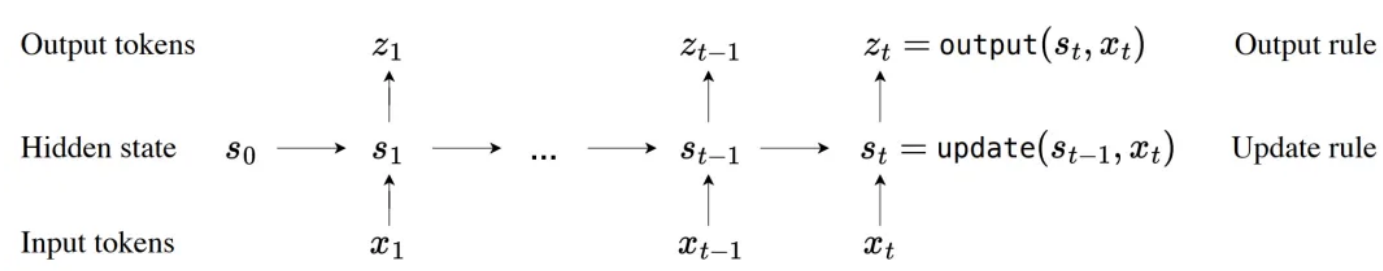
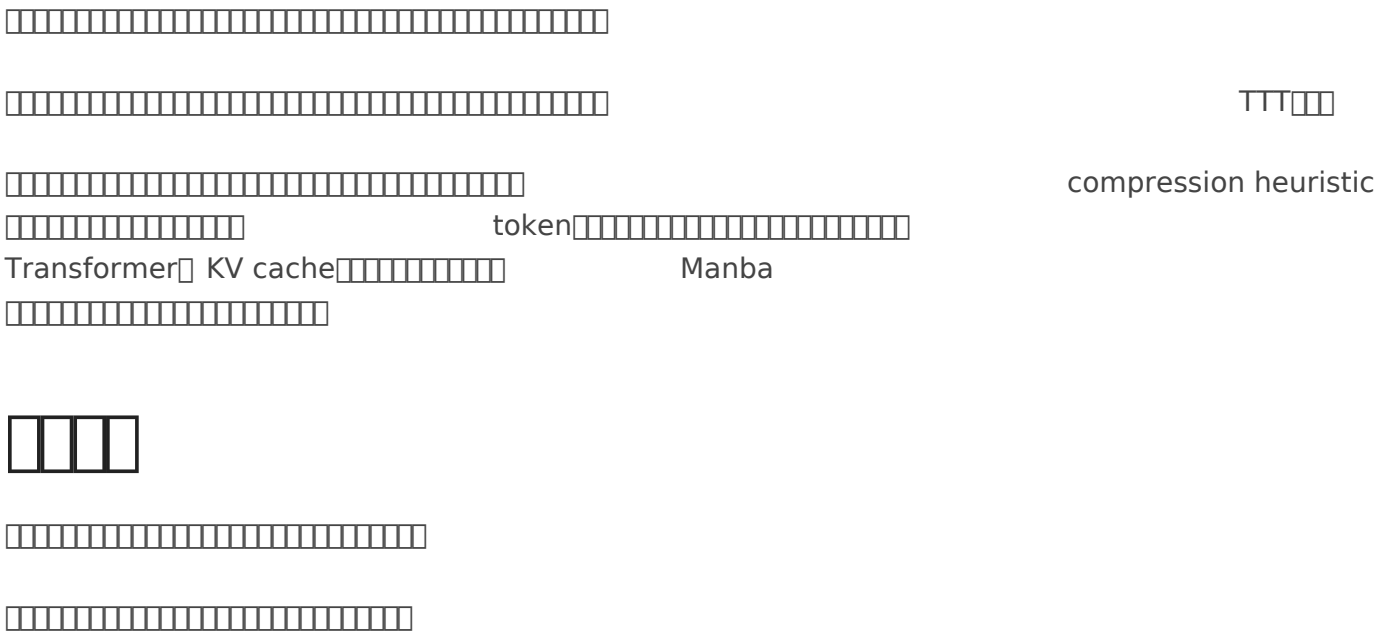


