A Implementation details

Our algorithm is an extension of the Soft Actor-Critic algorithm [7, 50], implemented in PyTorch. Like [36, 51], we initialize agents' replay buffer with a 1000 seed observations collected with a uniform random policy. We update the Q-network pairs to predict the augmented Q value function at every interaction step, and the actor network to maximize the augmented Q value function every second interaction step. The augmented Q targets take the following form:

$$y = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \left[Q(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \alpha(\log \pi_\theta(\mathbf{a}_{t+1}|\mathbf{s}_{t+1}) - \log \phi_\theta(\mathbf{a}_{t+1}|\mathbf{a}_{t-\tau:t})) \right] \tag{6}$$

where $\mathbf{a}_{t+1} \sim \pi_\theta(\cdot|\mathbf{s}_{t+1})$. To train the networks we sampled B tuples of state, actions, next state, reward and terminal flags, as well as the τ actions that led to them. To allow the agent to train on observations early in episodes, we sampled τ from a uniform distribution of integers between 5 and τ_{max} (see tables 1, 2) for every mini-batch sample used for training. We update the adaptive priors used in SPAC and MIRACLE together with the actor network. All actor and critic networks consisted of two hidden layers with 256 ReLU units [52] each. The action prior used in MIRACLE was implemented as a multivariate isotropic Gaussian with learnable mean and standard deviation.

Since actions are bounded between -1 and 1, we transform actions sampled from the policy using the $\tanh$ transform $\mathbf{a}_t = \tanh(\mathbf{u}_t), \mathbf{u}_t \sim \pi_\theta$. We transformed the log-likelihood of an action under a Gaussian policy π_θ or action prior ϕ_θ using the following formula [7, 50]:

$$\log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \log \mu(\mathbf{u}_t|\mathbf{s}_t) - \sum_{i=1}^{D} \log(1 - \tanh^2(u_i)) \tag{7}$$

$$\log \phi_\theta(\mathbf{a}_t|\mathbf{a}_{t-\tau:t-1}) = \log \psi_\theta(\mathbf{u}_t|\mathbf{a}_{t-\tau:t-1}) - \sum_{i=1}^{D} \log(1 - \tanh^2(u_i)) \tag{8}$$

where $\log \psi_\theta(\mathbf{u}_t|\mathbf{a}_{t-\tau:t-1})$ is the log likelihood of the untransformed action $\mathbf{u}_t$ under the untransformed sequence prior ψ_θ.

A.1 Quantifying compressibility

We used the `LZ4` algorithm to quantify the compressibility of action sequences. The following code snippet describes how we computed the sequence complexity term in Eq. 3

```
sequence_i = action_sequence.numpy().ravel()
# get length of compressed sequence at t
length_t1 = len(compression_algorithm.compress(sequence_i))
action_next = policy(state).numpy().ravel()  # get next on-policy action
sequence_j = np.concatenate((sequence_i, action_next), axis=0)
# get length of compressed sequence at t+1
length_t2 = len(compression_algorithm.compress(sequence_j))
delta = length_t - length_t2  # delta is the difference
```

Since the actions were continuous vectors, we quantized all action sequences with the following function:

```
def quantize(action_sequence, N=100):
    return (action_sequence*N).floor()
```

Here N determines the granularity of the quantization, with lower N producing more coarse-grained sequences. We set $N = 100$ for our experiments.

A.2 Pseudo-code for `lz4` algorithm

The `lz4` algorithm compresses sequences by replacing repeating sub-sequences in the data with references to an earlier occurring copy of the sub-sequence. These copies are maintained in a sliding

window. Repeating sub-sequences are encoded as *length-distance* pairs (l, d), specifying that a set of l symbols have a match d symbols back in the uncompressed sequence. The following pseudo-code sketches compression implemented by `LZ4` [32]:

Algorithm 1 LZ4 pseudo-code

Require: Buffer size b, window size w, sequence $\mathbf{k}$
  $t = 0$
  window $\leftarrow \langle\ \rangle$
  while t $<$ len($\mathbf{k}$) **do**
    match $\leftarrow$ longest repeated occurrence in window found in $\mathbf{k}_{t:t+b}$
    if match exists **then**
      $d \leftarrow$ distance to start of match
      $l \leftarrow$ length of match
      $c \leftarrow$ symbol at $\mathbf{k}_{t+l}$
    else
      $d \leftarrow 0$
      $l \leftarrow 0$
      $c \leftarrow 0$
    end if
    output (d, l, c)
    start $\leftarrow \max(t - w + l, 0)$
    end $t \leftarrow t + l$
    window $\leftarrow \mathbf{k}_{\text{start:end}}$
    $t \leftarrow t + l + 1$
  end while

B Hyperparameters

B.1 Increasing reward scale

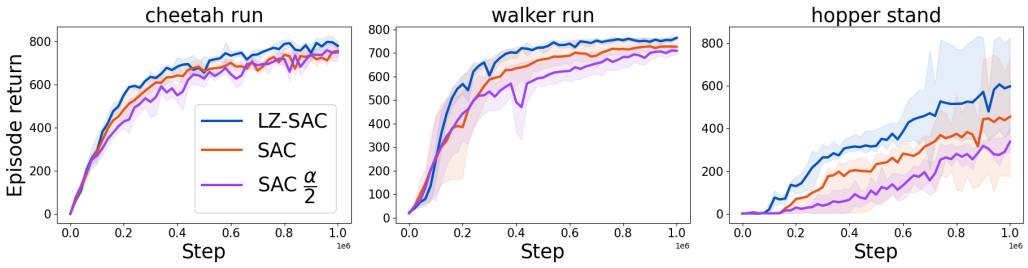

Figure 9: Halving the incentive of acting randomly does not close the performance gap between LZ-SAC and SAC.

We investigated whether LZ-SAC's performance improvement could simply be attributed to the incentive to act more deterministically. We tested whether we could attain the same level of performance with SAC just by lowering the incentive of acting randomly. Doubling the scale of extrinsic reward relative to the intrinsic reward of acting randomly did not close the gap between the algorithms (Fig. 9). Instead, we see a decline in performance when we set the incentive of randomness lower than $\alpha = 0.1$ (or $\alpha = 0.02$ in `walker run`). This indicates that there is value in having a preference for simplicity on the sequence level that goes beyond simply being predictable at the level of individual actions.

B.2 Algorithm hyperparameters

We implement all algorithms using hyperparameters from [36], with slight deviations depending on the task. Since the computational overhead of using `lz4` as a compressor is small compared to the

transformer, we train the agents using larger action sequences. All networks were trained with the Adam optimizer [53]. A full list of hyperparameters is given below:

Table 1: Hyperparameters used for SAC, MIRACLE, LZ-SAC, and SPAC

Hyperparameter	Value
Complexity cost α (`walker run`, `hopper hop`, `quadruped walk`)	0.02
Complexity cost (all other environments) α	0.1
Discount γ	0.99
Critic update frequency	1
Actor update frequency	2
Action prior update frequency	2
Soft update ρ	0.01
Batch size	128
Learning rate actor	10^{-3}
Learning rate critic	10^{-3}
Optimizer	Adam
Max context length LZ-SAC (τ_{max})	Interaction steps in episode $\times$ 0.4
Max context length LZ-SAC (τ_{max}; `walker run`)	Interaction steps in episode $\times$ 0.25

B.3 Transformer

The SPAC agent uses a causal transformer [9] to learn a prior over action sequences. Our transformer was implemented with the following hyperparameters:

Table 2: Transformer hyperparameters.

