1709.04905 - One-Shot Visual Imitation Learning via Meta-Learning

- **Yunqiu Xu**

- Other reference:
  - Presentation: https://www.youtube.com/watch?v=_9Ny2ghjwuY
  - Code: https://github.com/tianheyu927/mil

---

1. Introduction

- Challenge: learning each skill from a scratch is infeasible
- One-Shot Visual Imitation Learning via Meta-Learning
  - Reuse past experience to train the "base model", then adapt it to new task with only a single demonstration
  - Visual: use raw visual inputs
  - Meta-Learning: MAML C. Finn et.al. 2017
- Prior work : take task identity / demonstration as the input into a contextual policy
- Our work : learn parameterized policy, then adapt into new tasks through a few gradient updates

2. Related work

- In this work, the state of environment is unknown $\rightarrow$ we feed raw sensory inputs to learn it
- Two challenges for learning from demonstrations then applying it to real world:
  - Compounding errors: not in this work
  - **The need of a large number of demonstrations for each task**
- Why don't use Inverse RL:
  - How does it work : recover reward function from demonstrations

- Pros: reduce demonstrations, better than behavioral cloning
- Cons:
  - Requires additional robot experience to optimize the reward C. Finn et.al. 2016
  - Hard to evaluate learned reward, especially for high-dim data (image)
  - Gan-based IRL (e.g. GAIL) is hard to train
- How do we reduce the demonstrations: **share data across tasks**
  - First, use a dataset of demonstrations of many other tasks for meta learning, in this way we can build a base model
  - Then we can adapt this base model to new task with only a few demonstrations
  - Meta-learning is similar to transfer learning to some extent, the different is not the transfer on dataset, but the transfer on task
  - Take a simple instance, if the robot is learned to pick apple, orange, pear ... then it can pick peach easily

3. Meta-Imitation Learning

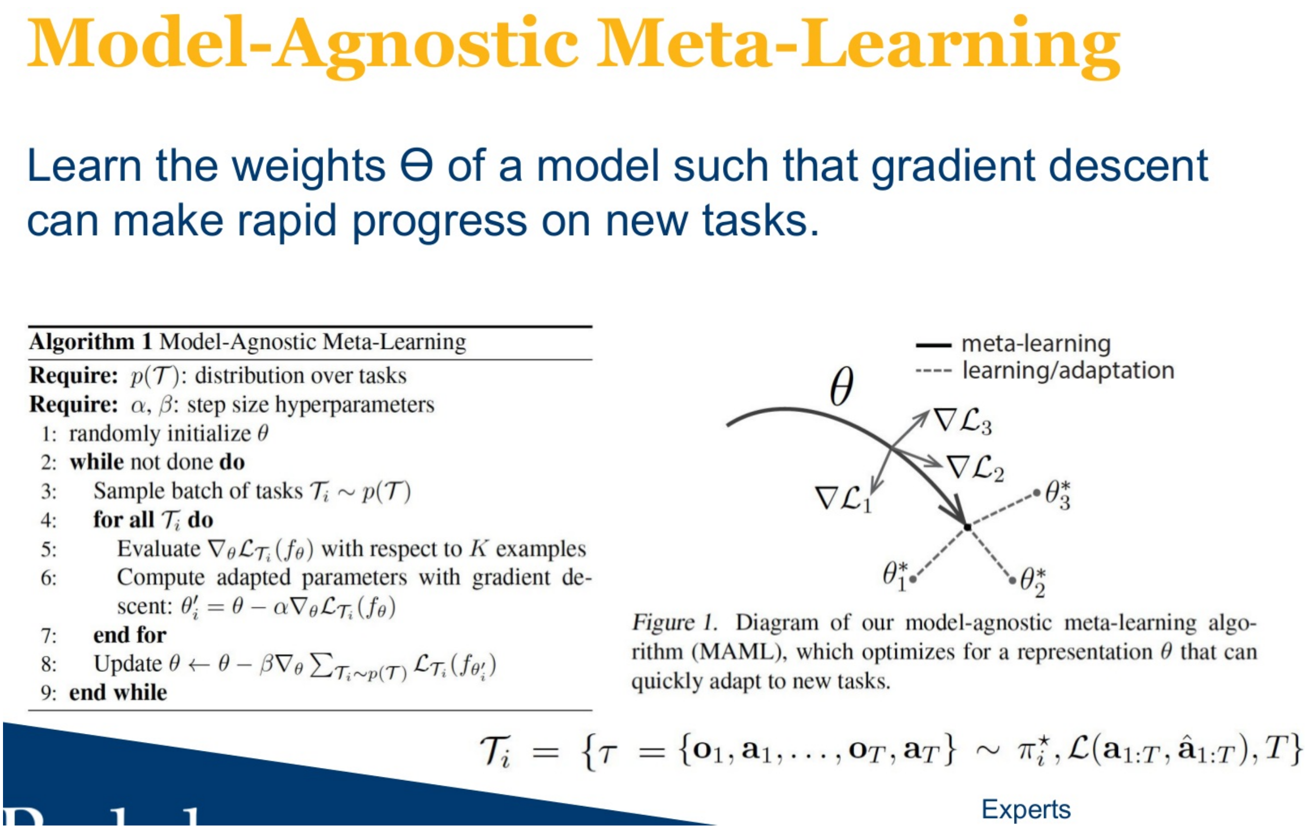
- Goal : learn a policy that can quickly adapt to new tasks from a single demonstration of that task
- Each imitation task $T_i = \left\{\tau = \{o_1, a_1, \ldots, o_T, a_T\} \sim \pi_i^*, L(a_{1:T}, \hat{a}_{1:T}), T\right\}$
  - τ : a demonstration generated by policy π_i^*
  - $L(a_1, \ldots, a_T, \hat{a}_1, \ldots, \hat{a}_T) \rightarrow R$: loss function to give feedback
  - **Note that this form is different from original MAML**

