Imagination-Augmented Agents for Deep Reinforcement Learning

Sébastien Racanière
Théophane Weber*, David Reichert*
&
Lars Buesing, Arthur Guez, Danilo Rezende, Adria Badia, Oriol Vinyals,
Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia,
Demis Hassabis, David Silver, Daan Wierstra

DeepMind

Plan

What I will cover:

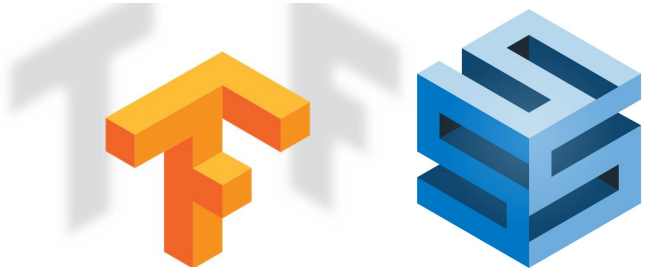
- Building of an environment model
- Building of I2A
- Training loss for rollout policy

What I will not cover:

- RL training of the agent

TF + Sonnet

- Sonnet Library built on top of TF
- Handles variable sharing transparently
- Object-oriented
  - Explicitly represent submodules
  - Configure then connect

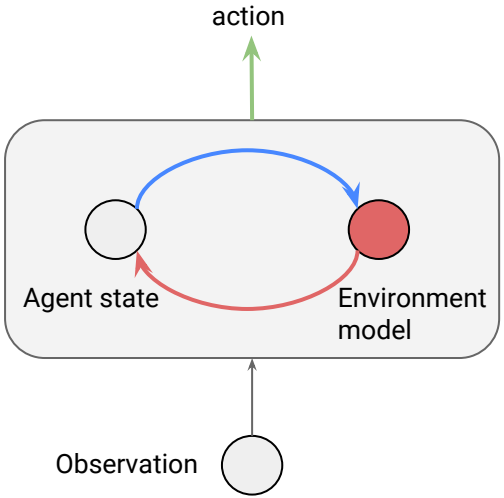


```python
import sonnet as snt

# Provide your own functions to generate data Tensors.
train_data = get_training_data()
test_data = get_test_data()

# Construct the module, providing any configuration necessary.
linear_regression_module = snt.Linear(output_size=FLAGS.output_size)

# Connect the module to some inputs, any number of times.
train_predictions = linear_regression_module(train_data)
test_predictions = linear_regression_module(test_data)
```

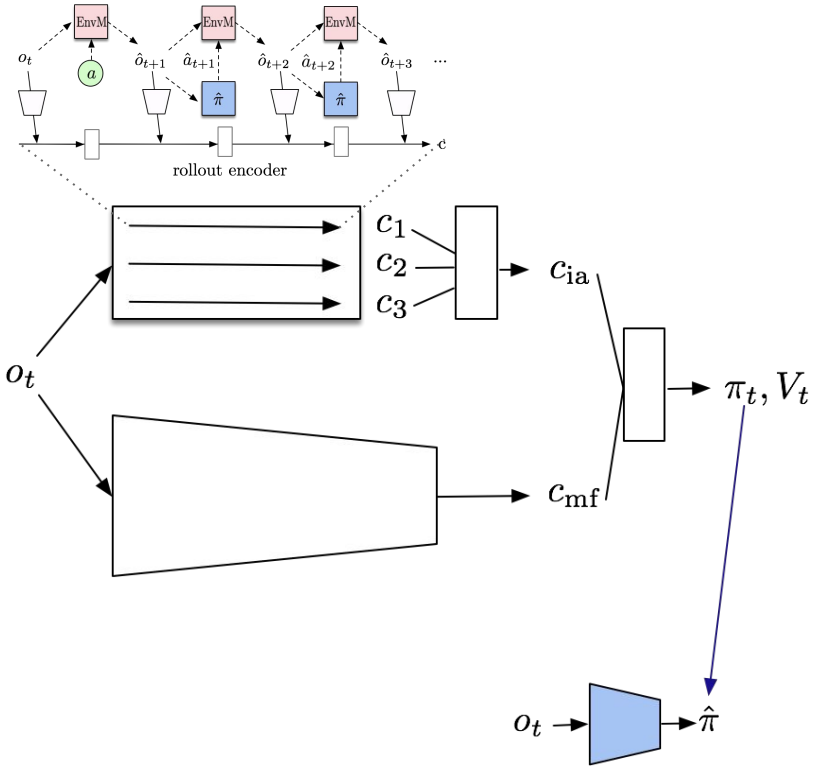
The I2A Agent

We introduce Imagination-Augmented Agent, a model combining model-based and model-free aspects:

- The agent learns a model of the world, queries it for information, and interprets the predictions in order to act.