Hyperparameter	Value
Attention heads	5
Embedding dimensions	30
Learning rate decay	Linear
Warmup tokens	10000
Max context length (τ_{max})	20
Number of layers	2
Learning rate	3×10^{-4}
Dropout	0.1
Optimizer	Adam

C Task specification

We evaluated agents on tasks from the DeepMind Control Suite. Though dynamics are otherwise deterministic, the starting state of an episode is sampled from a distribution $p(\mathbf{s}_0)$. All episodes consist of 1000 environment steps. However, in practice the episode length is reduced to a number of *interaction steps*, that is smaller than 1000. This is due to an action repeat hyperparameter which determines how many times an action $\mathbf{a}_t$ is repeated after it is selected. An action repeat value of 4 thus reduces the number of time steps where the agent needs to act to 250 interaction steps. The action repeat hyperparameter makes it more practical to train agents in the DeepMind Control Suite [34]. We adopt conventional action repeat settings from the literature [36]. In the `walker` and `hopper` domains we fitted the action repeat value for all agents among $[2, 4, 8]$ and chose the value that produced the best performance. Table 3 shows the action repeat values used in our experiments:

C.1 Pixel-based control

Our LZ-SAC and SAC implementations in the visual control domain differed little from the state-based implementations. We equipped the agents with the convolutional neural network architecture and image augmentation transformation from [54]. All MLPs had two hidden

Table 3: Action repeat values.

Task	Action repeat
`acrobot swingup`	8
`cheetah run`	4
`fish swim`	4
`hopper hop`	8
`hopper stand`	8
`quadruped walk`	4
`reacher hard`	4
`walker run`	2
`walker run` (SPAC)	4

Table 4: All final average model scores in the DeepMind Control 100k benchmark with pixel observations. Scores are averaged over 10 runs after 100k steps for five seeds. Scores of the other baselines are the ones reported in the respective papers [41] [36].

DMC 100k	LZ-SAC	SAC	CURL	SAC+AE
Finger, Spin	**814**	738	767	740
Cartpole, Swingup	**683**	609	582	311
Walker, Walk	**635**	609	403	394
Ball In Cup, Catch	653	499	**769**	391
Cheetah, Run	307	**396**	299	274
Reacher, Easy	513	427	**538**	274

layers and 512 ReLU units. We optimized the α hyperparameter both for the LZ-SAC and SAC agents for all tasks with a grid search. For tasks with higher dimensional action spaces a lower α of 0.01 worked best. In the end, the best performing α for both algorithms was the same across task. Since `LZ4` encoding lengths are not necessarily on the same scale as the log likelihoods of the actions given the state in Equation 4, we experimented with scaling the `LZ4` encoding cost by 0.5, which slightly improved performance. The full scores of the models are in Table 4.

Table 5: Action repeat values and complexity cost for both LZ-SAC and SAC agents in the pixel-based tasks.

Task	Action repeat	α
`finger spin`	2	0.01
`cartpole swingup`	8	0.1
`walker walk`	2	0.01
`ball in cup catch`	4	0.01
`cheetah run`	4	0.01
`reacher easy`	2	0.1

D Mutual information approximation

The mutual information $I(X;Y)$ between variables X and Y is a measure of how much they depend on each other. In our case we are interested in the mutual information between states and actions $I(\mathbf{s};\mathbf{a})$. The mutual information here quantifies how many bits of information knowing the outcome of the random variable $\mathbf{s}_t$ provides about the other random variable $\mathbf{a}_t$, in other words, how much the state reveals about what action will be selected. The more an agent's actions vary as a function of the state, the more bits of information the state reveals about the action that the agent will select.

The mutual information is defined as the following quantity

$$I(\mathbf{a};\mathbf{s}) = \mathcal{H}[\mathbf{a}] - \mathcal{H}[\mathbf{a}|\mathbf{s}] \tag{9}$$