3.1 MAML

- Consider a policy π with parameter vector θ
- Sample a task T_i from $p(T)$
- Train this task with K samples (adapt π to T_i to get new parameter θ')
- Test this task, then treat the testing error as the training error of meta-process (Use $\theta'_1, \ldots, \theta'_n$, to update θ)
- Meta objective:

$$\min_{\theta} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta'_i}) = \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta - \alpha \nabla_\theta L_{T_i}(f_\theta)}) \qquad (1)$$

- Finally, you can adapt trained π to a new task with only a few data / gradient updates



3.2 Extend MAML to MIL

- o_t is the observation at time t, i.e. an image, while a_t is the action
- For demonstration trajectory τ, we use MSE to compute loss:

$$L_{T_i}(f_\phi) = \sum_{\tau_j \sim T_i} \sum_t ||f_\phi(o_t^{(j)}) - a_t^{(j)}||_2^2 \qquad (2)$$

- Meta-training:
  - Assume each training task has at least 2 demonstrations, thus we can sample a set of tasks with **two demonstrations per task**
  - For each task T_i, train θ'_i with its one demonstration τ_i $\rightarrow$ inner loop of meta-learning
  - Use another demonstration τ'_i to "test" θ'_i , i.e. check the mse of predicted actions and demonstration actions

- Then we can update θ according to the gradient of meta-objective
- As we get a series of $\theta_i' s$ and their testing error, we can update θ
- Finally we can get trained parameters θ for meta-learner

Algorithm 1 Meta-Imitation Learning with MAML

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
randomly initialize θ
while not done **do**
    Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
    for all $\mathcal{T}_i$ **do**
        Sample demonstration $\tau = \{\mathbf{o}_1, \mathbf{a}_1, ...\mathbf{o}_T, \mathbf{a}_T\}$ from $\mathcal{T}_i$
        Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using τ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (2)
        Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
        Sample demonstration $\tau_i' = \{\mathbf{o}_1', \mathbf{a}_1', ...\mathbf{o}_T', \mathbf{a}_T'\}$ from $\mathcal{T}_i$ for the meta-update
    end for
    Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each τ_i' and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 2
end while
return parameters θ that can be quickly adapted to new tasks through imitation.