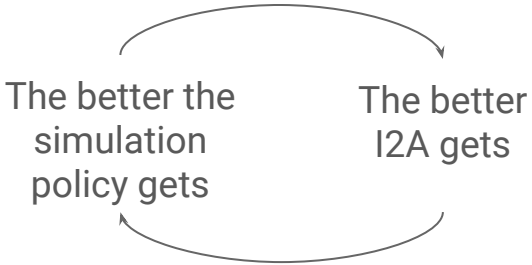


Building I2A block by block



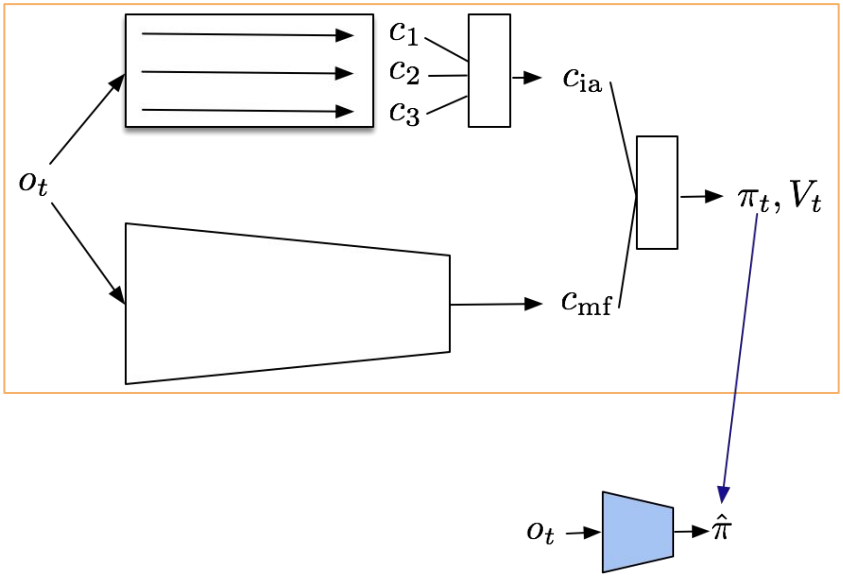
How to choose the imagination policy?

- Random
- Distillation



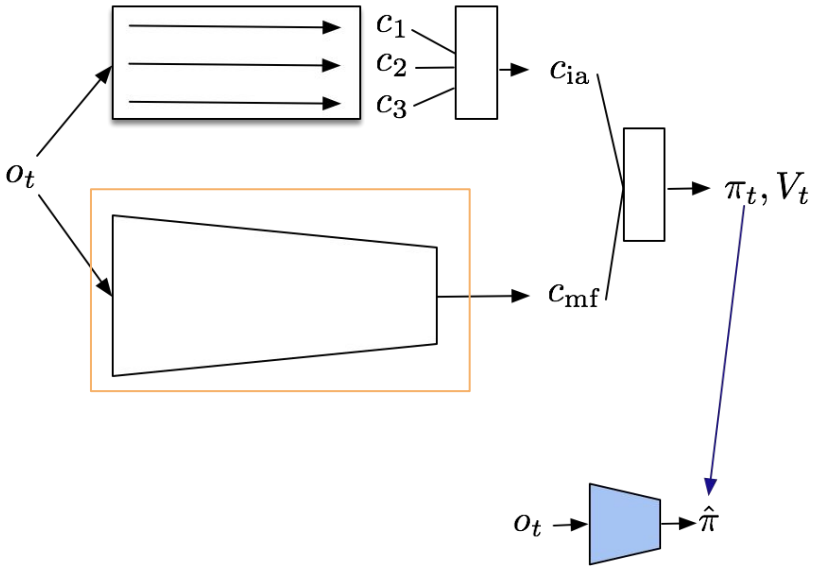
I2A is a form of learned policy improvement

Building I2A block by block




```
class I2AAgent(snt.RNNCore):
  def __init__(self, num_actions, model_free_path, imag_path,
               height, width, stack_size, name='i2a_agent'):
    …

  def _build(self, frame, prev_state):
    frame = tf.expand_dims(frame, axis=0)
    next_state = update_frame_stack(prev_state, frame)

    imag_feature = self.imag_path(next_state)
    model_free_feature = self.model_free_path(next_state)
    policy_feature = tf.concat(
       [imag_feature, model_free_feature], axis=1)

    value_and_logits = snt.Linear(
       output_size=1 + self.num_actions,
       name='value_and_logits')(policy_feature)
    baseline=value_and_logits[:, 0]
    policy_logits=value_and_logits[:, 1:]
    action = tf.multinomial(policy_logits, 1)
    action = tf.cast(action, tf.int32)
    return (action, policy_logits, baseline), next_state
```

