Autonomous and Adaptive Systems

Reinforcement Learning in TensorFlow
Advanced Topics

Mirco Musolesi

mircomusolesi@acm.org

Eager Execution in TensorFlow 2.x

- The execution in TensorFlow was based on definition of computational graphs. In v2.x, *eager execution* is default. You can imagine eager execution as the standard imperative style of execution of Python.
- Before, computational graphs had to be declared and this was rather cumbersome. However, there are different ways for improving the performance of TensorFlow or customising it. This is essential for example when we want to define non-standard gradients.
- We will see two types of customisation:
  - the use of `tf.GradientTape` for customised gradients;
  - the use of `tf.function` for defining TensorFlow graphs.

Computing (Customised) Gradients

- Until now, we have used the standard TensorFlow operations for training a network.
- The optimization process has been based on the `fit()` function. `fit()` takes care of the calculation of the gradient values and the backpropagation. But what happens if we want to define “customised” gradients?
- This is particularly important for example when we want to exploit deep learning architectures for reinforcement learning.
  - See for example calculation of the gradient in REINFORCE.

Gradient Tapes

- TensorFlow provides the `tf.GradientTape` API for automatic differentiation, i.e., computing the gradient of a computation with respect to its input variables.
- TensorFlow records all operations executed inside the context of a `tf.GradientTape` onto a tape.
- TensorFlow then uses that tape and the gradients of the outputs with respect to the intermediate values computed during a recorded `tf.GradientTape` context.

Gradient Tapes

```
x = tf.ones((2, 2))

with tf.GradientTape() as t:

   t.watch(x)

   y = tf.reduce_sum(x)

   z = tf.multiply(y, y)

#Derivative of z with respect to the original input tensor x

dz_dx = t.gradient(z, x)

dz_dx_0_0 = dz_dx[0][0].numpy() # 2 (2*x_0_0 at x_0_0 = 1)

dz_dx_0_1 = dz_dx[0][1].numpy() # 2 (2*x_0_1 at x_0_1 = 1)

dz_dx_1_0 = dz_dx[1][0].numpy() # 2 (2*x_1_0 at x_1_0 = 1)

dz_dx_1_1 = dz_dx[1][1].numpy() # 2 (2*x_1_1 at x_1_1 = 1)
```

Gradient Tapes

- By default, the resources held by a `GradientTape` are released as soon as the `GradientTape.gradient()` method is called.
- To compute multiple gradients over the same computation, it is necessary to create a persistent gradient tape.
- This allows multiple calls to the `gradient()` method.
- Resources are released when the tape object is garbage collected.

Gradient Tapes

```
x = tf.constant(3.0)

with tf.GradientTape(persistent=True) as t:

   t.watch(x)

   y = x*x

   z = y*y

dz_dx = t.gradient(z, x) #108.0 (4*x^3 at x = 3)

dy_dx = t.gradient(y, x) # 6 (2*x at x = 3)

del t # remove the reference to the tape and invoke garbage collection
```

`with… as` construct in Python

- When the `with` statement is executed, Python evaluates the expression, called the `__enter__` method on the resulting value, which is called a context guard and assign the object returned by `__enter__` to the variable given by `as`.
- Python will then execute the body of the code.
- In any case, also in case of an exception the `__exit__` method of the guard object is executed.

`with… as` construct in Python

```
class guarded_execution:

    def __enter__(self):

        <initialisation>

        return p

    def __exit__(self, type, value, traceback):

        <free resources and manage exceptions>



with guarded_execution as p:

    <some instructions>
```

`with… as` construct in Python

```
with open(“textfile.txt”) as f:

  data = f.read()

  <work with data>
```

- See: https://effbot.org/zone/python-with-statement.htm

`tf.function`

- `tf.function` allows to transform a subset of Python syntax into portable and high-performance TensorFlow graphs, which are the component that are “under its hood”.

- It is possible to write “graph code” using natural Python syntax.

  - This is topic is outside the scope of this module. You can find the definition of the language and more details in the TensorFlow documentation.

Custom Training Loops with Keras Models

- The steps are as follows:
  - Compute the gradients with `tf.GradientTape`;
  - Process the gradients (if necessary);
  - Apply the processed gradients with `apply_gradients()`.

Custom Training Loops with Keras Models

```
# Create an optimiser

opt = tf.keras.optimizers.SGD(learning_rate = 0.1)

# Compute the gradients for a list of variables

with tf.GradientTape() as tape:

   loss = <call_loss_function>

vars = <list_of_variables>

grads = tape.gradient(loss, vars)

# Process the gradients

processed_grads  = [process_gradient(g) for g in grads]

# Ask the optimiser to apply the processed gradients

opt.apply_gradients(processed_grads)
```

Custom Training Loops with Keras Models

- Some important additional notes:
  - You do not call `compile()` when you do not use `fit()`.
  - Remember `compile()` defines the loss function, the optimiser and the metrics.

TensorFlow tf.data API

- The `tf.data` API enables to build complex input pipelines.
- It is used for performance reasons.
- In the example, we will use the `tf.data.Dataset` abstraction that represents a sequence of elements, in which element consists of one or more components.
  - Example: in a training dataset an element might be a single training example, with a pair of tensor components representing the image and its label.
- There is a variety of methods of constructing dataset (please refer to the documentation).