- Meta-testing:
  - Sample a new task T and its one demonstration
  - This task can involve new goals or manipulating new, previously unseen objects
  - Then we can adapt θ to this task

3.3 Two Head Structure

- Why use this: more flexibility during adapting
- The parameters of pre-update head are not used for post-update head in final
- Modification : parameters of final layers are not shared, forming two heads
  - Change loss function as:

$$L_{T_i}(f_\phi) = \sum_{\tau_j \sim T_i} \sum_t ||W y_t^{(j)} + b - a_t^{(j)}||_2^2 \qquad (3)$$

    - $y_t^{(j)}$: post-synamptic activations of the last hidden layer
    - W, b : weights and bias for last layer
  - Then the meta-objective is about θ, W, b

$$\min_{\theta, W, b} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'}) = \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta - \alpha \nabla_\theta L_{T_i}(f_\theta)}) \qquad (4)$$

3.4 Learning to Imitate without Expert Actions

- Why use this : it is more practical to simply provide a video of the task being performed
- We just simplify this problem by simplify the loss function as

$$L_{T_i}(f_\phi) = \sum_{\tau_j \sim T_i} \sum_t ||W y_t^{(j)} + b||_2^2 \qquad (3)$$

- **This can be a future question for more robust loss function**

4. Network Architecture

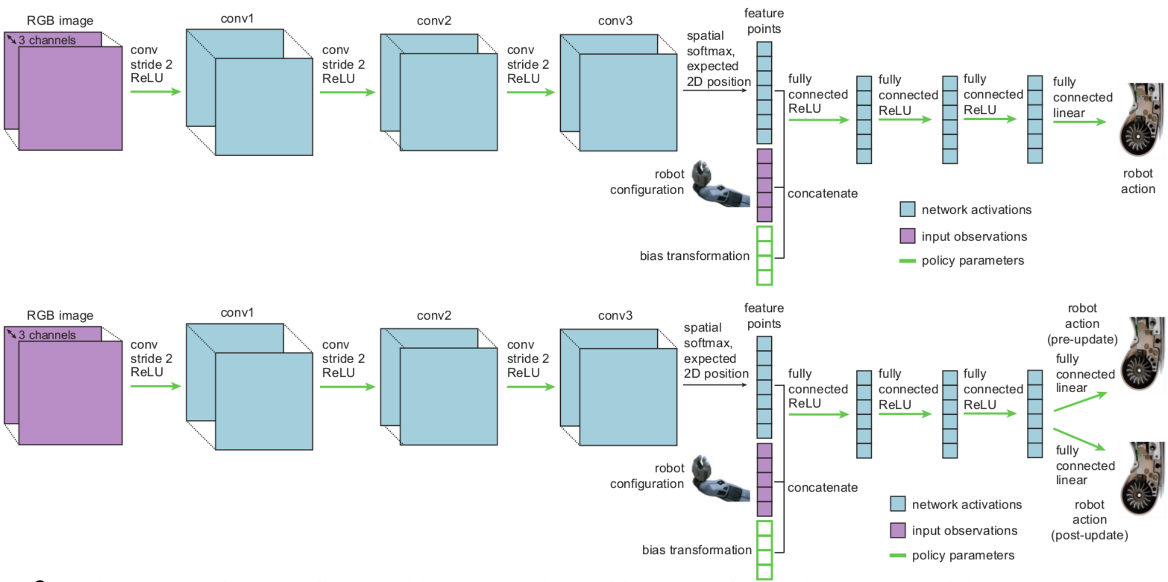


Figure 2: Diagrams of the policy architecture with a bias transformation (top and bottom) and two heads (bottom). The green arrows and boxes indicate weights that are part of the meta-learned policy parameters θ.