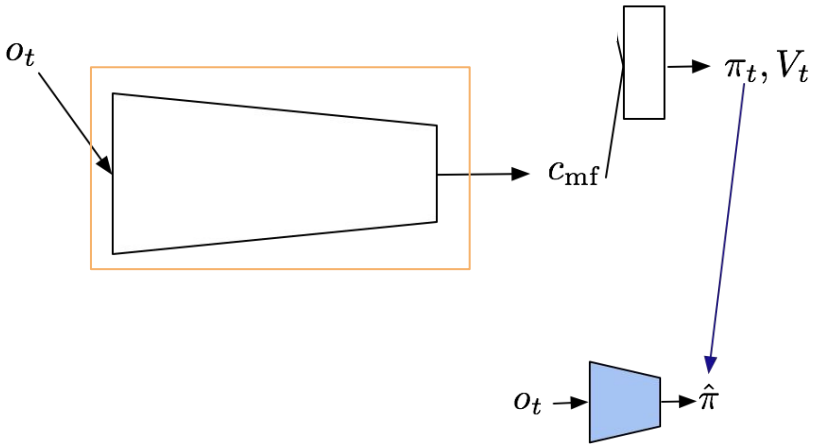
Building I2A block by block



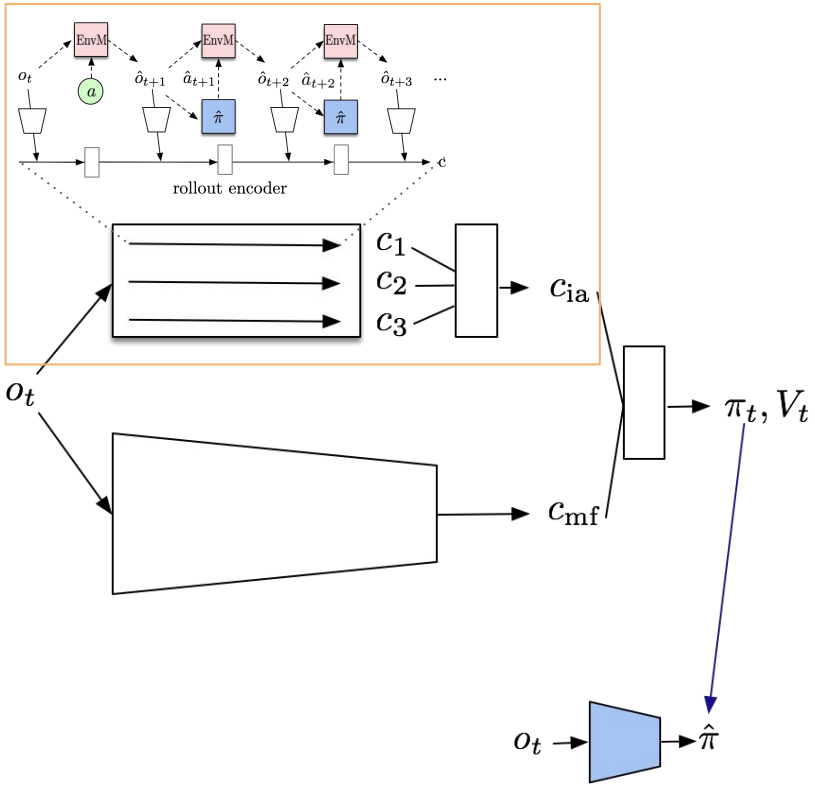

```
class FrameProcessing(snt.AbstractModule):

 def __init__(self, output_size,  name='frame_processing'):
  super(FrameProcessing, self).__init__(name=name)
  self.output_size = output_size

 def _build(self, frame):
  hidden = snt.Conv2D(
     output_channels=16, kernel_shape=3, stride=1)(frame)
  hidden = tf.nn.relu(hidden)
  hidden = snt.Conv2D(
     output_channels=16, kernel_shape=3, stride=2)(hidden)
  hidden = tf.nn.relu(hidden)
  hidden = snt.Linear(self.output_size)(snt.BatchFlatten()(hidden))
  return tf.nn.relu(hidden)
```

Building I2A block by block

```
class NoImagPath(snt.AbstractModule):

  def __init__(self, name='no_imag_path'):
    super(NoImagPath, self).__init__(name=name)

  def _build(self, frame_stack):
    batch_size = frame_stack.get_shape()[0].value
    return tf.zeros((batch_size, 0))
```

