Appendix

Diverse, multi-modal behaviors generated by our models on different environment are best experienced and understood in a video. We invite you to visit https://submission0.github.io to see BeT models in action.

A Environment and Dataset Details

Point mass environments: In the point mass environment, we have a simple point-mass agent with two-dimensional observation and action spaces. The observation of the agent denotes the (x, y) position of the agent, while the action sets the immediate $(\Delta x, \Delta y)$ displacement of the agent in the next timestep.

To show the effects of unimodal and multimodal behavioral cloning algorithms more cleanly, we also add a "snapping" effect to the environment which moves the agent close to the nearest integer coordinates after each step.

We generate random trajectories for each of our Multipath experiment datasets.

1. In the first one (Fig. 2), our dataset has two modes, which are colored differently in the figure based on the path taken at the fork.
   (a) In the first set of demonstrations, the point mass follows the trajectory $(1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4), (4, 3), (4, 2), (5, 2)$.
   (b) In the second set of demonstrations, the point mass follows $(1, 2), (2, 2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (5, 2)$.
2. For the second Multipath environment (Fig. 5), there are three modes of demonstration, which are colored in the figure according to their first step direction.
   (a) In the first set of demonstration, the point mass follows $x = y$ from $(0, 0)$ to $(8, 8)$ with $\sqrt{2}$ size step increments.
   (b) In the second set of demonstration, the point mass follows straight lines from $(0, 0) \rightarrow (0, 4) \rightarrow (4, 4) \rightarrow (8, 4) \rightarrow (8, 8)$ with step size 1.
   (c) In the third set of demonstration, the point mass follows straight lines from $(0, 0) \rightarrow (4, 0) \rightarrow (4, 4) \rightarrow (4, 8) \rightarrow (8, 8)$ with step size 1.

CARLA environment: We use the CARLA [19] self-driving environment to examine BeT performance in environments with high-dimensional observation spaces. CARLA uses the Unreal Engine to provide a photo-realistic driving simulation. We create our environment on the Town04 map in CARLA 0.9.13. The observation space is $224 \times 224 \times 3$ RGB images from the vehicle, which are processed by an ImageNet-pretrained, frozen ResNet-18 to a 512-dimensional real-valued vector. The action space is $[-1, 1]^2$ with an accelerator-brake axis and a steering axis.

The dataset on this environment is collected with the built-in PID agent with minor tuning. We fix waypoints in the trajectory that the demonstration agent needs to follow. The waypoints fork around two central blocks: one set of trajectories thus go to the left, while another set of demonstration trajectories go to the right. While collecting the demonstrations, we add some noise in the environment before executing an action so that there is some variation in the set of 100 total demonstrations that we collect in the environment.

We do not introduce any traffic participants in this environment intentionally as we intend to show the effects of cleanly bi-modal distributions on the learning algorithms in an environment more complicated than the point-mass environments.

Block-push environment: We use a simulated environment similar to Multimodal Push environment described in [23]. We take the environment implementation directly from the PyBullet [14] based implementation provided by Florence et al. [23] in https://github.com/google-research/ibc/tree/master/environments.

In our environment, an XArm robot is situated in front of two blocks in a 0.75×1 plane. On the plane there are also two square targets. The goal of the agent is to push the blocks inside of the

squares. However, the exact order of the block being pushed, or the combination of which block is pushed in which square doesn't matter. A block is considered successfully pushed if the center of the block is less than 0.05 away from a square.

On initialization, the blocks' positions are randomly shifted within a rectangle of side lengths $(0.2, 0.3)$, while the squares are randomly shifted within a rectangle of size $(0.01, 0.015)$. Additionally, the blocks were rotated at an uniformly arbitrary angle, while the target squares were rotated at an angle between $(\frac{\pi}{6}, -\frac{\pi}{6})$.

The demonstrations in this environments were collected with a hard-coded controller. There are two modes of multimodality inherent in the controller generated demonstartions. The controller:

1. Selects a block to start pushing first,
2. At the same time, independently chooses a target for the block to be pushed into.
3. Once the first block is pushed to a target, it pushes the second block to the remaining target.

Thus combinatorially, the controller is capable of four different modes of behavior. There are additional stochasticity in the controller behavior since there are many ways of pushing the same block into the same target.

The controller pushes the blocks to their targets following specific behavior primitives, such as moving to origin position, moving to a place collinear with a block and its target, and making a straight motion from that position towards the target unless the block rotates too much from its starting position.