- **Layer normalization** after each layer
  - Data within a demonstration trajectory is highly correlated across time
  - Thus BN was not effective
- Bias transformation $\rightarrow$ improve the performance of meta-learning
  - Concatenate a vector of parameters to a hidden layer of post-synaptic activations
  - Thus vector is treated as same as other parameters during meta-learning and final testing

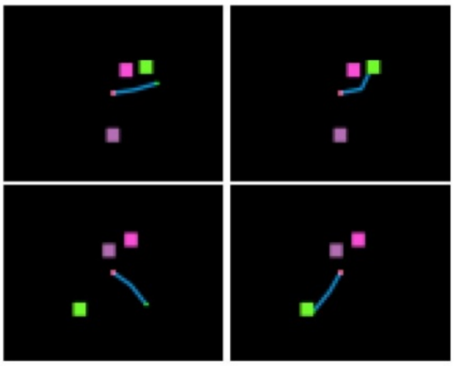
$$y = Wx + b \ \rightarrow \ y = W_1 x + W_2 z + b$$

        - z is the parameter vector form bias transformation
        - $W = [W_1, W_2]$
    - This modification **increases the representational power of the gradient**
    - Does not affect the representation power of the network itself

4. Experiment

- Questions:
    - Can a policy be learned to maps from image pixels to actions using a single demonstration of a task
    - How does our meta-imitation learning method compare to other one-shot imitation learning methods
    - Can we learn without expert actions
    - How well does our method scale to real world tasks
- Methods for comparison:
    - Our method
    - Random policy: output random actions from standard normal distribution
    - Contextual policy:
        - Input the final image of demonstration
        - Indicate goal and current image (observation)
        - Then output current action
    - LSTM:
        - Input demonstration and current observation
        - Output current action
    - LSTM + attention: only applicable to non-vision tasks
- Task 1 : simulated reaching
    - Goal: reach a target of a particular color
    - Both vision and non-vision versions are tested
    - meta-learning can handle raw-inputs
    - Our method can handle small dataset (demonstration) well
    - Bias transformation (bt) can perform more consistently across dataset sizes

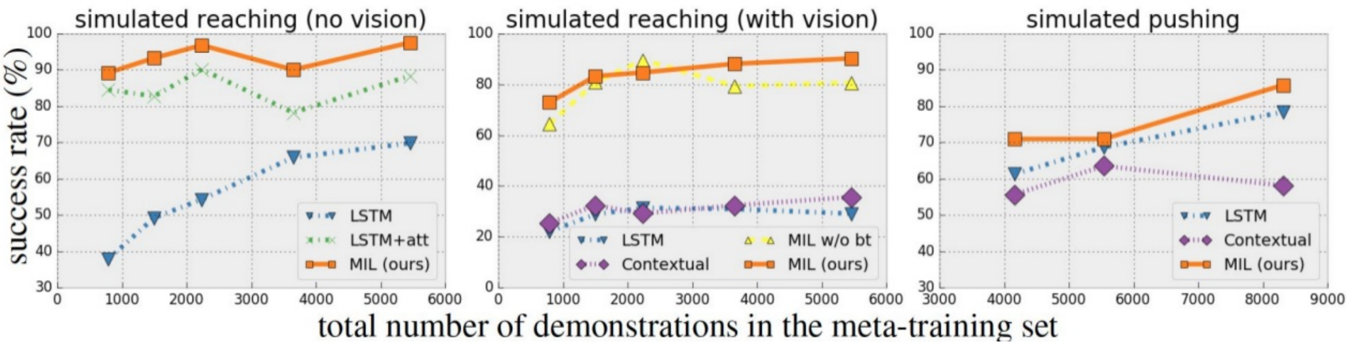
Simulated reaching



Simulated reaching

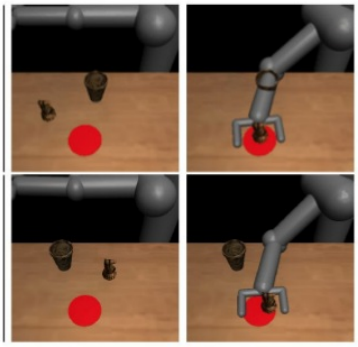
Task:

- reaching a target of a particular color
- Policy must learn to localize the target using the demonstration and generalize to new positions
- Meta-training must learn to handle different colors
- 150 tasks x 10 different trials per task