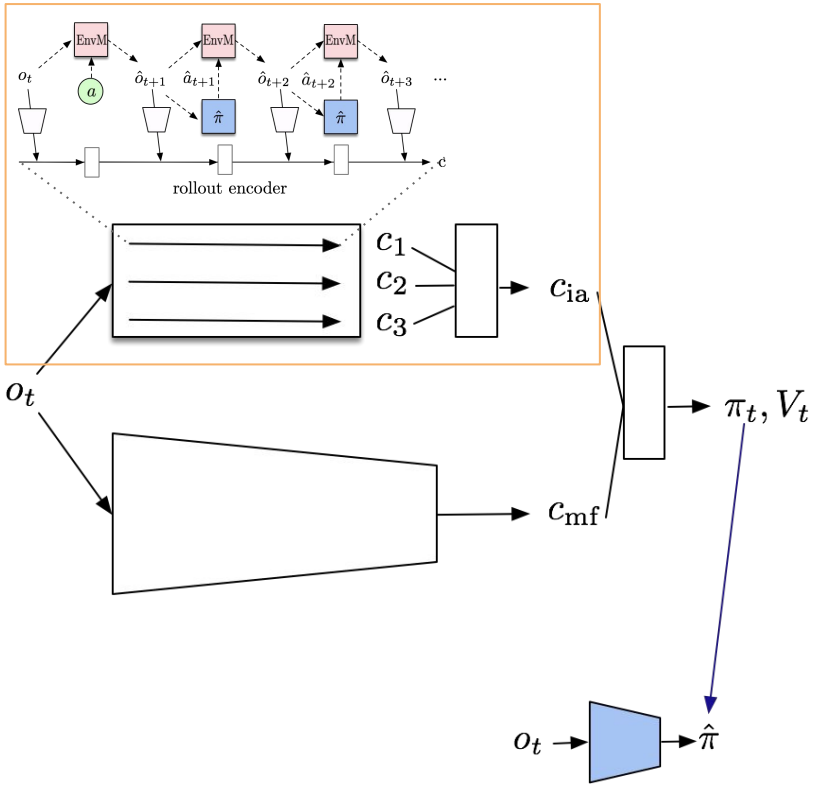


Building I2A block by block



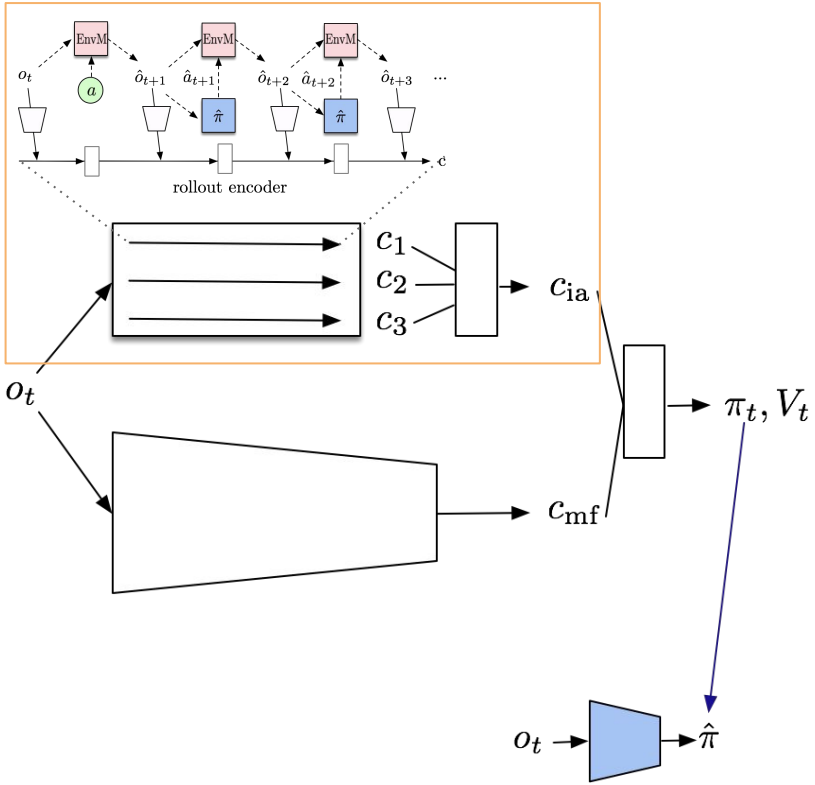

```
class ImagPath(snt.AbstractModule):
  def __init__(self, num_actions, rollout_depth,
               env_model, rollout_policy, single_imag_feature,
               name='imag_path'):
    ...

  def _tile_by_actions(self, tensor):
    return tf.concat(
      [tf.expand_dims(tensor, axis=1)] * self.num_actions, axis=1)
```

Building I2A block by block



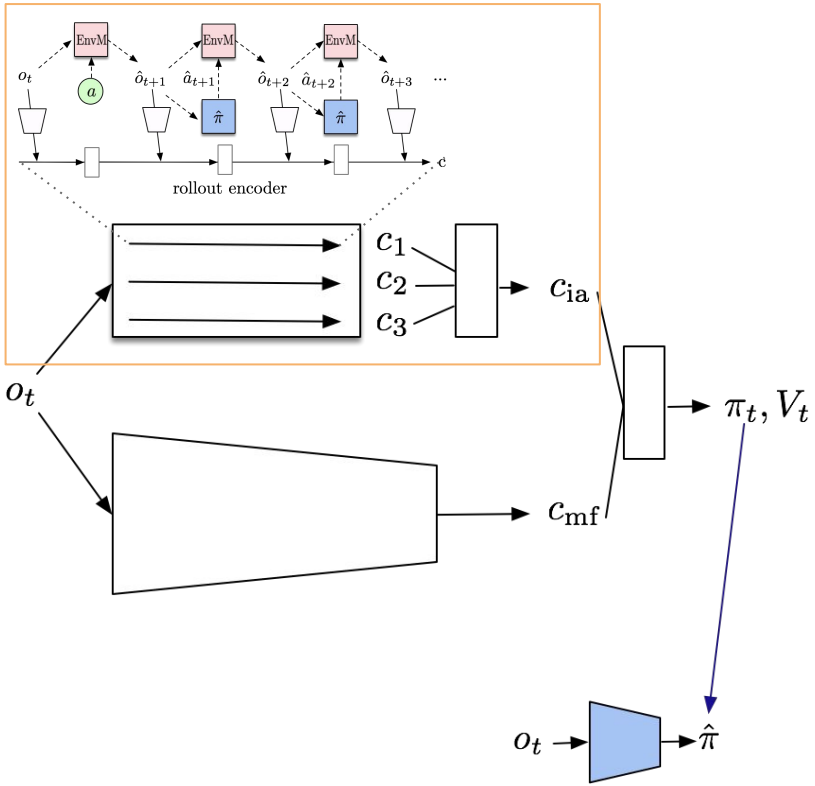

```
class ImagPath(snt.AbstractModule):
  def __init__(self, num_actions, rollout_depth,
               env_model, rollout_policy, single_imag_feature,
               name='imag_path'):
    ...

  def _tile_by_actions(self, tensor):
    return tf.concat(
        [tf.expand_dims(tensor, axis=1)] * self.num_actions, axis=1)

  def _build(self, frame_stack):
    batch_size = frame_stack.get_shape()[0].value
    frame_stack = self._tile_by_actions(frame_stack)
    imag_features = []
    ...
```

