TensorForce: A TensorFlow library for applied reinforcement learning

11.07.2017

This blogpost will give an introduction to the architecture and ideas behind TensorForce, a new reinforcement learning API built on top of TensorFlow.

This post is about a practical question: How can the applied reinforcement learning community move from collections of scripts and individual examples closer to an API for reinforcement learning (RL) — a 'tf-learn' or 'skikit-learn' for RL? Before discussing the TensorForce framework, we will discuss observations and thoughts that motivated the project. Feel free to skip this part if you are just interested in the API walkthrough. We want to emphasize that this post does not contain an introduction to deep RL itself, and neither presents a new model or discusses latest state-of-the-art algorithms, hence the content might be of limited interest to pure researchers.

Motivation

Say you are a researcher in computer systems, natural language processing, or some other applied domain. You have a basic understanding of RL and are interested in exploring deep RL to control some aspect of your system.

There is a number of blog-posts with introductions to deep RL, DQN, vanilla policy gradients, A3C, and so forth (we like Karpathy's, in particular for its great description of the intuition behind policy gradient methods). There is also a lot of code out there to help with getting started, e.g. the OpenAI starter agents, rllab, and many Github projects implementing specific algorithms.

However, we observe a significant gap between these research frameworks and using RL for practical applications. Here are a few potential issues when moving to applied domains:

- Tight coupling of RL logic with simulation handles: Simulation environment APIs are very convenient, for instance, they make it possible to create an environment object and then use it somewhere in a for loop that also manages internal update logic (e.g. by collecting output features). This makes sense if the goal is to evaluate an RL idea, but it is harder to disentangle RL code and simulation environment. It also touches on the question of control flow: Can the RL code call the environment when it is ready, or does the environment call the RL agent when it requires a decision? For RL library implementations to be applicable in a wide range of domains, we often need the latter.
- Fixed network architectures: Most example implementations contain hardcoded neural network architectures. This is usually not a big problem, as it is straightforward to plug in or remove different network layers as necessary. Nonetheless, it would be better for an RL library to provide this functionality as a declarative interface, without having to modify library code. In addition, there are cases where modifying the architecture is (unexpectedly) more difficult, for instance, if internal states need to be managed (see below).
- Incompatible state/action interface: A lot of early open-source code using the popular OpenAI Gym environments follows the simple interface of a flat state input and a single discrete or continuous action output. DeepMind Lab, however, uses a dictionary format for, in general, multiple states and actions, while OpenAI Universe uses named key events. Ideally, we want an RL agent to be able to handle any number of states and actions, with potentially different types and shapes. For example, one of the TensorForce authors is using RL in NLP and wants to handle multimodal input, where a state conceptually contains two inputs, an image and a corresponding caption.
- Intransparent execution settings and performance issues: When writing TensorFlow code, it is natural to first focus on the logic. This can lead to a lot of duplicate/unnecessary operations being created or intermediate values unnecessarily being materialized. Further, distributed/asynchronous/parallel reinforcement learning is a bit of a moving target and distributed TensorFlow requires a fair amount of hand-adjusting to a particular hardware setting. Again, it would be neat to eventually have an execution configuration that could just declare available devices or machines and have everything else managed internally,

e.g. two machines with given IPs, which are supposed to run asynchronous VPG.

To be sure, none of these issues is meant to critize research code, since there is usually no intent for the code to be used as an API for other applications in the first place. Here we are presenting the perspective of researchers who want to apply RL in different domains.

The TensorForce API

TensorForce provides a declarative interface to robust implementations of deep reinforcement learning algorithms. It is meant to be used as a library in applications that want to utilize deep RL, and enables the user to experiment with different configurations and network architectures without caring about all the underlying bells and whistles. We fully acknowledge that current RL methods tend to be brittle and require a lot of fine-tuning, but that does not mean it is not the time yet to think about general-purpose software infrastructure for RL solutions.

TensorForce is *not* meant to be a raw collection of implementations which require significant work to be used for application in environments other than research simulations. Any such framework will inevitably include some structural decisions which make non-standard things more annoying (leaky abstractions). This is why core RL researchers might prefer to build their models from scratch. With TensorForce we are aiming to capture the overall direction of state-of-the-art research, with its emerging insights and standards.