- Task 2 : simulated pushing

Simulated pushing

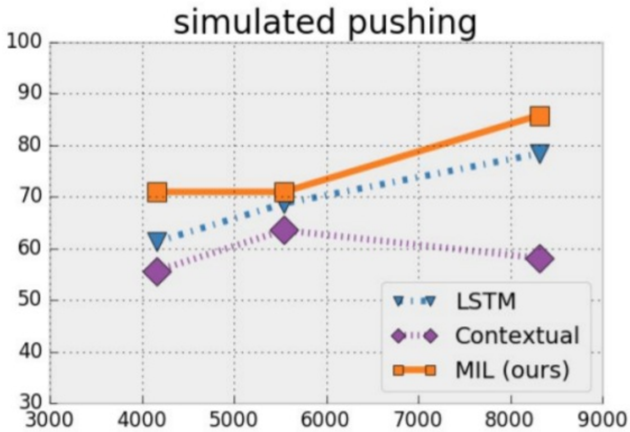


Task:

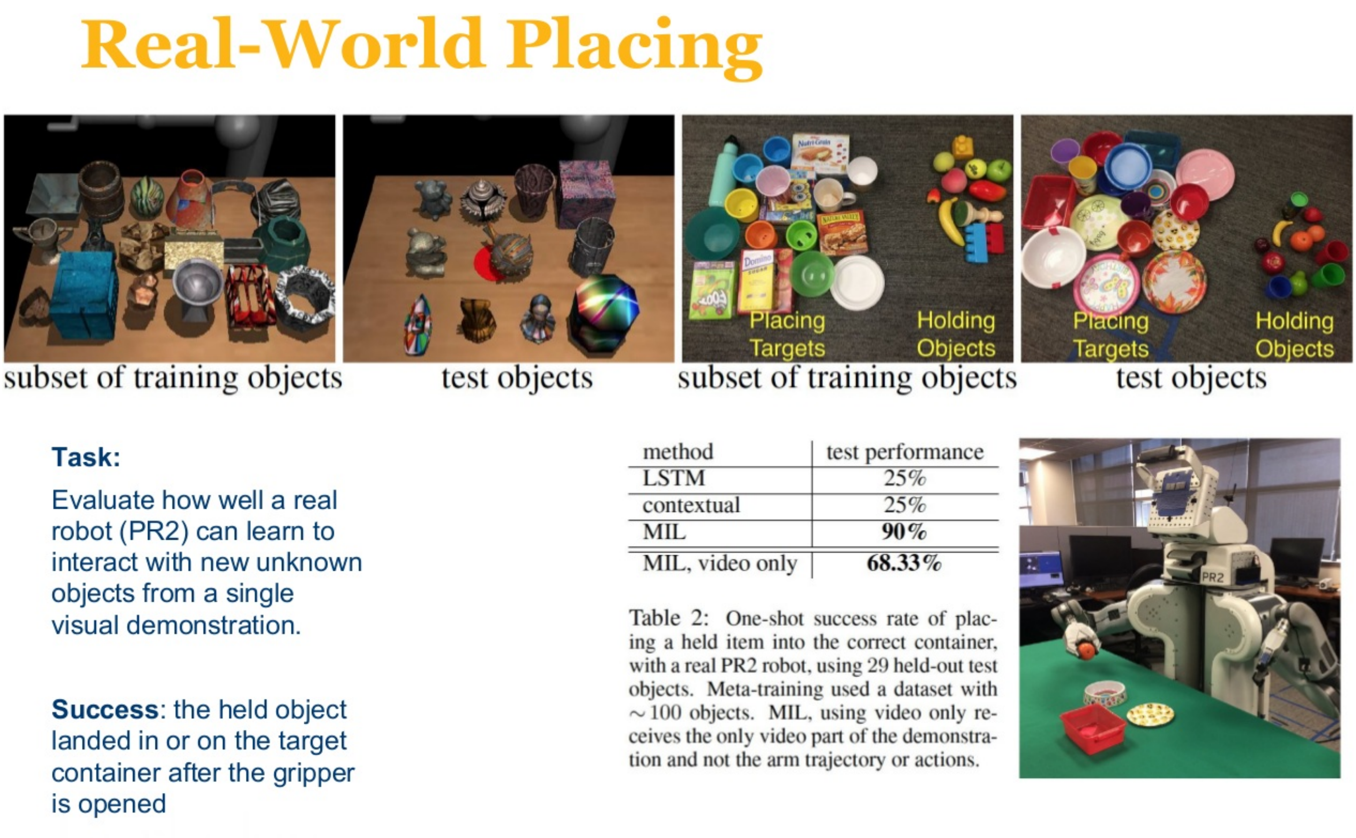
- ❏ A push is considered as success if the center of the target object lands on the red target circle for at least 10 timestamps.
- ❏ Each task is defined as pushing a particular objects
- ❏ 74 tasks x 6 different trials per task