Building I2A block by block



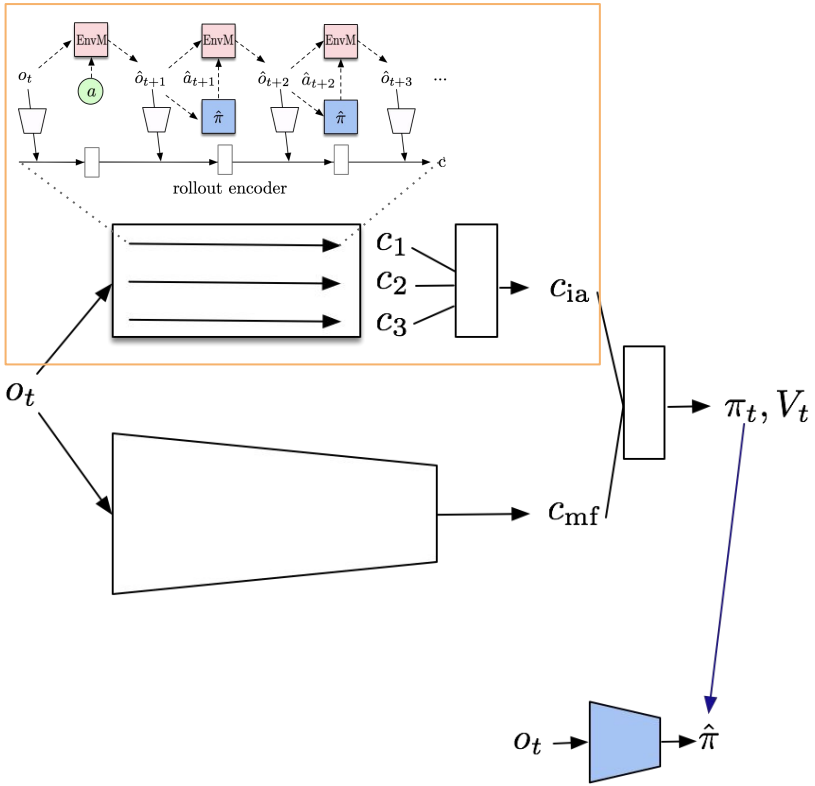

```
class ImagPath(snt.AbstractModule):
  def __init__(self, num_actions, rollout_depth,
               env_model, rollout_policy, single_imag_feature,
               name='imag_path'):
    ...

  def _tile_by_actions(self, tensor):
    return tf.concat(
      [tf.expand_dims(tensor, axis=1)] * self.num_actions, axis=1)

  def _build(self, frame_stack):
    ...
    action = tf.constant(range(self.num_actions),
                         shape=(self.num_actions,),
                         dtype=tf.int32)
    action = tf.stack([action] * batch_size)
    ...
```

Building I2A block by block



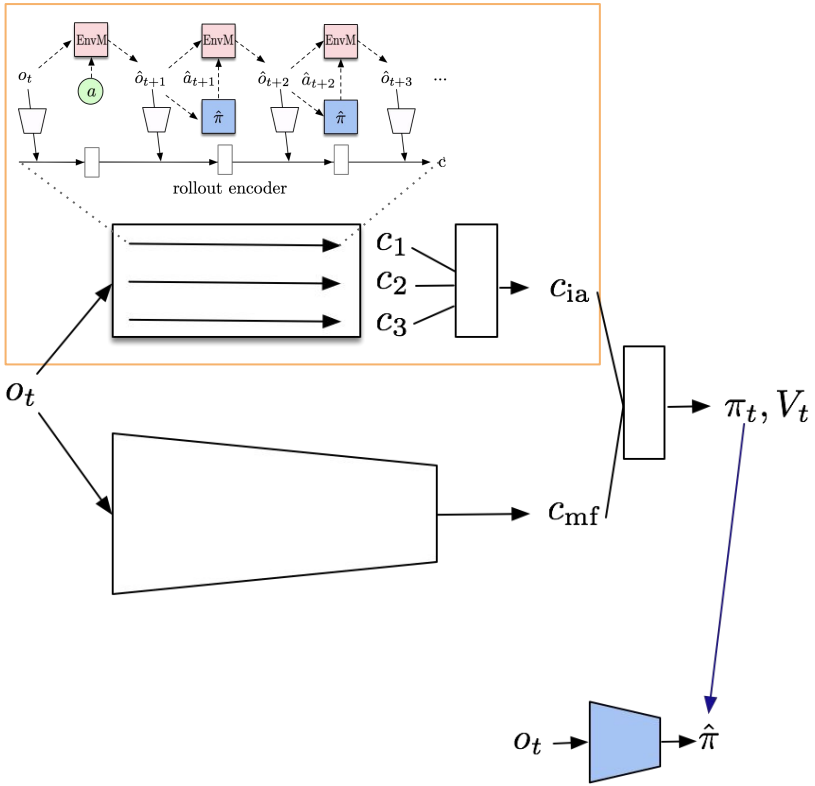

```
class ImagPath(snt.AbstractModule):
  def __init__(self, num_actions, rollout_depth,
               env_model, rollout_policy, single_imag_feature,
               name='imag_path'):
    ...

  def _tile_by_actions(self, tensor):
    return tf.concat(
        [tf.expand_dims(tensor, axis=1)] * self.num_actions, axis=1)

  def _build(self, frame_stack):
    ...
    env_model = snt.BatchApply(self.env_model)
    imag_feature = snt.BatchApply(self.single_imag_feature)
    frame, reward, _ = env_model(frame_stack, action)
    frame_stack = update_frame_stack(frame_stack, frame)
    imag_features.append(imag_feature(frame, reward, action))
    ...
```