In the following, we will go through various fundamental aspects of the TensorForce API and discuss our design choices.

Creating and configuring agents

We begin by creating an RL agent using the TensorForce API.

```
from tensorforce import Configuration
from tensorforce.agents import DQNAgent
from tensorforce.core.networks import layered_network_builder

# Define a network builder from an ordered list of layers
layers = [dict(type='dense', size=32),
          dict(type='dense', size=32)]
network = layered_network_builder(layers_config=layers)

# Define a state
states = dict(shape=(10,), type='float')

# Define an action (models internally assert whether
# they support continuous and/or discrete control)
actions = dict(continuous=False, num_actions=5)

# The agent is configured with a single configuration object
agent_config = Configuration(
    batch_size=8,
    learning_rate=0.001,
    memory_capacity=800,
    first_update=80,
    repeat_update=4,
    target_update_frequency=20,
    states=states,
    actions=actions,
    network=network
)
agent = DQNAgent(config=agent_config)
```

The state and action in the example are a short-form of the more general state/action interface. Multimodal input consisting of an image and a caption, for instance, is defined as shown below. Similarly, multiple output actions can be defined. Note that the single-state/action short-form has to be used consistently for communication with the agent throughout the code.

```
states = dict(
    image=dict(shape=(64, 64, 3), type='float'),
    caption=dict(shape=(20,), type='int')
)
```

Configuration parameters depend on the underlying agent and model used. A full list of parameters for every agent can be found in the example configurations.

Currently, the following RL algorithms are available in TensorForce:

- A random agent baseline (RandomAgent)
- Vanilla policy gradient with generalized advantage estimation (VPGAgent)
- Trust region policy optimization (TRPOAgent)
- Deep Q-learning / double deep Q-learning (DQNAgent)
- Normalized advantage functions (NAFAgent)
- Deep Q-learning from expert demonstration (DQFDAgent)
- Asynchronous advantage actor-critic (A3C) implicitly via the distributed option

The last point means that there is no such thing as an A3CAgent, because A3C really describes a mechanism for asynchronous updates, not a particular agent. Consequently, an asynchronous update mechanism using distributed TensorFlow is part of the generic Model base class which all agents are derived from. The A3C agent as described in the paper is hence implicitly implemented by setting the distributed flag for the VPGAgent. It should be noted that A3C is not the optimal distributed update style for every model (or does not make sense at all for some models), and we will discuss implementing other approaches (such as PAAC) at the end of this post. The important point is to conceptually separate the question of agent and update semantics from execution semantics.

We also want to mention the distinction between model and agent. The Agent class defines the interface to use reinforcement learning as an API, and manages various things like incoming observation data, preprocessing, exploration, etc. The two key methods are agent.act(state), which returns an action, and agent.observe(reward, terminal), which updates the model according to the agent's mechanism, e.g. off-policy memory replays (MemoryAgent) or on-policy batches (BatchAgent). Note that for the agent's internal mechanisms to work correctly, these functions have to be called alternately. The Model class implements the core RL algorithm and provides the necessary interface via the methods get_action and update, which the agent calls internally at the relevant points. For instance, the DQNAgent is a MemoryAgent agent with a DQNModel and an additional line for target network updates:

```
def observe(self, reward, terminal):
    super(DQNAgent, self).observe(reward, terminal)
    if self.timestep >= self.first_update \
            and self.timestep % self.target_update_frequency == 0:
        self.model.update_target()
```

Neural network configuration

A key issue in RL is designing effective value functions. Conceptually, we view a model as a description of an update mechanism, which is separate from what is actually being updated — which in the case of deep RL is a (or multiple) neural network(s). Hence, there are no hardcoded networks in a model, but they are instantiated according to a configuration.