method		video+state +action	video +state	video
LSTM	1-shot	78.38%	37.61%	34.23%
contextual	1-shot	n/a	58.11%	56.98%
MIL (ours)	1-shot	**85.81%**	**72.52%**	**66.44%**
LSTM	5-shot	83.11%	39.64%	31.98%
contextual	5-shot	n/a	64.64%	59.01%
MIL (ours)	5-shot	**88.75%**	**78.15%**	**70.50%**

Table 1: One-shot and 5-shot simulating pushing success rate with varying demonstration information provided at test-time. MIL can more successfully learn from a demonstration without actions and without robot state and actions than LSTM and contextual policies.



- Task 3 : real-world placing
  - Experiment goal : place a held item into a target container, such as a cup, plate, or bowl, while ignoring two distractors



5. Summary and Ongoing Work

- Summary:
  - reuse prior experience when learning in new settings
  - learning-to-learn enables effective one-shot learning
- Ongoing work: one-shot imitation from human video → during demo, let human put the ball in cup

6. Code

6.1. `main()` function in `main.py`

- I will focus on training part, while validation and testing part will be similar
- In this function **data_generator is used twice**: network initialization and model training
- **Why mention this: during training these 2 generated parts will be concated**

- Network initialization: performed before training
  - Get `train_image_tensors` via data_generator
  - Get `inputa`, `inputb` to shape `train_input_tensors`
  - network initialization

```
# build train_input_tensors
train_image_tensors = data_generator.make_batch_tensor(network_config, restore_iter=FLAGS.restore_iter)
inputa = train_image_tensors[:, :FLAGS.update_batch_size*FLAGS.T, :]
inputb = train_image_tensors[:, FLAGS.update_batch_size*FLAGS.T:, :]
train_input_tensors = {'inputa': inputa, 'inputb': inputb}
# build val_input_tensors, simillar to above
...
val_input_tensors = ...
# network initialization
model.init_network(graph, input_tensors=train_input_tensors, restore_iter=FLAGS.restore_iter)
model.init_network(graph, input_tensors=val_input_tensors, restore_iter=FLAGS.restore_iter, prefix='Validation_')
```

- Training:
  - After initialize network we will perform training, here data_generator will be called again in each iteration
  - Once an iter is ended we can get `result` to update `prelosses` and `postlosses`

```
# build training data
state, tgt_mu = data_generator.generate_data_batch(itr)
statea = state[:, :FLAGS.update_batch_size*FLAGS.T, :]
stateb = state[:, FLAGS.update_batch_size*FLAGS.T:, :]
actiona = tgt_mu[:, :FLAGS.update_batch_size*FLAGS.T, :]
actionb = tgt_mu[:, FLAGS.update_batch_size*FLAGS.T:, :]
feed_dict = {model.statea: statea, model.stateb: stateb, model.actiona: actiona, model.actionb: actionb}
input_tensors = [model.train_op]
# get result
results = sess.run(input_tensors, feed_dict=feed_dict)
prelosses.append(results[-2])
train_writer.add_summary(results[-3], itr)
postlosses.append(results[-1])
```