Building I2A block by block



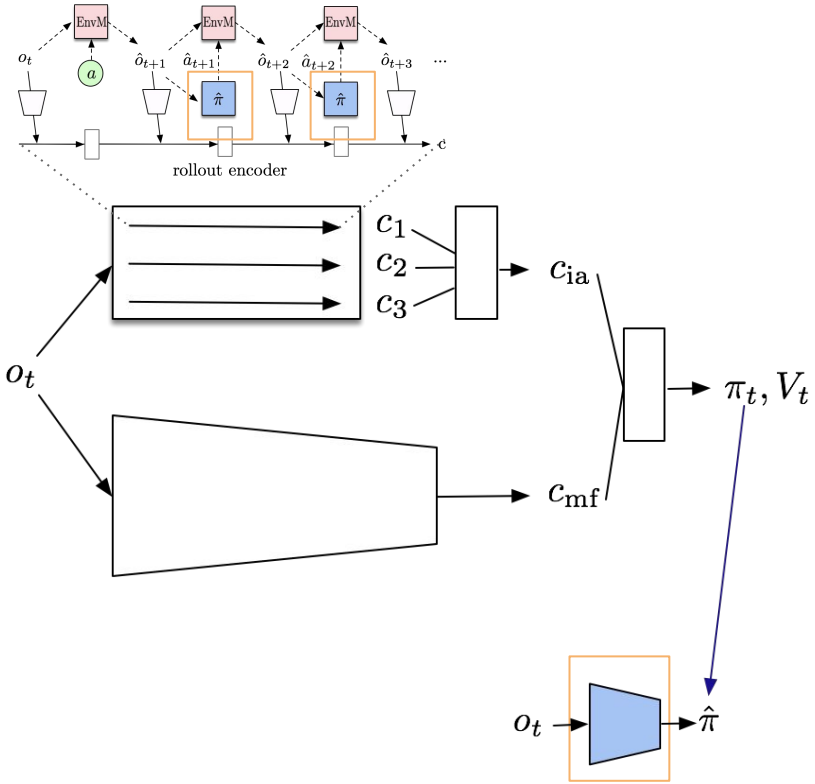

```
class ImagPath(snt.AbstractModule):
  def __init__(self, num_actions, rollout_depth,
               env_model, rollout_policy, single_imag_feature,
               name='imag_path'):
    ...

  def _tile_by_actions(self, tensor):
    return tf.concat(
        [tf.expand_dims(tensor, axis=1)] * self.num_actions, axis=1)

  def _build(self, frame_stack):
    ...
    for _ in range(self.rollout_depth - 1):
      action, _ = snt.BatchApply(self.rollout_policy)(frame)
      frame, reward, _ = env_model(frame_stack, action)
      frame_stack = update_frame_stack(frame_stack, frame)
      imag_features.append(imag_feature(frame, reward, action))
    ...
```

Building I2A block by block



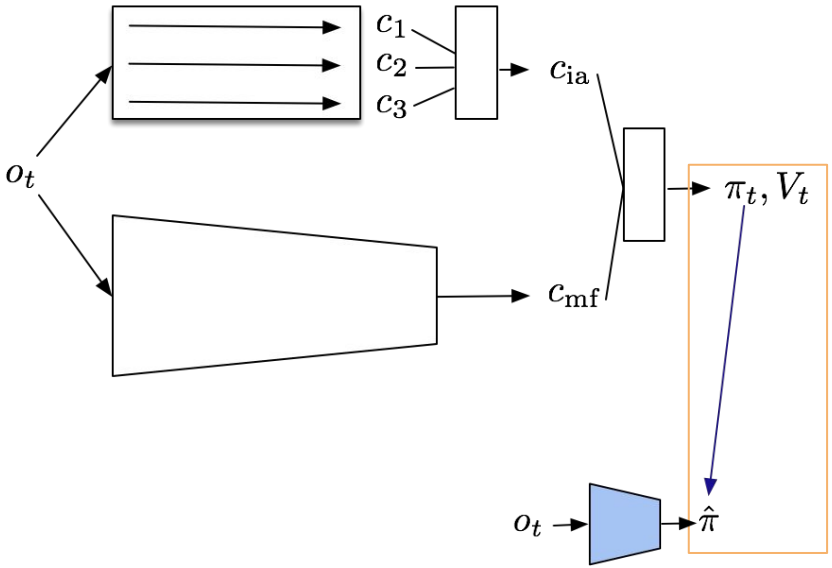

```
class ImagPath(snt.AbstractModule):
  def __init__(self, num_actions, rollout_depth,
               env_model, rollout_policy, single_imag_feature,
               name='imag_path'):
    ...

  def _tile_by_actions(self, tensor):
    return tf.concat(
        [tf.expand_dims(tensor, axis=1)] * self.num_actions, axis=1)

  def _build(self, frame_stack):
    ...
    lstm = snt.LSTM(256)
    lstm_state = [self._tile_by_actions(t) for t in
                  lstm.initial_state(batch_size)]
    for feature in imag_features[::-1]:
      lstm_output, lstm_state = snt.BatchApply(lstm)(
          feature, lstm_state)
    encoded_rollouts = lstm_output
    return tf.reshape(encoded_rollouts, [1, -1])
```

