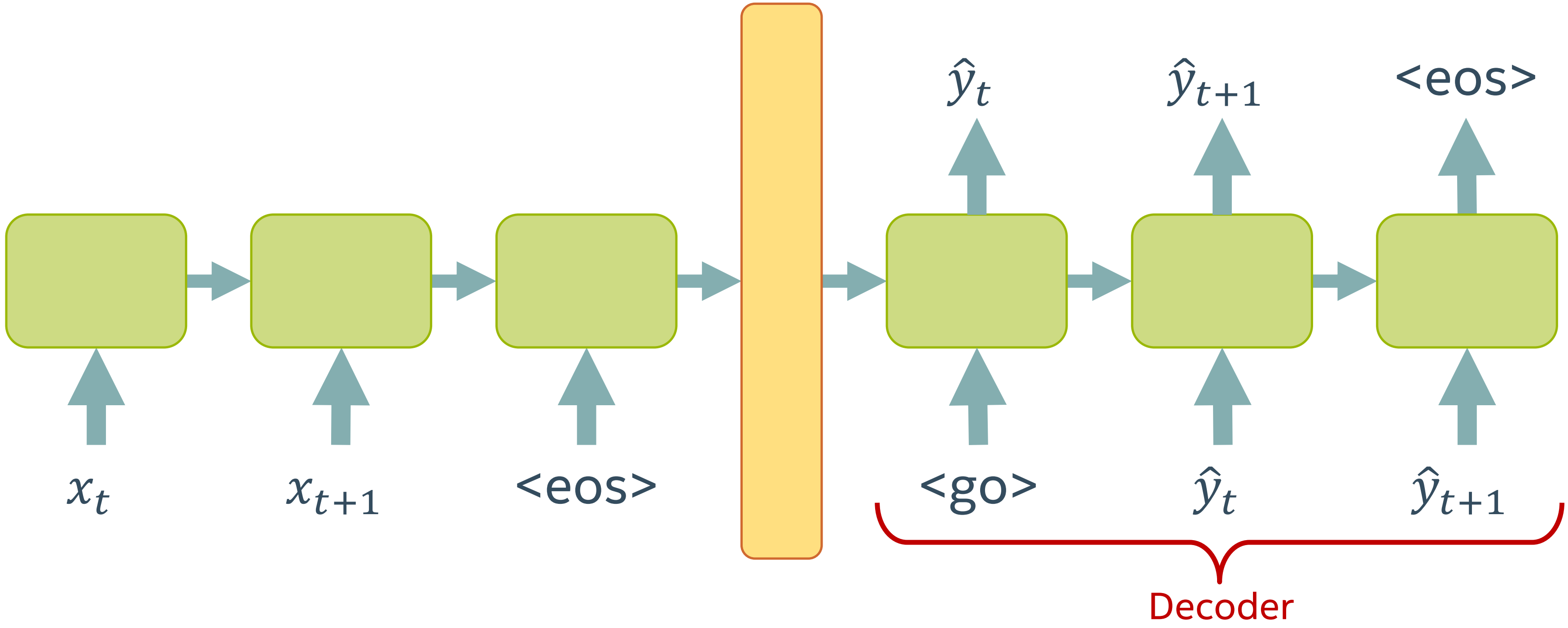
SEQUENCE TO SEQUENCE



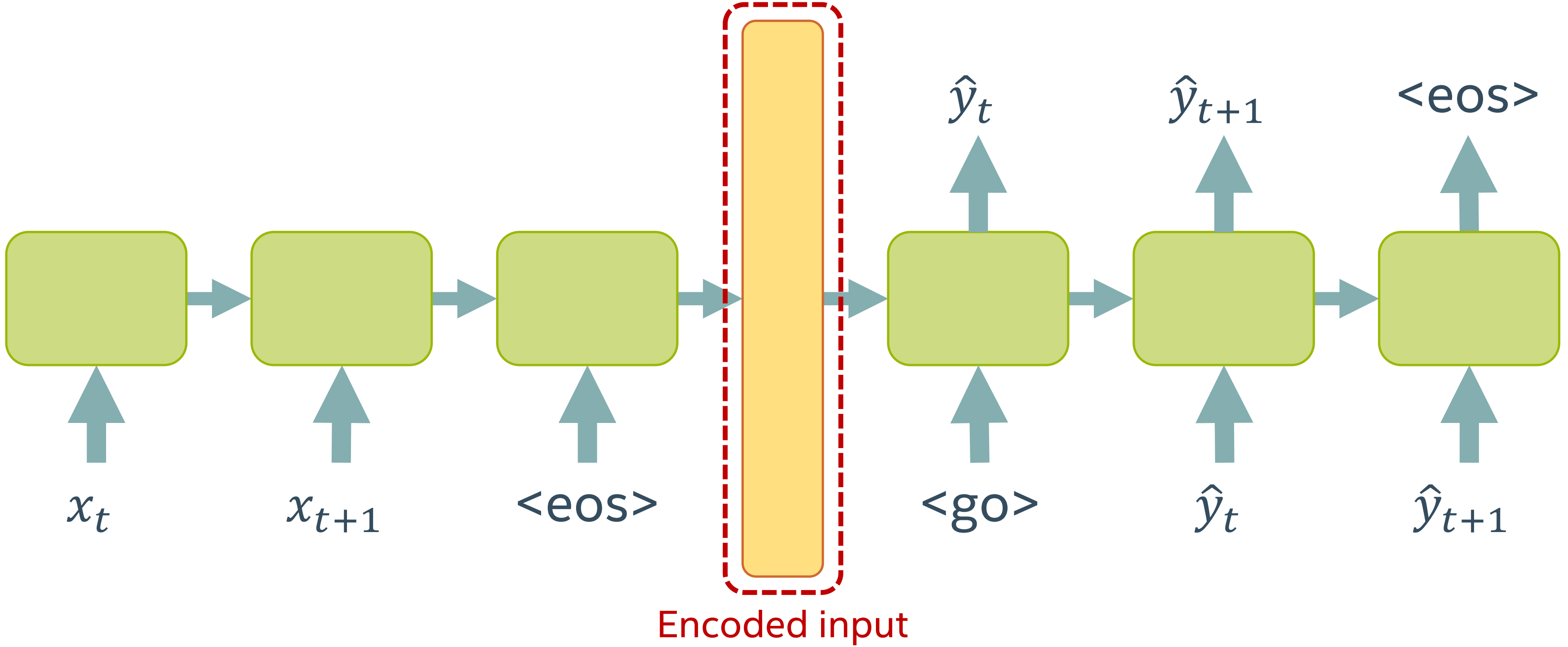
SEQUENCE TO SEQUENCE



SEQUENCE TO SEQUENCE

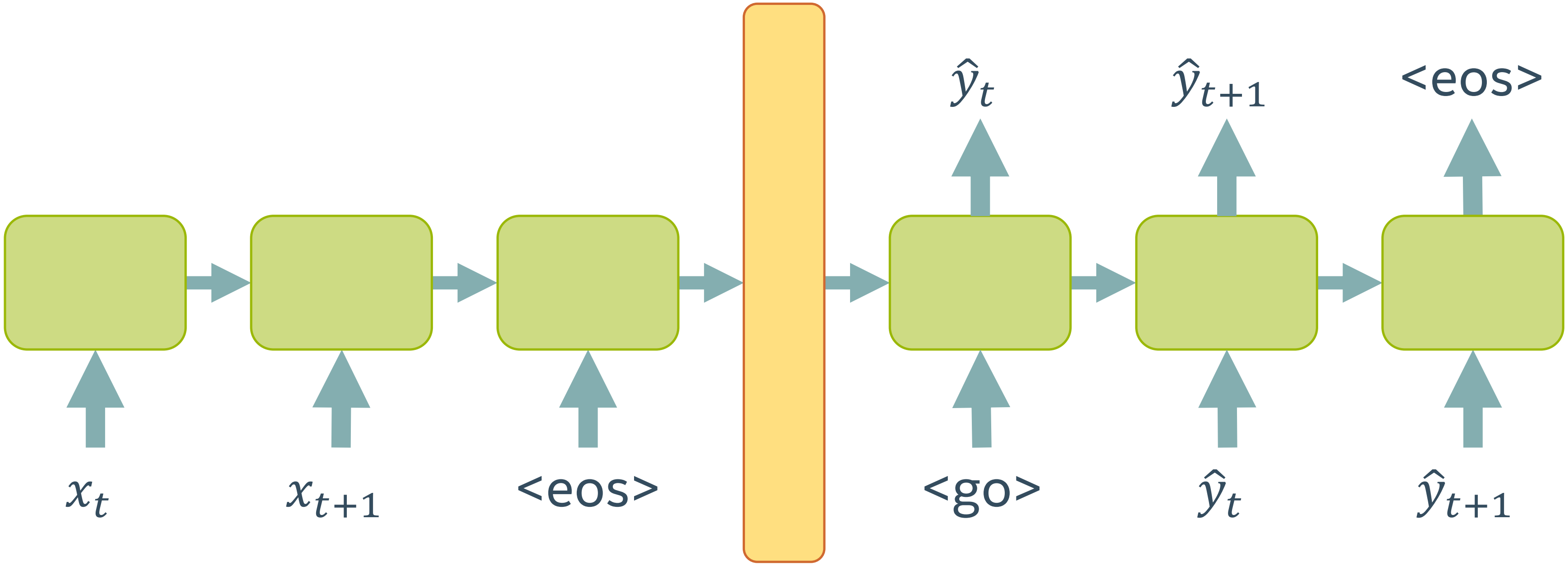


SEQUENCE TO SEQUENCE



