Progress report

Person Re-ID

2nd stage

Content

1. Tuning parameters of YOLOv5, SORT
2. Design Pattern
3. Person Re-ID

1. Tuning parameters of YOLOv5, SORT

Input size	IoU thresh	max_age (s)	min_hits (s)	HOTA	DetA	AssA	GT_Dets	GT_IDs	Dets	IDs
640	0.45	1	1	31.103	20.637	47.117	336891	1638	90195	957
1280	0.45	1	1	35.668	26.745	47.899			117807	1215
1280	0.50	1	1	35.688	26.699	48.038			117693	1218
1280	0.55	1	1	35.833	26.76	48.312			118065	1224
1280	0.60	1	1	35.479	26.512	47.808			117474	1224
1280	0.55	0.5	1	36.021	27.468	47.591			120159	1347
1280	0.55	1.5	1	36.064	27.228	48.113			119547	1296
1280	0.55	2	1	36.043	27.195	48.114			119379	1290
1280	0.55	1.5	0.15	41.625	36.583	47.88			169167	2343
1280	**0.55**	**1.5**	**0.25**	**40.728**	**34.763**	**48.18**			**157383**	**1974**
1280	0.55	1.5	0.5	38.789	31.384	48.329			138270	1563

1.1. YOLOv5

Note:

- On evaluation, either:
  - reduce `min_hits` should be as small as possible
  - or, as soon as a track is created (i.e object is successively detected `min_hits` times), trace back to previous frame and add that object to the return list.
- On deployment, specific problems might allow skipping few seconds of `min_hits`.

2. Design Pattern

2.1. Overview

3 categories:

- Creational patterns: object creation mechanism
    - Factory method
    - Builder
- Structural patterns: object assembly mechanism
- Behavioural patterns: object communication and assignment mechanism

2.2. Factory Method

```
# dialog_win.py

# . . .
button1 = WinButton()

# . . .
button2 = WinButton()

# . . .
button3 = WinButton()

# . . .
button4 = WinButton()
```

```
# dialog_mac.py

# . . .
button1 = MacButton()

# . . .
button2 = MacButton()

# . . .
button3 = MacButton()

# . . .
button4 = MacButton()
```

```
# dialog.py

# . . .
if os == 'Win':
    button1 = WinButton()
elif os == 'Mac':
    button1 = MacButton()

# . . .
if os == 'Win':
    button2 = WinButton()
elif os == 'Mac':
    button2 = MacButton()

# . . .
if os == 'Win':
    button3 = WinButton()
elif os == 'Mac':
    button3 = MacButton()
```

⇒ Code is ugly, hard to maintain, hard to extend

2.2. Factory Method

replace direct object construction calls with calls to a special factory method

```
# dialog

# . . .
if os == ‘Win’:
    creator = WinCreator()
elif os == ‘Mac’:
    creator = MacCreator()

# . . .
button1 = creator.create_button()

# . . .
button2 = creator.create_button()

# . . .
button3 = creator.create_button()
```

```
# base

interface Button:

class WinButton implement Button
    # . . .

class MacButton implement Button
    # . . .


interface Creator:
    abstractmethod create_button()

class WinCreator implement Creator
    create_button() -> WinButton

class MacCreator implement Creator
    create_button() -> MacButton
```

⇒ allow altering the type of object with minor change in client code

2.2. Factory Method

How to use:

1. All products must follow the same interface.
2. Implement an interface for creator.
3. The creator must be subclassed for each type of products.
4. Replace all direct object construction calls with calls to factory method.

⇒ factory design pattern does not focus on the details of how to construct/build a class efficiently, it rather focus on how to call it efficiently.

When to use:

- When there are many classes representing the same entity, and you'll have to alter/switch between them. These classes does the same works, but the implementation details are different.

2.3. Builder

We want the construction can vary in the signature.

```
class Pizza:
    # ...

def main():

    Pizza(size=10)
    Pizze(size=5, cheese=10)
    Pizza(cheese=7, hotdog=GermanHotDog(), sauce=True)
    # ...
```

2.3. Builder

Sol 1: a giant constructor with all possible parameters

```
class Pizza:

    def __init__(size, cheese, sauce, hotdog, peper,
    sugar, heat, tomatoes):
        # ...

def main():

    Pizza(10, None, 7, GermanHotDog(), None, None, None
    , None)
```

most of the parameters will be unused

code inside constructor pretty ugly

Sol 2: telescoping constructor

```
class Pizza{

    constructor Pizza() {...}
    constructor Pizza(int size) {...}
    constructor Pizza(int size, int cheese) {...}
    constructor Pizza(int size, int cheese, HotDog
    hotdog) {...}
    # ...
}
```

hard to write and read client code in case there are many parameters

2.3. Builder

```
class Pizza:

    class Builder:

        def __init__(self):
            self.reset()

        def reset(self):
            self._product = Pizza()

        def set_cheese(self, value):
            self._product.cheese = value
            return self

        def set_hotdog(self, value):
            self._product.hotdog = value
            return self

        def build(self):
            product = self._product
            self.reset()
            return product

```

```
pizza = Pizza.Builder().set_cheese(10).set_hotdog(15).
    build()
builder = Pizza.Builder()
pizza1 = builder.set_cheese(10).set_hotdog(15).build()
pizza2 = builder.set_cheese(0).set_hotdog(1).build()
```

Builder has none of the previous drawbacks

A single builder can be used to build multiple objects.