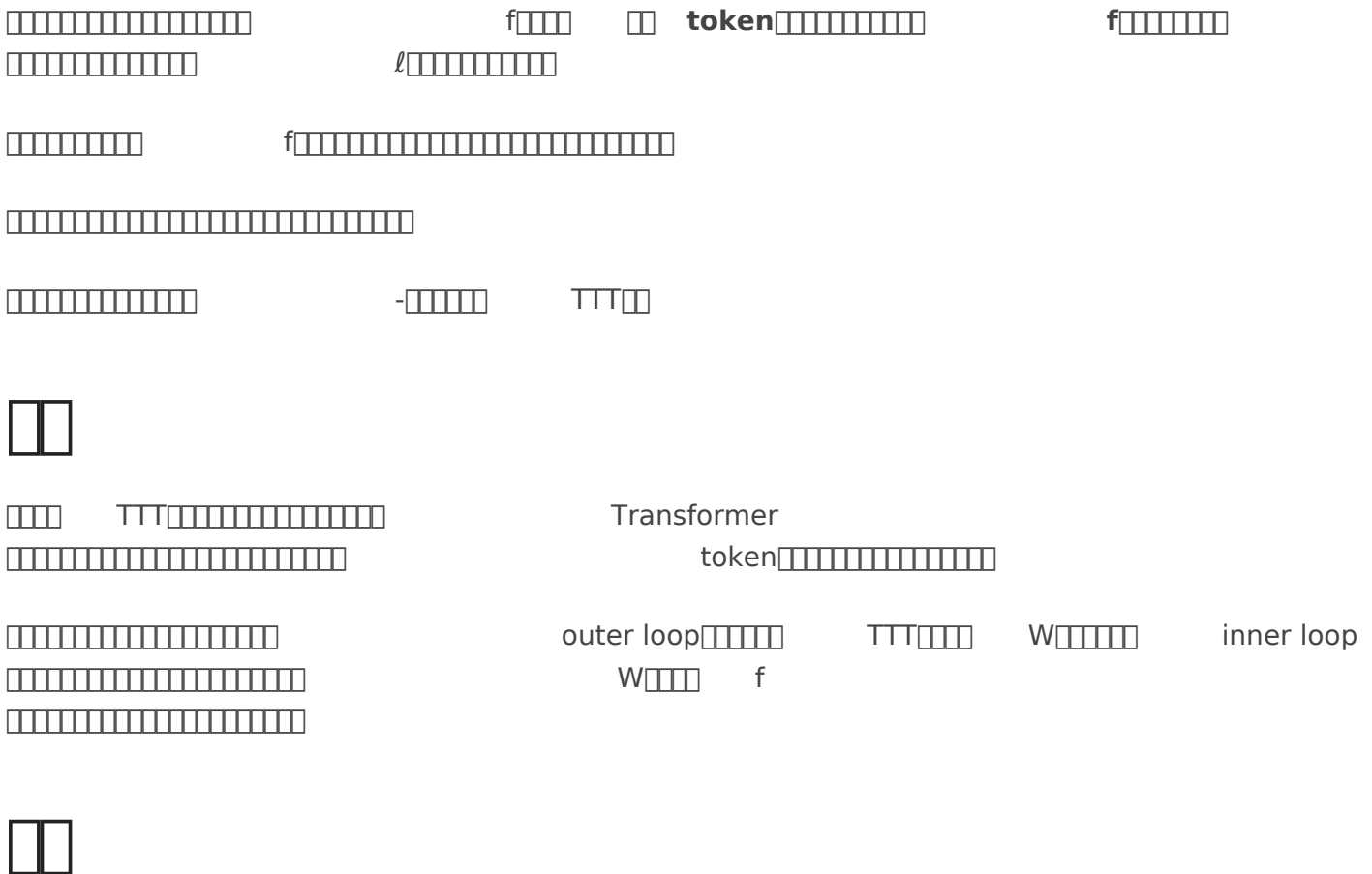
# TTT - Learning to (Learn at Test Time)