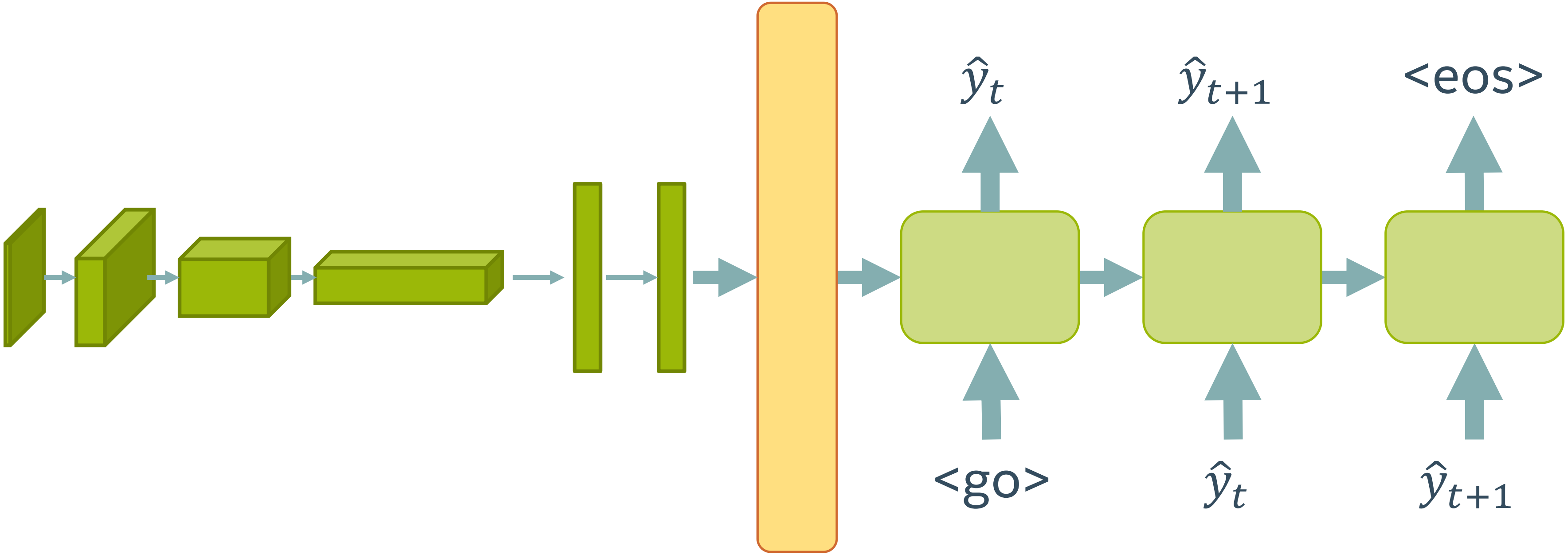
SHOW AND TELL

Instead of this:



SHOW AND TELL

Try this!



SHOW AND TELL

Idea: instead of encoding with an RNN, use a CNN

Use flattened output vector as the initial input of RNN

Run one step of RNN to calculate "initial hidden state"

Then have the RNN decode words as before

https://arxiv.org/pdf/1411.4555.pdf

SUCCESS AND LIMITATIONS

intel® Nervana™ AI Academy