2.3. Builder

How to use:

1. declare construction steps in builder. Pay attention to builder `__init__`, `reset`, and `build` method.

⇒ builder design pattern does not focus on the details of how to call a class efficiently, it rather focus on how to construct it efficiently.

When to use:

- when designing classes whose constructors would have more than a handful of parameters (might be 4 parameters), especially if most of those parameters are optional.

2.3. Application

```python
class SimpleSCT(SCTBase):
    """YOLOv5 + SORT"""

    def create_detector(self) -> DetectorBase:
        detector = YOLOv5.Builder(HERE / './configs/yolov5s.yaml').get_product()
        return detector

    def create_tracker(self, loader: LoaderBase) -> TrackerBase:
        kalmanbox_builder = KalmanBox.Builder(HERE / './configs/kalmanboxstandard.yaml')
        tracker = SORT.Builder(HERE/'./configs/sort.yaml', loader, kalmanbox_builder).get_product()
        return tracker
```

```python
                                    for d in unmatched_dets:
                                        self.objects.append(self.kalmanbox_builder.set_box(dets[d]).get_product())
```

```python
# create detector and tracker
# TODO different options here: if opt.hardware == 'weak':
sct = SimpleSCT()
detector = sct.create_detector()
tracker = sct.create_tracker(loader)
```

3. Person Re-ID

3.1. Overview

Feature Learning
Metric Learning
Ranking Optimization

Training strategy:

- the trained network is refined frame by frame in different sequences through online training.

3.1. Triplet Loss in Siamese Network for Object Tracking

Focus on: Metric Learning

Replace the logistic loss with triplet loss.

$$\sum_{i}^{M} \frac{1}{2M} \log(1 + e^{-vp_i}) + \sum_{j}^{N} \frac{1}{2N} \log(1 + e^{vn_j})$$

Margin-based Triplet loss: $dp + \alpha < dn$, where margin α is manually selected.

Argue: Logistic loss maximizes the similarity score on a positive pair and minimizes it on a negative pair, but ignore the relative relationship between them.

Advantages:

- More (hard) training examples
- Larger gradient response

3.1. Triplet Loss in Siamese Network for Object Tracking

Proposed Triplet loss:

$$-\frac{1}{MN}\sum_{i}^{M}\sum_{j}^{N}\log\frac{e^{vp_i}}{e^{vp_i}+e^{vn_j}}. = \frac{1}{MN}\sum_{i}^{M}\sum_{j}^{N}\log(1+e^{vn_j-vp_i})$$

⇒ avoid manually selecting the suitable margin

Question:

- Is this formula ensure a margin?
- As for the frequently used Triplet loss $dp + \alpha < dn$, which leads to minimize $\Sigma(dp + \alpha - dn)$, isn't it identical if we omit the α term?

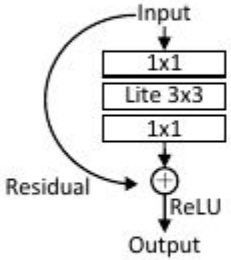
3.2. Omni-Scale Feature Learning for Person Re-Identification

Focus on Metric Learning

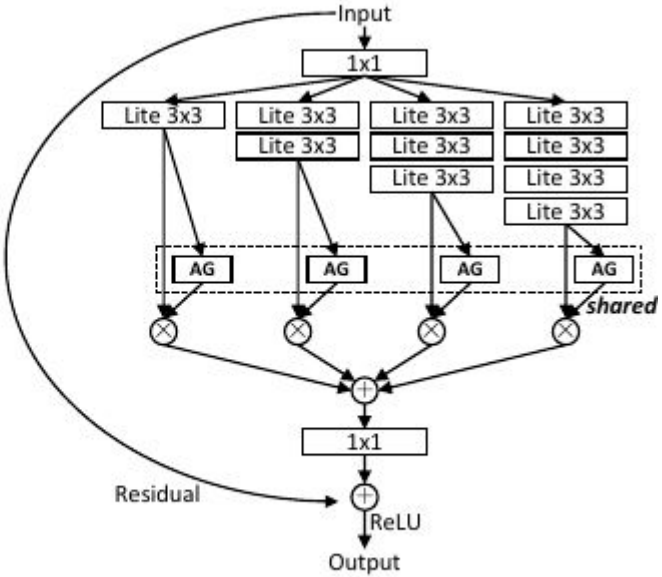
Argue: features corresponding small local regions (e.g. shoes, glasses) and global whole body regions are equally important.

Proposed architecture:

- multiple convolutional streams, each detecting features at a certain scale
- employ channel-wise weights, but is dynamically adjusted depending on input
- employ pointwise and depthwise convolutions



(a)



(b)

3.2. Omni-Scale Feature Learning for Person Re-Identification

Advantages:

- capture the local discriminative patterns
- lightweight model (FLOPS and #params)

Question:

- Does it really generalize well on new unseen person?
- How about tracking people in uniform?

Progress report

Person Re-ID

2nd stage