Our models were trained on 1,000 demonstrations, all generated from the controller under the above randomized modes.

Franka kitchen environment: For the final set of experiments, we use the Franka Kitchen environment originally introduced in the Relay Policy Learning [30] paper. In that paper, the authors introduce a virtual kitchen environment where human participants in VR manipulated seven different objects in the kitchen: one kettle, one microwave, one sliding door, one hinged door, one light switch, and two burners. In total, we use 566 demonstrations collected by the researchers in that paper, where in each demonstration episode, each participant performed four manipulation task specified by the researchers in advanced.

The manipulator agent in simulator is a Franka Emika Panda robot, which is controlled through a 9-dimensional action space controlling the robot's joint and end-effector position. The 60-dimensional observation space is split into two parts, the first 30 dimension contains information about the current position of the interesting factors in the environment, while the last 30 dimensions contain information about the goal of the demonstrator or the agent. Note that in our demonstrations and our environments, we zero out the last 30 dimensions in all cases since we assume goal is not labelled in the demonstrations and is not specified in the unconditioned rollouts of the model.

One thing to note that, while the D4RL [24] paper also has three versions of the dataset, we chose to use the original version of the collected data from the Relay Policy Learning [30] paper. That is because the relay policy learning dataset is not labeled with intended tasks of the participants or rewards, while the D4RL dataset is geared towards that.

B Implementation Details and Hyperparameters

B.1 Baselines

Multi-layer Perceptron with MSE For our MLP with MSE baselines, we trained fully connected neural networks with optionally BatchNorm layers. In each of our environment, we varied the depth and the width of the MLPs to fit them best according to the bias-variance trade-off, while training them on 95% of the dataset and testing on the remaining 5% on the dataset in terms of MSE loss.

Nearest Neighbor Nearest Neighbor is conceptually the simplest baseline we show in this paper. During training, our Nearest Neighbor model simply stores all the (o, a) pairs. During test time,

given a query observation, o, we find the observation o' with the minimum Euclidean distance to that in the representation space, and execute the associated action a' in the environment.

While it is a simple baseline, we show that it has a surprisingly high effectiveness in simple environments like CARLA, or dense environments like Kitchen where there is less of a chance in going OOD simply by executing seen actions. On the other hand, in environments like Block-push where the model needs to interpolate or extrapolate more, the NN model fails more.

k-Nearest Neighbor with Locally Weighted Regression A slightly more robust version of NN for regression problems, k-NN with locally weighted regression or LWR, is the next baseline we use. In this baseline, we take the k-nearest neighbors (in all our cases, 5) in the observation representation space, and take a weighted average of their associated actions. The weighting is based on the negative exponent of the distance, or namely, $\exp -||o - o'||$, as seen in [56]. This model is better than simple Nearest Neighbors in interpolations, and thus we see a higher success in the Kitchen environment.

Continuous Generative Model: VAE with Gaussian Prior Following prior works[60], we use variational auto-encoders (VAE) for encoding and decoding sequences of actions into a smaller latent space. The VAE here learns to compress a sequence of $T = 10$ actions into a single latent variable z of 10 dimensions. The hyperparameters for training the VAE has been taken directly from Pertsch et al. [60].

Concurrently with training the VAE, we train a state-conditioned latent prior model that tries to predict $P(z \mid o)$. This latent generator produces a vector of μ and σ which is sampled to find latent z, and we feed a Gaussian distributed variable z back into the decoder network where the action sequence is reconstructed. For the current observation o_t, sequence of reconstructed actions $a_t, \cdots, a_{t+9}$ are performed in a simulated environment.

The design choices of this algorithm has been heavily inspired by [60]. Although this model shows promise in theory, we found in practice that unconditional rollout from this model is not very successful. We believe the shortcoming is a result of random sampling from the z space that does not take into account the recently executed actions, and using a single-mode Gaussian as the state prior similar to [60], and thus this baseline is only slightly better than the MLP-MSE model.

Continuous Generative Models: Normalizing Flow with and without Prior Similar to Singh et al. [70], we use a Normalizing Flow [18] based generative model. We follow the architectural choices and the hyperparameters from [70] in our baseline implementation.

Our observation-conditioned Flow model is trained on the distribution $P(a \mid o)$ to continuously transform it into an identity Gaussian distribution of the same dimensions as a. To find a better prior than simply an identity Gaussian, we also trained a prior model that generates μ, σ of a Gaussian distribution given the observation o. We found that the prior improves the quality of the rollouts, however slightly.