In the example above, we created a network configuration programmatically as a list of dicts describing each layer. Such a configuration can also be given as JSON, and utility functions are provided that turn it into a network constructor. Here an example of a JSON network specification:

```
[
    {
        "type": "conv2d",
        "size": 32,
        "window": 8,
        "stride": 4
    },
    {
        "type": "conv2d",
        "size": 64,
        "window": 4,
        "stride": 2
    },
    {
        "type": "flatten"
    },
    {
        "type": "dense",
        "size": 512
    }
]
```

As before, this configuration has to be added to the agent's configuration object:

```
from tensorforce.core.networks import from_json
```

```
agent_config = Configuration(
    ...
    network=from_json('configs/network_config.json')
    ...
)
```

The default layer activation is relu, but there are other activation functions available (currently, elu, selu, softmax, tanh and sigmoid). Moreover, other properties of a layer can be changed. For instance, a modified dense layer could look like this:

```
[
    {
        "type": "dense",
        "size": 64,
        "bias": false,
        "activation": "selu",
        "l2_regularization": 0.001
    }
]
```

We opted to not use existing layer implementations (e.g. from tf.layers) to have explicit control over the internal operations and guarantee that they properly integrate with the rest of TensorForce. We also wanted to avoid dependencies on changing wrapper libraries, and hence just rely on more low-level TensorFlow operations.

Our layer library so far only provides very few basic layer types, but will be extended in the future. It is also easily possible to integrate your own layer, shown in the following for the example of a batch normalization layer:

```
def batch_normalization(x, variance_epsilon=1e-6):
    mean, variance = tf.nn.moments(x, axes=tuple(range(x.shape.ndims - 1)))
    x = tf.nn.batch_normalization(x, mean=mean, variance=variance,
                                  variance_epsilon=variance_epsilon)
    return x
```

```
{
    "type": "[YOUR_MODULE].batch_normalization",
    "variance_epsilon": 1e-9
}
```

Thus far, we presented TensorForce functionality to create a layered network, that is, a network which takes a single input state tensor and applies a sequence of layers to produce an output tensor. In some cases, however, it might be required/preferred to deviate from such a layer-stack structure. Most obviously, this is necessary when having to handle multiple input states, which does not naturally fit with a single sequence of processing layers.

We do not (yet) provide a higher-level configuration interface to automatically create a corresponding network builder. Consequently, one has to programmatically define the network builder function for such cases, and add it to the agent configuration as before. Picking up on the earlier example of multimodal input (image and caption), we can define a network in the following way:

```
def network_builder(inputs):
    image = inputs['image']  # 64x64x3-dim, float
    caption = inputs['caption']  # 20-dim, int

    with tf.variable_scope('cnn'):
        weights = tf.Variable(tf.random_normal(shape=(3, 3, 3, 16), stddev=0.01))
        image = tf.nn.conv2d(image, filter=weights, strides=(1, 1, 1, 1))
        image = tf.nn.relu(image)
        image = tf.nn.max_pool(image, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1))

        weights = tf.Variable(tf.random_normal(shape=(3, 3, 16, 32), stddev=0.01))
        image = tf.nn.conv2d(image, filter=weights, strides=(1, 1, 1, 1))
        image = tf.nn.relu(image)
        image = tf.nn.max_pool(image, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1))

        image = tf.reshape(image, shape=(-1, 16 * 16, 32))
        image = tf.reduce_mean(image, axis=1)

    with tf.variable_scope('lstm'):
```