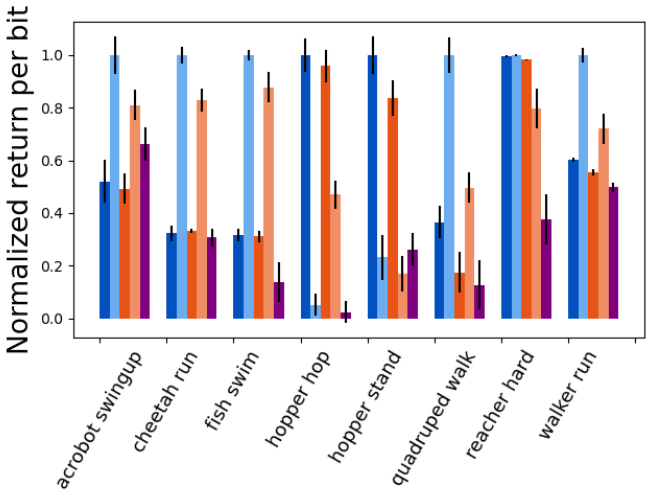

Figure 10: Return per bit for the stochastic policies. LZ-SAC attains the highest return per bit ratio in most tasks.

To compute the mutual information we need to know the entropy of the distribution of actions used to solve the task, and the conditional entropy of $\mathbf{a}|\mathbf{s}$. Since we make the policies deterministic, we know that $\mathcal{H}[\mathbf{a}|\mathbf{s}] = 0$. This way the mutual information reduces to the entropy over actions $\mathcal{H}[\mathbf{a}]$. Given a sample of actions produced by the agent solving the task, we approximate $\mathcal{H}[\mathbf{a}]$ the following way: We first quantized each selected action into $100 \times |\mathcal{A}|$ bins, where $\mathcal{A}$ is the action space. We then calculated a categorical distribution over actions based on the frequencies of the quantized actions, the entropy of which we used as our approximation for $\mathcal{H}[\mathbf{a}]$. The categorical distribution was calculated based on actions selected over 50 episodes.

D.1 Return per bit for stochastic policies

We evaluated the information efficiency of the stochastic variants of the policies learned by LZ-SAC, SPAC, SAC and MIRACLE. We approximated the entropy of the distribution of actions in the same way described above, sampling actions over 50 episodes. To compute the conditional entropy of actions given the state $\mathcal{H}[\mathbf{a}|\mathbf{s}]$, we sampled 1000 actions from the policy at every state $\mathbf{s}_t$. We then calculated a categorical distribution (described in previous section) based on this sample, the entropy of which we used as our approximation of the conditional entropy $\mathcal{H}[\mathbf{a}|\mathbf{s}]$. In this setting too, the SPAC algorithm tends to produce the most information efficient agents (Fig. 10).

E Open-loop control

Increasing the number of closed-loop actions used to prompt the transformer makes it generate more rewarding action sequences. This shows the importance of providing the sequence models with enough context, to be able to predict rewarding behaviors (Fig. 11).

F Partial observability

The augmented reward function that induces the preference for simple action sequences depends on the actions the agent selected in the past. This makes the reward function partially observable for a purely state-conditioned policy. Our agents learn to maximize this reward function despite this partial observability. We tested whether augmenting the state to contain information about actions selected in the past produced substantial differences in the learned policies. We equipped the LZ-SAC agents with a recurrent neural network (a Gated Recurrent Unit [55]) whose inputs were sequences of actions. We trained this network along with a single readout layer to produce embeddings $\mathbf{e}_t$ of action sequences with which we defined the augmented state $\mathbf{a}_t \sim \pi_\theta(\cdot|\tilde{\mathbf{s}}_t)$ where

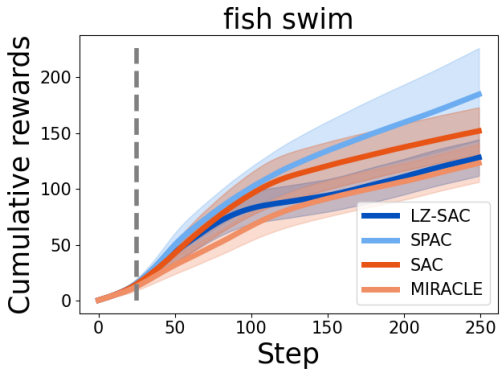

Figure 11: Cumulative rewards attained by sequence priors in the `fish swim` environment, with a prompt length of 25.