Building I2A block by block



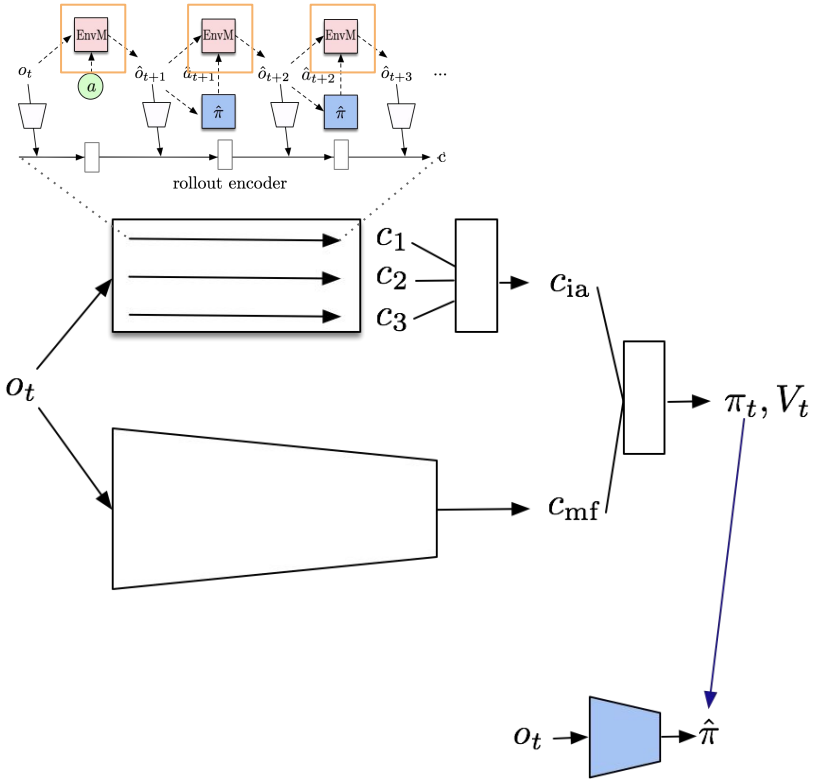

```python
class RolloutPolicy(snt.AbstractModule):

  def __init__(self, num_actions, hidden_size, name='rollout_policy'):
    super(RolloutPolicy, self).__init__(name=name)
    self.num_actions = num_actions
    self.hidden_size = hidden_size

  def _build(self, frame):
    frame_feature = FrameProcessing(self.hidden_size)(frame)
    logits = snt.Linear(
        output_size=self.num_actions,
        name='logits')(frame_feature)
    action = tf.multinomial(logits, 1)
    action = tf.cast(action, tf.int32)
    return tf.squeeze(action, axis=1), logits
```

Building I2A block by block



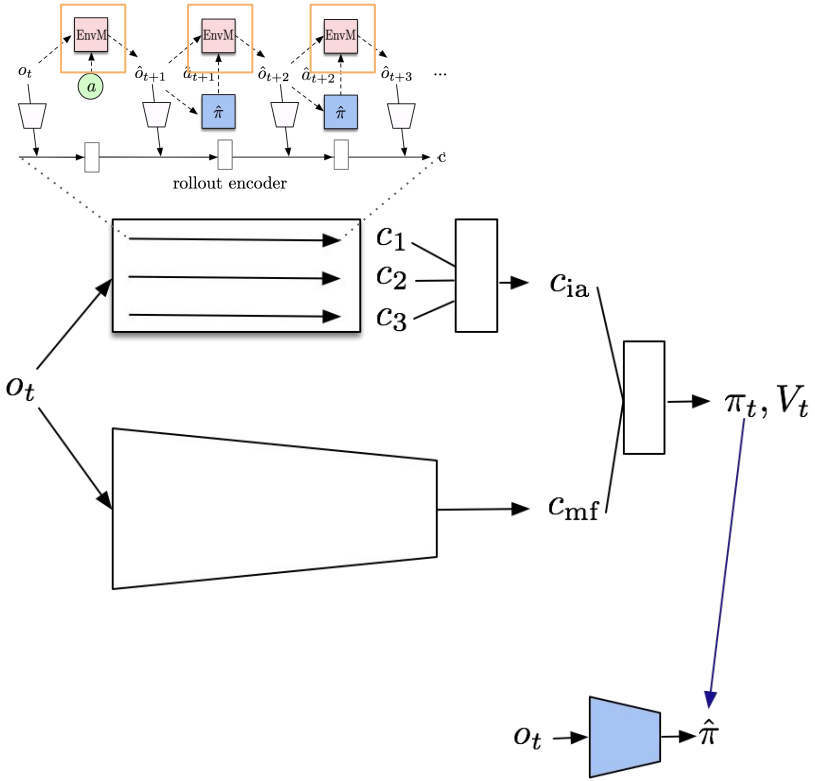

```
_, rollout_logits = rollout_policy(
    tf.expand_dims(input_frame, axis=0))
distill_loss = tf.nn.softmax_cross_entropy_with_logits(
    logits=rollout_logits,
    labels=tf.stop_gradient(tf.nn.softmax(policy_logits)))
optim_step = tf.train.RMSPropOptimizer(
    learning_rate=learning_rate, epsilon=0.1).minimize(distill_loss)
```