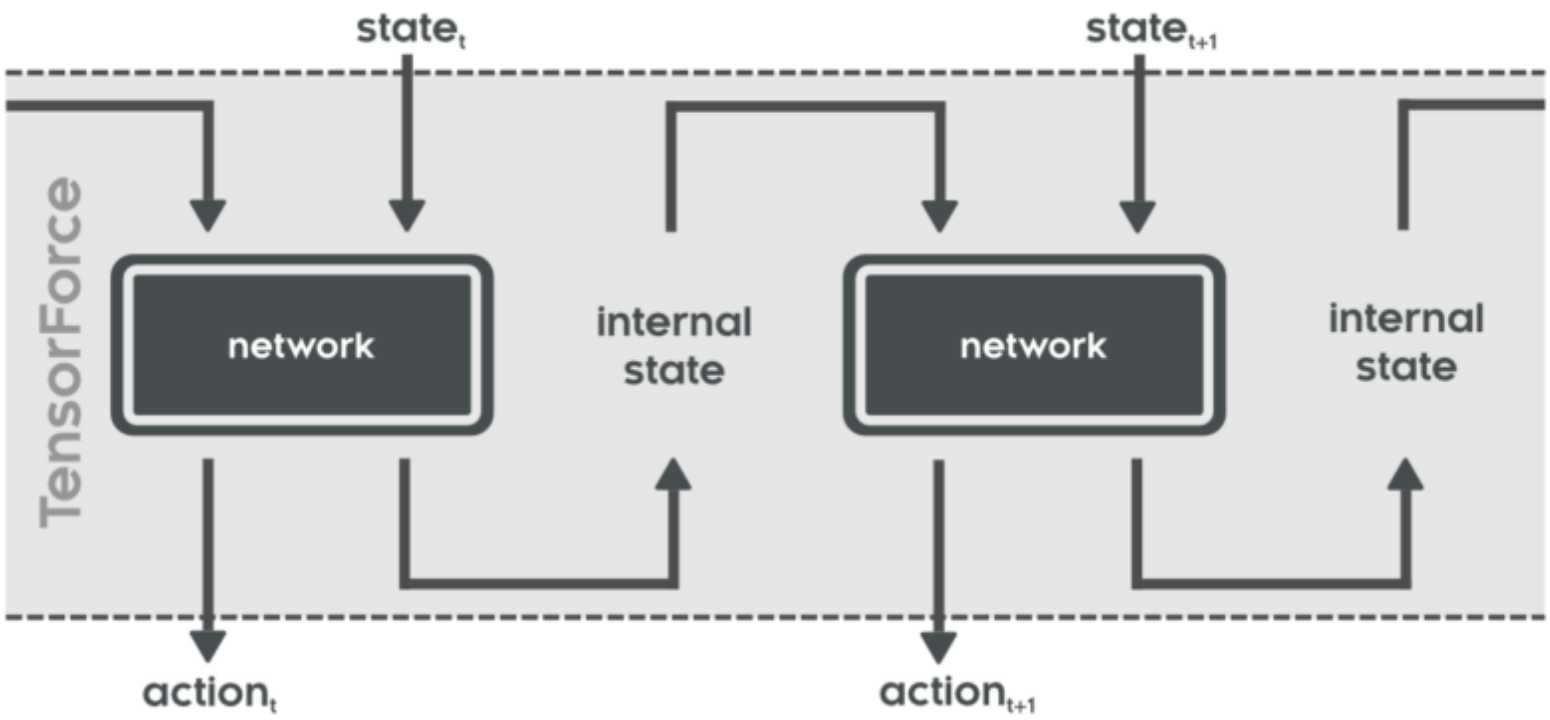
```
        weights = tf.Variable(tf.random_normal(shape=(30, 32), stddev=0.01))
        caption = tf.nn.embedding_lookup(params=weights, ids=caption)
        lstm = tf.contrib.rnn.LSTMCell(num_units=64)
        caption, _ = tf.nn.dynamic_rnn(cell=lstm, inputs=caption, dtype=tf.float32)
        caption = tf.reduce_mean(caption, axis=1)

    return tf.multiply(image, caption)


agent_config = Configuration(
    ...
    network=network_builder
    ...
)
```

Internal states and episode management

In contrast to the classic supervised learning setup where instances, and thus calls to a neural network, are (assumed to be) independent, the timesteps within one episode in RL do depend on previous actions and influence subsequent states. It is hence conceivable how a neural network, in addition to its *per-timestep* state inputs and action outputs, might have *within-episode* internal states with corresponding in/outputs per timestep. The following diagram illustrates the working of such a network over time:



The management of these internal states, i.e. forwarding them between timesteps and reseting them at the beginning of a new episode, is handled entirely by the TensorForce agent and model class. Note that this handles all relevant use cases (one episode within batch, multiple episodes within batch, episode without a terminal within batch). So far, there is the lstm layer type utilizing this functionality:

```
[
    {
        "type": "dense",
        "size": 32
    },
    {
        "type": "lstm"
    }
]
```

In this example architecture, the output of the dense layer is fed to an LSTM cell which then produces the final output for a timestep. When advancing the LSTM by one step, its internal state gets updated and represents the internal state output here. For the next timestep, both the new state inputs and this internal state are given to the network, which then advances the LSTM by another step and outputs both the actual output and the new internal LSTM state, and so on.