	Initial state	Update rule	Output rule	Cost
Naive RNN	$s_0 = \text{vector}()$	$s_t = \sigma(\theta_{ss}s_{t-1} + \theta_{sx}x_t)$	$z_t = \theta_{zs}s_t + \theta_{zx}x_t$	$O(1)$
Self-attention	$s_0 = \text{list}()$	$s_t = s_{t-1}.\text{append}(k_t, v_t)$	$z_t = V_t \text{softmax}(K_t^T q_t)$	$O(t)$
Naive TTT	$W_0 = f.\text{params}()$	$W_t = W_{t-1} - \eta \nabla \ell(W_{t-1}; x_t)$	$z_t = f(x_t; W_t)$	$O(1)$

Figure 4. **Top:** A generic sequence modeling layer expressed as a hidden state that transitions according to an update rule. All sequence modeling layers can be viewed as different instantiations of three components in this figure: the initial state, update rule and output rule. **Bottom:** Examples of sequence modeling layers and their instantiations of the three components. The naive TTT layer was shown in Figure 1. Self-attention has a hidden state growing with context, therefore growing cost per token. Both the naive RNN and TTT layer compress the growing context into a hidden state of fixed size, therefore their cost per token stays constant.





```
class TTT_Layer(nn.Module):
    def __init__(self):
        self.task = Task()

    def forward(self, in_seq):
        state = Learner(self.task)
        out_seq = []
        for tok in in_seq:
            state.train(tok)
            out_seq.append(state.predict(tok))
        return out_seq

class Task(nn.Module):
    def __init__(self):
        self.theta_K = nn.Param((d1, d2))
        self.theta_V = nn.Param((d1, d2))
        self.theta_Q = nn.Param((d1, d2))

    def loss(self, f, x):
        train_view = self.theta_K @ x
        label_view = self.theta_V @ x
        return MSE(f(train_view), label_view)
```

```
class Learner():
    def __init__(self, task):
        self.task = task
        # Linear here, but can be any model
        self.model = Linear()
        # online GD here for simplicity
        self.optim = OGD()

    def train(self, x):
        # grad function wrt first arg
        # of loss, which is self.model
        grad_fn = grad(self.task.loss)
        # calculate inner-loop grad
        grad_in = grad_fn(self.model, x)

        # starting from current params,
        # step in direction of grad_in,
        self.optim.step(self.model, grad_in)

    def predict(self, x):
        test_view = self.task.theta_Q @ x
        return self.model(test_view)
```

Figure 6. Naive implementation of a TTT layer with a linear model and online GD in the style of PyTorch. TTT\_Layer can be dropped into a larger network like other sequence modeling layers. Training the network will optimize the parameters of Task in TTT\_Layer, because both are subclasses of `nn.Module`. Since Learner is not a subclass of `nn.Module`, `state.model` is updated manually in the inner loop for each call of `state.train`. For simplicity, we sometimes overload `model` as `model.parameters`.

**theta\_K**

theta\_V 

--	--	--	--	--	--

sequence 

--	--	--	--

 theta\_K | theta\_V | theta\_Q |[illegible]

--	--

--	--	--	--	--

Pile ☐ ☐ ☐ ☐ $2k \square 8k \square \square \square \square \square \square \square \square \square \square$ Pile 

--	--	--	--	--	--	--	--

LLM 

--	--	--	--	--	--	--	--

TTT-MLP M

FLOP□□□□□□□□

[illegible]

TTT-Linear

[illegible]


FLOP 

□ 8k□□□□

TTT-Linear  $\square$  M  $\square$ TTT-MLP  $\square$  M  $\square \square \square \square \square \square \square \square$ Mamba 

## Transformer

TTT-MLP TTTTTT

Mamba 

TTT TTTT

## Mamba

--	--	--	--	--	--	--	--	--

Revision #1

Created 11 January 2025 09:44:05 by Colin

Updated 11 January 2025 09:44:09 by Colin