We believe the under-performance of these continuous generative approaches were based on two major problems. One is that they fail to take historical context in concern, and by being a continuous distribution, returned less likely actions that led to more rollouts going OOD. Second, they were designed with a focus of making RL approachable by compressing the action space, which requires having a prior that is not so strict. However, most of BeT's performance comes from having a strong prior over the actions, which is only augmented by the action offset prediction.

Implicit Behavioral Cloning Implicit Behavioral Cloning (IBC) [23] takes a different approach in behavioral cloning, where instead of learning a model $f(o) := a$, we learn an energy based model $E(o, a)$ where the intended action a at any observation is defined as $\arg\min_a E(o, a)$. While this suffers from all the classic issues of training an EBM, like higher sample complexity and higher complexity in sampling, IBC models have been shown to have higher success in learning multi-modal and discontinuous actions.

As a baseline, we use the official implementation provided in https://github.com/google-research/ibc For the CARLA environment, we use equivalent hyperparameters from the "pushing from pixels" hyperparameters. For the Block-pushing environment, we use the "pushing from states" hyperparameters. Finally, for the Kitchen environment, we use the "D4RL kitchen" hyperparameters.

While IBC is our strongest baseline, in our experience it is also one that is quite easy to overfit to our datasets. As a result, we monitored test performance over the training and had to employ early stopping for both the CARLA and the Block-pushing tasks.

Trajectory Transformers Trajectory Transformers [35], especially the variant that is trained without any rewards only on states and actions from demonstrations, seem similar to our approach, there are a few crucial differences. While we agree that BeT and Trajectory Transformer based behavior cloning both use some type of discretization to fit demonstration datasets with a minGPT, we believe that is where the similarities end. The primary differences between the algorithms is in our design choices: namely what distributions they model, and consequently how they treat the observations. The differences are explained more thoroughly below.

- **Modeled distribution**: From a provided set of demonstrations, trajectory transformers model the joint distribution P(action, observations). On the other hand, BeT models the conditional distribution P(action | observations). Modeling the joint distribution requires MinGPT to model the forward dynamics of the environment, which can be arbitrarily difficult based on the environment.
- **Observation discretization**: Because trajectory transformers have to model the observations as well, it needs to discretize the observation space. As a result, TT cannot extend to high dimensional observational spaces, such as visual observations. This limitation is also acknowledged by the authors of Trajectory Transformers. BeT, on the other hand, does not model the observations and thus does not need to discretize them. Thus BeT can scale to arbitrarily high dimensional observations, as we show in the CARLA environment experiments, where BeT learns behaviors from high dimensional visual observations.
- **Efficient historical encoding**: Trajectory transformer encodes each (state, action) pair into a total of $|S| + |A|$ input/output tokens, while BeT encodes them into one input/output token. On a base MinGPT implementation that means a $O((|S| + |A|)^2)$ efficiency gain for BeT, or for example 4761x less compute for the same historic context in the Kitchen environment.

As a baseline, we trained and rolled out Trajectory Transformer on the Kitchen environment. It failed to complete any tasks for unconditioned, greedy, or beam search rollouts. We would like to note that the Kitchen environment is more complicated than the MuJoCo environments (HalfCheetah, Hopper, Walker2d, and Ant) that the paper experimented on. At the same time, this environment has an order of magnitude fewer samples on the training set (10^6 vs. approximately 120k). We tried both our own implementation and the implementation from https://github.com/Howuhh/faster-trajectory-transformer with the recommended parameters for the AntMaze environment, which is the largest environment used by the authors.

B.2 Algorithm Details

Loss function details: In this paper, we use two loss functions that are inspired by practices in computer vision, in particular object detection. The first of them is the Focal loss [44], and the second one is the Multi-task loss [26].

The Focal loss is a simple modification over the cross entropy loss. While the normal cross entropy loss for binary classification can be thought of $\mathcal{L}_{ce}(p_t) = -\log(p_t)$, the Focal loss adds a term $(1 - p_t)^\gamma$ to this, to make the new loss

$$\mathcal{L}_{focal}(p_t) = -(1 - p_t)^\gamma \log(p_t)$$

This loss has the interesting property that its gradient is more steep for smaller values of p_t, while flatter for larger values of p_t. Thus, it penalizes and changes the model more for making errors in the low-probability classes, while is more lenient about making errors in the high probability classes. Using this error in the object detection world has helped with class imbalance between different classes, and here it helps BeT learn to predict different k-means from the dataset even if their appearance in the dataset is not completely balanced.