Lambda Layers

- The Lambda layer so that arbitrary TensorFlow functions can be used when constructing sequen
- They are typically used for fast experimentation.
- They are defined as follows:

```
tf.keras.layers.Lambda( function, output_shape=None, mask=None, arguments=Non
**kwargs)
```

- For example if I want to add a layer that takes the square of the input I will write:

```
model.add(Lambda(lambda x:x**2))
```

- A possible use it for variable casting.

TF-Agents Library

- The TF-Agents Library is a Reinforcement Learning library based on TensorFlow.
- It provides a series of off-the-shelf environments including:
  - a wrapper for OpenGym environments
  - DM Control Library
  - Unity's ML-agents library
- It also provides support for a variety of ML algorithms, including REINFORCE and a variety of components such as support for replay buffers, etc.

Training Architecture

- A TF-Agent is composed of a several components implementing functionalities we saw during the module.
- In the phase of training a driver explores the environment using a collect policy to choose actions.
- It collects trajectories (experiences) and these are sent to an observer that saves them in a replay buffer.
- An agent pulls batches of trajectories from the replay buffer and train a neural network (or more than one depending on the algorithm) with them.

Training Architecture

- This is an architecture designed for parallelism: a driver might explore multiple environments in parallel.
- Why do we need an observer? You might wonder why the driver is not saving the trajectories directly. Indeed, this would be possible, but this would also make the architecture less flexible.
- In fact an observer can be used for example to process the trajectories in parallel. An observer is any function that takes in input a trajectory as argument.

OpenAI Gym Environment

- For example, the following is used to instantiate the Atari 2600 environment of OpenAI Gym:

```
from tf_agents.environments import suite_gym

env = suite_gym.load(“Breakout-v1”)
```

OpenAI Gym Environment

- For example, the following is used to instantiate the Atari 2600 environment of OpenAI Gym:

```
from tf_agents.environments import suite_gym

env = suite_gym.load(“Breakout-v1”)
```

OpenAI Gym Environment

- TF-Agents environments are very similar to OpenAI Gym environments, but there are a few differences.
- The `reset()` method returns a `TimeStep` object that wraps the observation (plus some extra information).
- The `step()` method returns a `TimeStep` object as well.

OpenAI Gym Environment

- The `reward` and `observation` attributes are the same as in OpenAI Gym, except for the fact that the reward is represented using a NumPy array.

- The `step_type` attribute is equal to 0 for the first time in the episode, 1 for intermediate time steps and 2 for the final time step.

OpenAI Gym Environment

- The observations are screenshots of the Atari screen represented as NumPy array of shape [210, 160, 3].
- To render an environment you can call:

```
env.render(mode="human")
```

- If you want to get the image as a RGB array you have to call:

```
env.render(mode="rgb_array")
```

Replay Buffers

- The TF-Agents library provides various replay buffer implementations in the `tf_agents.rep` package.
- We will use the `TFUniformReplayBuffer` class in the `tf_agents.replay_buffers.tf_uniform_replay_buffer` package.
- It provides a high performance implementation of a replay buffer with uniform sampling.

Implementing Atari Games

- We will implement Breakout in Atari as example. It is worth noting that we will apply some pre-processing as follows:
- Grayscale and downsampling: observations are converted to grey-scale and downsampled (by default to 84x84 pixels).
- Frame skipping: the agent only gets to see n frames of the game (by default $n = 4$. Its actions are repeated for each frame. This is used to speed up the training).

Implementing Atari Games

- Since the default Atari environment already applies random frame skip, we use the raw, non-skipping variant of the games, for example for Breakout is called `BreakoutNoFrameskip-v4`.
- A single frame is also not sufficient to know the direction and the speed of the ball. It would be impossible to play the game with a default feed forward network.
- As we said, the best way to handle this is to us an environment wrapper that will output observations composed of multiple frames (a stack). This is implemented by the `FrameStack4` wrapper.

Collect Driver

- A driver is an object that explores an environment using a given policy, collect experiences and broadcasts them to some observers.
- At each step, the following things happen:
  - The driver passes the current time step to the collect policy, which uses this time step to choose an action and returns an action step containing the action.
  - The driver then passes the action to the environment, which returns the next step.
  - Finally, the driver creates a trajectory object to represent this transition and broadcasts it to all observers.

Collect Driver

- There are two main driver classes: `DynamicStepDriver` and `DynamicEpisodeDriver`.
- The first one collects experiences for a given number of steps, while the second collects experiences for a given number of episodes.
- In our example, we will collect experiences for given number of episodes.
- We want to collect experiences for four steps for each iteration (as in the original DeepMind's DQN paper).

Initialising the Replay Buffer

- It is usually a good idea to initially fill the replay buffer with observation. In order to do some use a `RandomTFPolicy` class and create a second driver that will run this policy for 20000 steps (which is equivalent to 80000 simulator frames as in the 2015 paper).

- In the code we call it `initial_collect_policy.`

Extensions

- Implementation with a Policy Networks (DQN) with Actor-Critic
- Use of Advantage Actor-Critic (A2C)
- Use of Asynchronous Advantage Actor-Critic (A3C)
- Use of Soft-Actor Critic
- Use of Trust-Region Optimization (available in TF-Agents)
- Use of Proximal Policy Optimization (available in TF-Agents)

References

- Aurelien Geron Hands-On Machine Learning with Scikit-Learn , Keras and Tensor Flow. Third Edition. O'Reilly. 2022.

- TensorFlow Documentation.