For a custom implementation of a layer with internal states, the function has to not only return the layer output, but also a list of internal state input placeholders, the corresponding internal state output tensors, and a list of internal state initialization tensors (all of the same length, in this order). The following code snippet presents (a simplified version of) our LSTM layer implementation, and illustrates how a custom layer with internal states can be defined:

def lstm(x):

```
    size = x.get_shape()[1].value
    internal_input = tf.placeholder(dtype=tf.float32, shape=(None, 2, size))
    lstm = tf.contrib.rnn.LSTMCell(num_units=size)
    state = tf.contrib.rnn.LSTMStateTuple(internal_input[:, 0, :],
                                          internal_input[:, 1, :])
    x, state = lstm(inputs=x, state=state)
    internal_output = tf.stack(values=(state.c, state.h), axis=1)
    internal_init = np.zeros(shape=(2, size))
    return x, [internal_input], [internal_output], [internal_init]
```

Preprocessing states

We can define preprocessing steps that are applied to the state (or states, if specified as a dictionary of lists), for instance, to downsample visual input. Below is an example for the Arcade Learning Environment preprocessor as used in most DQN implementations:

```
config = Configuration(
    ...
    preprocessing=[
        dict(
            type='image_resize',
            kwargs=dict(width=84, height=84)
        ),
        dict(
            type='grayscale'
        ),
        dict(
            type='center'
        ),
        dict(
            type='sequence',
            kwargs=dict(
                length=4
            )
        )
    ]
    ...
)
```

Every preprocessor in the stack has a type and optionally a list of args or/and a dict of kwargs. The sequence preprocessor, for instance, takes the last four states (i.e. frames) and stacks them on top of each other, to emulate the Markov property. As a side node: This is obviously not necessary when using, for instance, the aforementioned lstm layer, which is able to model and communicate time dependencies via internal states.

Exploration

Exploration can also be defined in the configuration object, which is applied by the agent to the action its model decided on (to handle multiple actions, again, a dictionary of specifications can be given). For instance, to use Ornstein-Uhlenbeck exploration for a continuous action output, the following specification is added to the configuration:

```
config = Configuration(
    ...
    exploration=dict(
        type='OrnsteinUhlenbeckProcess',
        kwargs=dict(
            sigma=0.1,
            mu=0,
            theta=0.1
        )
    )
    ...
)
```

The following lines add epsilon exploration for a discrete action, which decays over time to a final value:

```
config = Configuration(
    ...
    exploration=dict(
        type='EpsilonDecay',
        kwargs=dict(
            epsilon=1,
            epsilon_final=0.01,
            epsilon_timesteps=1e6
        )
    )
    ...
)
```

Using agents with the runner utility

Let's use an agent: The code below runs an agent on our test environment (MinimalTestEnvironment), which we use for continuous integration — a minimal environment verifying that the act, observe and update mechanisms for a given agent/model work. Note that all our environment implementations (OpenAI Gym, OpenAI Universe, DeepMind Lab) use the same interface, thus running tests with another environment is straightforward.

The Runner utility facilitates the process of running an agent on an environment. Given an arbitrary agent and environment instance, it manages the number of episodes, maximum length per episode, termination conditions, etc. The runner also accepts a cluster_spec argument and, if provided, manages distributed execution (TensorFlow supervisors/sessions/etc). Via the optional episode_finished argument, one can periodically report results and give criteria for stopping the execution before the maximum number of episodes.

```
environment = MinimalTest(continuous=False)

network_config = [
    dict(type='dense', size=32)
]
agent_config = Configuration(
    batch_size=8,
    learning_rate=0.001,
    memory_capacity=800,
    first_update=80,
    repeat_update=4,
    target_update_frequency=20,
    states=environment.states,
    actions=environment.actions,
    network=layered_network_builder(network_config)
)

agent = DQNAgent(config=agent_config)
runner = Runner(agent=agent, environment=environment)

def episode_finished(runner):
    if runner.episode % 100 == 0:
        print(sum(runner.episode_rewards[-100:]) / 100)
    return runner.episode < 100 \
        or not all(reward >= 1.0 for reward in runner.episode_rewards[-100:])

runner.run(episodes=1000, episode_finished=episode_finished)
```