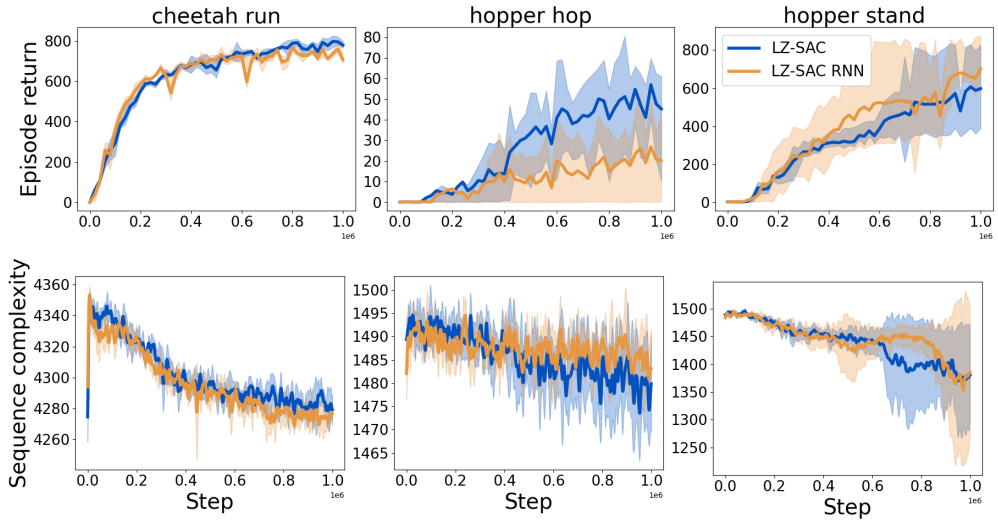

Figure 12: Returns and compressibility of action sequences when the reward function is partially observable and fully observable.

$\tilde{\mathbf{s}}_t = Concatenate(\mathbf{s}_t, \mathbf{e}_t)$. In three tasks we observed only minor differences in the policies learned (Fig 12).

G Pretraining the prior

We trained Transformer models to perform next-action prediction from action sequences produced by the converged LZ-SAC agent for all tasks. Using the pretrained transformers (with frozen weights) rather than a randomly initialized one whose weights were updated with stochastic gradient descent sped up learning significantly and allowed the SPAC agent to learn more rewarding behaviors (see Fig. 13). This showcases an interesting possible connection between our sequence compression framework and behavioral cloning.

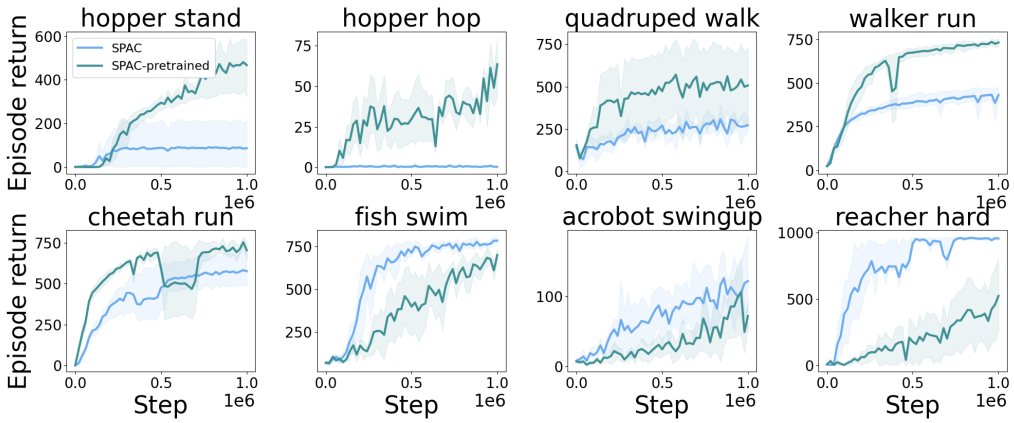

Figure 13: Learning curves for SPAC using a pretrained transformer with frozen weights and a randomly initialized transformer. Using a pretrained transformer allows SPAC to solve the more challenging tasks in the benchmark. Learning curves are averaged over three seeds. Shaded region represents 20-80 percentile.