For the multi-task loss, we use the formulation

$$\text{MT-Loss}\left(\mathbf{a}, \left(\langle \hat{a}_i^{(j)} \rangle\right)_{j=1}^{k}\right) = \sum_{j=1}^{k} \mathbb{I}[\lfloor \mathbf{a} \rfloor = j] \cdot \|\langle \mathbf{a} \rangle - \langle \hat{a}^{(j)} \rangle\|_2^2$$

This helps us penalize only the offset for the ground truth class, thus making sure the MinGPT is not trying to predict the right action offset through all classes and instead only trying to predict the action offset through the right class.

In practice, we optimize the combined loss, $\mathcal{L}_{focal} + \alpha\mathcal{L}_{mt}$ while α is a hyperparameter that just makes sure at initialization the two losses are of the same order of magnitude.

Compute details: All of our code was run in a single NVIDIA RTX 3080 GPU for state-based environments and RTX 8000 for image-based environments.

Performance measurement details: We measured the performance reported in the Section 3.5 in an NVIDIA RTX 3080 machine with AMD Threadripper 5950x CPUs. We took the average over three runs to minimize inter-run variances, and measured wall-clock time to report in the paper.

In terms of raw computation time to determine one action from the observations, in the Kitchen environment, BeT took 2.8 ms, while IBC took 52 ms and MLP, as the fastest point of comparison, took 0.5 ms. On the same environment, a single step of Trajectory Transformer took 867.86 ms, on an implementation that used more advanced tricks such as attention caching.

Hyperparameters list: We present the BeT hyperparameters in Table 4 below:

Table 4: Environment-dependent hyperparameters in BeT.

Hyperparameter	Point-mass	CARLA	Block-push	Kitchen
Layers	1	3	4	6
Attention heads	2	4	4	6
Embedding width	20	256	72	120
Dropout probability	0.1	0.6	0.1	0.1
Context size	2	10	5	10
Training epochs	10	40	350	50
Batch size	64	128	64	64
Number of bins k	2; 3	32	24	64

However, we have found that as long as the model does not overfit, a wide range of parameters all yield favorable results for BeT; thus, this table should be taken as reference values for reproducing our results rather than the only parameter sets that work.

Apart from that, we have some hyperparameters that are shared across all BeT experiments. They are reproduced in Table 5.

Table 5: Shared hyperparameters for BeT training

Name	Value
Optimizer	Adam
Learning rate	1e-4
Weight decay	0.1
Betas	(0.9, 0.95)
Gradient clip norm	1.0

B.3 Pseudocode

See the pseudocode described on Algorithm 1.

B.4 Architecture and Implementation

For our implementation, we used the MinGPT [36] repository almost as-is. We modified the input token conversion layer to a linear projection layer to handle our continuous, instead of discrete, inputs. Apart from that, we followed the MinGPT architecture quite exclusively, with successive attention layers with a number of attention head and embedding dimensions. Between the layers, we used dropout regularization same as [36].

Algorithm 1 Learning Behavior Transformer from a dataset of behavior sequences.

Input: Dataset $(o_{t,i}, a_{t,i})_{t,i}$ for $0 \leq i \leq$ number of demonstrations, $0 \leq t \leq$ maximum episode lengths, intended number of clusters k and context history length h.

Initialize: θ_M the parameters for MinGPT, $\{A_i\}_{i=1}^k$ cluster centers randomly in the action space.

Learn k-means encoder/decoder:
Using all possible $a_{t,i}$, learn the k cluster centers using the k means algorithm.
Set $\{A_i\}_{i=1}^k$ as the learned cluster centers.

Define functions:
$\lfloor a \rfloor := \arg\min_{i=1}^k ||a - A_i||$
$\langle a \rangle = a - \lfloor a \rfloor$
$\text{Enc}(a) = (\lfloor a \rfloor, \langle a \rangle)$
$\text{Dec}(\lfloor a \rfloor, \langle a \rangle) = A_{\lfloor a \rfloor} + \langle a \rangle$