6.2. `construct_model()` in `mil.py`

- This is similar to `construct_model()` of maml.py
- Suffix 'a' is for inner training and suffix 'b' is for inner testing
- **Difference 1: concat two parts of input**:
  - `obs`: the input data we generate during network initialization
  - `state` : the input data we generate during training
  - **此处存疑, 我感觉这两个不会同时出现, 应该总有一个是placeholder**

```
# if these item does not exist --> placeholder
# inputb is similar
self.obsa = obsa = input_tensors['inputa'] # network initialization
statea = self.statea # training
actiona = self.actiona # training
inputa = tf.concat(axis=2, values=[statea, obsa])
```

- Convert to image dims

```
inputa, _, state_inputa = self.construct_image_input(inputa, self.state_idx, self.img_idx, network_config=network_config)
if FLAGS.zero_state:
    state_inputa = tf.zeros_like(state_inputa)

inputb, flat_img_inputb, state_inputb = self.construct_image_input(inputb, self.state_idx, self.img_idx, network_config=network_config)
```

- Perform `batch_metalearn` → inner loop
  - Pre-update : inputa, get outputa → final_eept_lossa, local_lossa
  - Compute fast gradients
  - Post-update: inputb, get outputb → final_eept_lossesb, local_lossesb
  - Here I omit `final_eept_lossa` and `final_eept_lossesb` : 我猜这个是用来构造双头结构的, 先省略略
  - **Edulidean distance is used for computing loss**
  - 这里虽然分成了pre-update和post-update, 我感觉和原版MAML差不多, 前者数据a, 用于计算训练误差, 后者数据b用来计算测试误差(内循环)

```
# pre-update
local_outputa, final_eept_preda = self.forward(inputa, state_inputa, weights, network_config=network_config)
# Compute train loss of inner loop
# act_loss_eps: default 1, the coefficient of the action loss
local_lossa = act_loss_eps * euclidean_loss_layer(local_outputa, actiona, multiplier=loss_multiplier, use_l1=FLAGS.use_l1_l2_loss)
```

```
# compute fast gradients, similar to maml
grads = tf.gradients(local_lossa, weights.values())
gradients = dict(zip(weights.keys(), grads))
```

```
# post-update
outputb, final_eept_predb = self.forward(inputb, state_inputb, fast_weights, meta_testing=True, network_config=network_config)
local_outputbs.append(outputb)
# compute test loss of inner loop
local_lossb = act_loss_eps * euclidean_loss_layer(outputb, actionb, multiplier=loss_multiplier, use_l1=FLAGS.use_l1_l2_loss)
local_lossesb.append(local_lossb)
```

- Output of `batch_metalearn`

```
local_fn_output = [local_outputa, local_outputbs, local_outputbs[-1], local_lossa, local_lossesb, final_eept_lossesb, flat_img_inputb, gradients_summ]
```

- Output of `construct_model`: map `batch_metalearn` to all training data

```
result = tf.map_fn(batch_metalearn, elems=(inputa, inputb, actiona, actionb), dtype=out_dtype)
```

6.3 `init_network()` in `mil.py`

- This is similar to meta update process in `maml.py`
- By calling `construct_model` we can get the result of inner loop

```
with Timer('building TF network'):
    result = self.construct_model(input_tensors=input_tensors, prefix=p
```

```
refix, dim_input=self._dO, dim_output=self._dU,
network_config=self.network_params)
outputas, outputbs, test_output, lossesa, lossesb, final_eept_lossesb,
flat_img_inputb, gradients = result
```

- Compute **average loss** for meta-update

```
total_loss1 = tf.reduce_sum(lossesa) / tf.to_float(self.meta_batch_size
)
total_losses2 = [tf.reduce_sum(lossesb[j]) / tf.to_float(self.meta_batc
h_size) for j in range(self.num_updates)]
total_final_eept_losses2 = [tf.reduce_sum(final_eept_lossesb[j]) / tf.t
o_float(self.meta_batch_size) for j in range(self.num_updates)]

# assign variables (this is for training, validation is similar)
self.total_loss1 = total_loss1
self.total_losses2 = total_losses2
self.total_final_eept_losses2 = total_final_eept_losses2
```

- Meta update:

```
# recall that in train(), input_tensors = [model.train_op]
self.train_op =
tf.train.AdamOptimizer(self.meta_lr).minimize(self.total_losses2[self.
num_updates - 1])
```