For the sake of completeness, we explicitly give a minimal loop for running an agent on an environment:

```
episode = 0
episode_rewards = list()

while True:
    state = environment.reset()
    agent.reset()

    timestep = 0
    episode_reward = 0
    while True:
```

```
        action = agent.act(state=state)
        state, reward, terminal = environment.execute(action=action)
        agent.observe(reward=reward, terminal=terminal)

        timestep += 1
        episode_reward += reward

        if terminal or timestep == max_timesteps:
            break

    episode += 1
    episode_rewards.append(episode_reward)

    if all(reward >= 1.0 for reward in episode_rewards[-100:]) \
            or episode == max_episodes:
        break
```

As noted in the introduction, being able to use the runner class depends on the control flow in a given application scenario. If RL can be used in a way such that it makes sense to query state information from within TensorForce (e.g. via a queue or a webservice) and return actions (to another queue or service), it can be useful to implement the environment interface and consequently be able to use (or extend) the runner utility.

A more common case might be to use TensorForce as a library from an external application which drives control, thus not being able to provide an environment handle. This might seem like a negligible aspect to researchers, but it is a typical deployment issue in e.g. computer systems, which is also a fundamental reason why most research scripts are impractical to use for anything but simulations.

Another point worth mentioning is that the declarative central configuration object makes it straightforward to interface all components of an RL model with hyperparameter optimization, particularly also the network architecture.

Further considerations

We hope some of you might find TensorForce useful. Our focus thus far has been on getting an architecture in place which we feel allows us to implement different RL concepts and new approaches in a consistent way, and take away the friction of exploring deep RL use cases in novel domains.

Deciding on features to include in a practical library in a fast moving field is difficult. There is a large number of algorithms and concepts out there, and seemingly every week a new idea gets better results on a subset of the Arcade Learning Environment (ALE) environments. There is also the issue that many ideas only really work in environments that are easy to parallelize or have a very particular episodic structure — there is no precise notion of environment properties and how they relate to different approaches. Yet, there are some clear trends:

- Hybridization of policy gradient and Q-learning approaches to improve sample efficiency (PGQ, Q-Prop,...): This is a logical thing to do, and while it remains to be seen which hybrid strategy will prevail, this is what we would view as something that will become the next ‘standard approach’. Understanding the utility of these approaches in different applied domains (data-rich/data-sparse) is something we are really interested in. A highly subjective opinion of ours is that most applied researchers tend to use vanilla policy gradient variants because they are easy to understand, implement and, importantly, less brittle than newer algorithms, which might require a lot of fine-tuning to handle potential numerical instabilities. A different view is that non-RL researchers are simply not aware of relevant novel approaches, or would not want to put in the effort of implementing them, and this is what motivates TensorForce. Finally, it is worth considering that the update mechanism in applied domains is often less important than modelling states, actions, and rewards, as well as the network architecture.
- Better utilisation of GPUs and available devices for parallel/asynchronous/distributed approaches (PAAC, GA3C,..): One issue with approaches in this domain are implicit assumptions about how much time is spent collecting data versus updating. These assumptions may not hold true in non-simulation domains, and understanding how environment properties should affect device execution semantics requires more research. We are also still using ‘feed_dicts’ but are thinking about improving input processing performance.
- Exploration modes (e.g. count based exploration, parameter space noise,...)
- Decomposition of large discrete action spaces, hierarcical models and subgoals: For instance, see Dulac-Arnold et al.. Complex discrete action spaces (e.g. many state-dependent suboptions) are highly relevant in applied domains, but difficult to provide as an API at this point. We are expecting a lot of work here in the coming years.
- Internal modules for state prediction and novel model-based approaches: For instance, see the Predictron.
- Bayesian deep reinforcement learning and reasoning about uncertainty

The bottom line is that we are following these developments and will try to adopt established techniques we are missing (arguably many), as well as new ideas once we are convinced they have the potential to become robust standard methods. In this sense, we are explicitly not competing with research frameworks with higher coverage.

Final note: We have an internal version to experiment with ideas on how to make recent advanced approaches usable as library functions. Once we are happy with something, we will consider moving it open-source, so if there is a longer period without updates on Github, that is likely because we are trying to make something work internally (or because our PhDs get too busy), not because we have abandoned the project. If you have an interesting applied use case, do get in touch.

We aim to publish more blog posts in the future, with more detailed walkthroughs of newer algorithms.

Michael Schaarschmidt, Alexander Kuhnle, Kai Fricke

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment

- © 2017 reinforce.io
- contact@reinforce.io
-