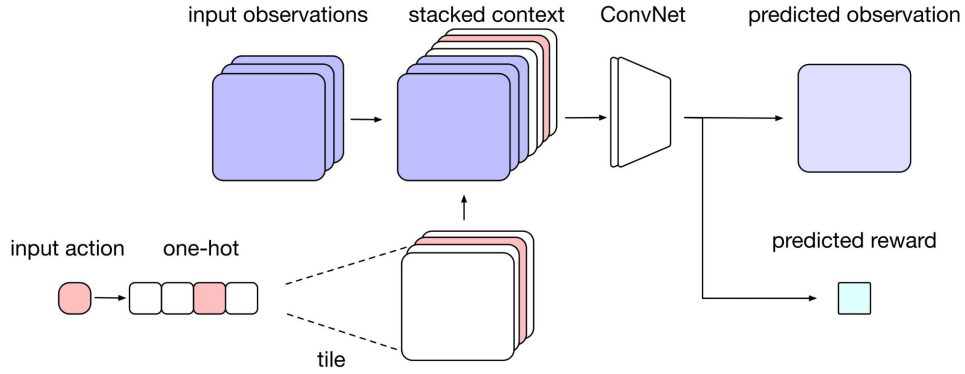
Building I2A block by block



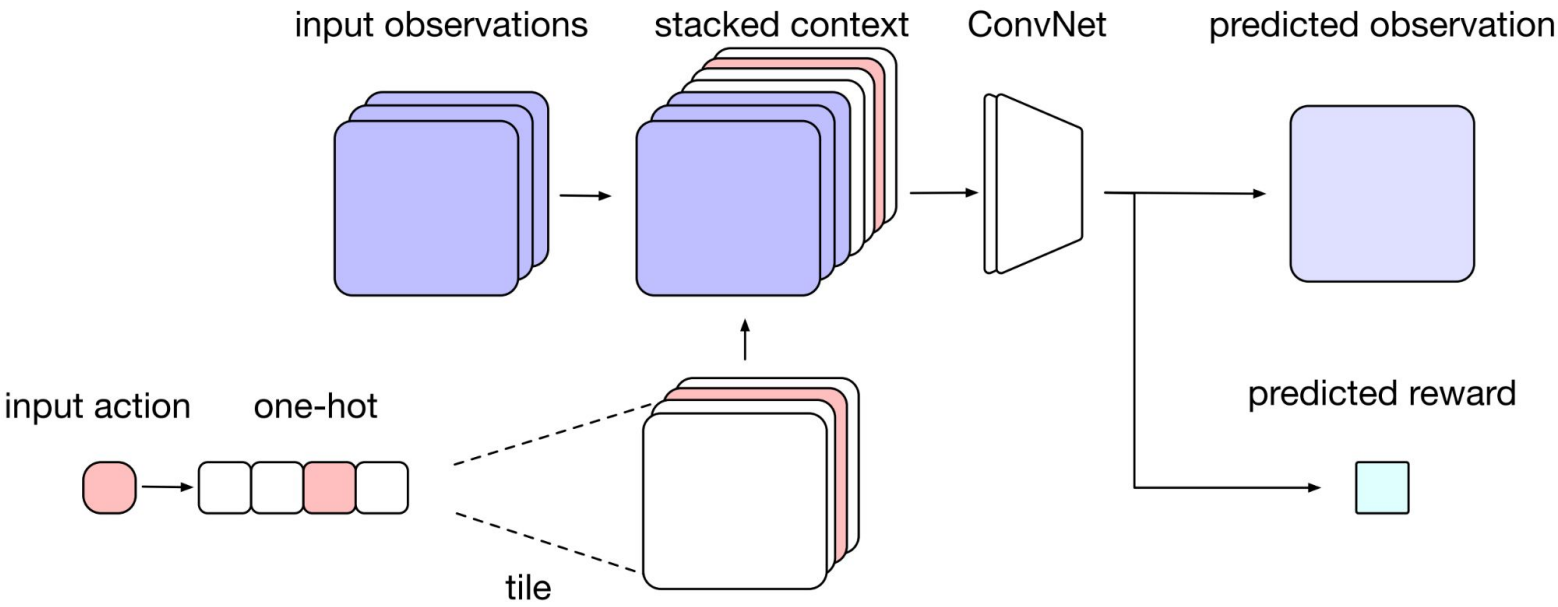

```python
class CopyModel(snt.AbstractModule):
  def __init__(self, num_reward_bins, name='copy_model'):
    super(CopyModel, self).__init__(name=name)
    self.num_reward_bins = num_reward_bins

  def _build(self, frame_stack, action):
    dummy_reward = tf.constant(
        0, shape=(frame_stack.get_shape()[0],), dtype=tf.float32)
    dummy_reward_bins = tf.constant(
        0, shape=(frame_stack.get_shape()[0], self.num_reward_bins),
        dtype=tf.float32)
    last_frame = frame_stack[:, :, :, -3:]
    return last_frame, dummy_reward, dummy_reward_bins
```

Building I2A block by block





The Environment Model



Conclusion

- Use TF + Sonnet
  - modularity
  - variable sharing
- Modular construction of I2A
  - ImagPath / NoImagPath
  - CopyModel / SizePreservingConvNetModel
  - could also do RolloutPolicy / RandomPolicy
  - Rapid turnaround of experiments

Thank you