Train MinGPT trunk of BeT:
while *Not converged* **do**
Sample trajectory subsequence $(o_t, a_t), \cdots, (o_{t+h-1}, a_{t+h-1})$ from the dataset.
Feed in the observations $(o_t, o_{t+1}, \cdots, o_{t+h-1})$ into the MinGPT.
Get categorical distribution probabilities $p_{\tau,c}$ for $t \leq \tau \leq t+h-1, 1 \leq c \leq k$.
Compute focal loss $\mathcal{L}_{ce}$ of $p_{\tau,c}$ against ground truth class $\lfloor a_\tau \rfloor$, for all τ, c .
Get the residual action offset per class, $\langle a_{\tau,c} \rangle$, for all τ, c from MinGPT.
Calculate the multi-task loss, $\mathcal{L}_{mt}$, against true class predicted offset, $\sum_\tau ||\langle a_{\tau,\lfloor a_\tau \rfloor} \rangle - \langle a_\tau \rangle||_2^2$
Backprop using the normalized loss, $\mathcal{L}_{ce} + \alpha \mathcal{L}_{mt}$ where α makes the losses of equal magnitude.

Running on the environment:
while *Episode not completed* **do**
Stack the last h observations in the environment, $(o_t, o_{t+1}, \cdots, o_{t+h-1})$ and feed into MinGPT.
Get categorical probabilities $p_{\tau,c}$ for $t \leq \tau \leq t+h-1, 1 \leq c \leq k$ from the MinGPT.
Sample a class c from $p_{t+h-1,c}$ for $1 \leq c \leq k$.
Get the associated action offset, $\langle a_{t+h-1,c} \rangle$ from the MinGPT.
Decode into full continuous action, $\overline{a}_{t+h-1} := \text{Dec}(c, \langle a_{t+h-1,c} \rangle)$
Execute decoded action $\overline{a}_{t+h-1}$ into environment.

For the smallest tasks, like point-mass environments, we used models with approximately 10^4 parameters, which went up to around 10^6 for Kitchen environments.

C Ablation studies

In this section, we provide more details about the ablation studies presented in the main paper, as well as present detailed plots of our ablation studies that compare different versions of the BeT architecture.

C.1 Ablating historical context

One of the reasons why we used transformer-based generative networks in our work is because of our hypothesis that having historical context helps our model learn better behavioral cloning. Our experiments are performed by using the same model and simply providing sequences of length one on training and test time. As we can see on Sec. 3.5, having some historical context helps our model learn much better.

C.2 Ablating the number of discrete bin centers, k

Since BeT is trained with a sum of focal loss for the binning head and MSE loss for the offset head, the number of cluster centers present a trade-off in the architecture. Concretely, as the number of bins

go up, the log-likelihood loss goes up but the MSE loss goes down. In Sec. 3.5, we showed that using only one bin ($k = 1$) decreases the performance level of BeT.

In this section, we present the plot of the variation in performance as k value changes.

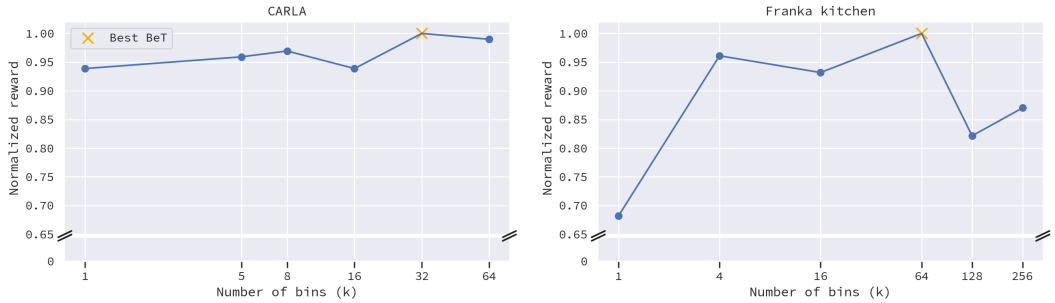

Figure 6: Ablating the number of discrete bin centers k for BeT. Reward is normalized with respect to the best performing model.

C.3 Ablating the core model in the architecture

To ablate the core MinGPT transformer model in the architecture, we replace it with a fully-connected MLP network, an LSTM network, and a temporal convolution architecture.

Since generally MLP networks are not capable of taking in historical context in consideration, we instead stack the last t frames of observation to pass into the MLP network. Near the beginning of a trajectory, the stack of observation is zero-padded to t frames. For the intermediate layers in the MLP, we keep the same width and the number of layers as the corresponding MinGPT.

For the LSTM network and the temporal convolution, we simply replace the MinGPT trunk with an LSTM trunk and try to train the same sequence-to-sequence model with the same historical context size.