Realtime Doodle Inpainting

Nicholas Batchelder*
University of Washington
Seattle, Washington
nicbat@cs.washington.edu

Eashver Elango*
University of Washington
Seattle, Washington
eashelon@cs.washington.edu

1. Introduction

In recent months, State Space Models (SSM) [6] have grown incredibly popular as a replacement for transformer-style models for generative tasks. In practice, SSMs like Mamba's [5] ability to retain its modeling power while scaling with sequence length alongside hardware-specific algorithms that speed up inference by 3x makes them in theory, a far better architecture for real-time applications.

In this paper, we aim to develop a novel pipeline incorporating the Mamba architecture for the application of real-time content in-painting. We also explore whether Mamba is suitable for use in real-time applications.

The main problem we aim to explore is whether SSM-based architecture can be used in real-time applications. To our knowledge, we know no pipelines that attempt to use SSMs like Mamba in real-time applications.

Using Google's Quick Draw dataset [1], we intend to use Mamba as an auto-completion engine for quick drawings (doodles) in real-time.

To allow for real-time performance, our idea is to apply learnings from real-time transformer pipelines. Papers in this area often try to simplify the multi-head attention block to scale linearly with input rather than quadratically and also improve its performance on GPU hardware [19]. However, SSMs like Mamba already include those optimizations [5].

Our application will have two parts:

1. A web-based canvas for a user to draw a doodle. This canvas will send data about the user's drawing to the completion model described below.

2. An SSM "completing" the user's doodle and suggesting the best line to improve the doodle.

The SSM completion model can be evaluated and trained on Google's Quick Draw dataset [1].

*Equal Contribution. Ordered alphabetically by last name. Nicholas proposed using the QuickDraw! dataset, proposed focusing on sketch autocomplete, and developed the demo. Eashver proposed using the Mamba architecture, trained/finetuned the models, and developed the experiments. Both authors mutually proposed and developed the branched architecture.

The biggest challenge of a real-time in-painting completion model is decreasing its latency to a level where the user interprets the auto-completions as "real-time." However, decreasing model latency did not prove to be a challenge, and "close to real-time" auto-completion was achieved, as we will discuss further in the paper.

Our experiment lead to a quick drawing equivalent of sentence auto-complete, where a user can start a sketch and the web app shows a "completed" drawing on a separate canvas. The quality of this outcome is dependent on many factors, like the accuracy of the model (how accurately it predicts pen movements), the number of classes we train on, and the quality of our data itself.

2. Related Work

There are two approaches to inpainting that we've found: pixel and vectorized inpainting.

2.1. Pixel-based Inpainting

In Pathak et. al., a GAN was used to fill in a pixel mask based on surrounding pixels in the original image [14]. These excel in scenarios that require textural synthesis such as removing objects from an image [11, 12]. However, GANs struggle in areas where an irregular mask is used and often lack diversity when inpainting a scene [10, 12, 15]. Thus, Zheng et. al. and Zhao et. al. developed VAE-based networks which provide far more control of the generated image during inference and in turn allow for a balance between deterministic and diverse pixel generation [20,21]. In a similar approach, Lugmayr et. al. proposed a diffusion-based model which excels in inpainting a diverse set of images while remaining mask-agnostic [12]. Commercial examples of these models are seen in Photoshop and Google Photos.

2.2. Vectorized Inpainting

Vectorized inpainting methods have also seen some interest. Simhon et. al. developed a Hierarchical Hidden Markov Model which drew curves based on a combina-

tion of the previous curve and a set of predefined constraints [16]. More recent approaches focus on deep learning methods specifically Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM) to generate full sequences [4]. Ha et al. used a combination of LSTMs and Mixture Density Models (MDN) to generate fake Chinese characters by generating the strokes [7].

The QuickDraw! dataset that we've chosen to use has been analyzed multiple times. In the original paper for QuickDraw!, Ha et. al. applied a Variational Autoencoder in a vectorized format [8]. Building upon their work, Ribeiro et. al. replaced the VAE with an encoder-decoder Transformer based on Vaswani et. al. Most similar to our work, George et. al. proposed a decoder-only approach that tokenizes the points in the drawing first and then used a branched Transformer decoder to predict both the point positions and the pen state [3].

Our approach replaces the Transformer decoder approach of George et. al. with a model based on the Mamba layer [3]. Mamba is designed as an alternative approach to the Transformer blocks [5]. Recent research has shown that Mamba blocks can be used as a viable approach in sequence-to-sequence scenarios [2, 18]. We believe it will uniquely succeed in our scenario of needing mask-agnostic inpainting with a vectorized methodology. Concerning our real-time constraint on inference, we have not seen any work that uses Mamba in a real-time scenario. Our paper aims to explore this avenue of research.

3. Methods

3.1. Data Input

The model is given a sequence of points $p_1, p_2, ..., p_n$. Each point p_i is formatted with the same 5 data point format explained in Ha et. al. [8]. Essentially, we encode offset from the previous point in the x and y directions in the first two indices, and then store a one-hot encoding of whether the drawing pen is down on the canvas, the drawing pen is lifted from the canvas, and whether the drawing is completed in the remaining 3 indices. These sequences are then right padded with our end token $[0, 0, 0, 0, 1]$ (representing pen does not move, drawing is finished) to be of equal length. For all our experiments, we train on these 5 classes from The QuickDraw! Dataset: airplane, cat, headphones, marker, and sailboat.

3.2. Model Architecture

We employ a branched decoder model based on George et. al., comprised of blocks that include the Mamba layer (S6) [3]. Our models start with a linear layer that projects the input into the feature dimension. As a form of regularization, we then include a Dropout layer with a probability of 0.1. We then have i blocks as described in Gu et. al. with an $Add \rightarrow LayerNorm \rightarrow Mamba$ layout [5]. We then separate the decoder into two branches: offset regression and pen state inference. The offset regression branch is comprised of j blocks and the pen state inference branch has k blocks. The offset branch ends with a hidden Linear layer activated with ReLU and then another Linear layer that projects the hidden state into the offset predictions. The pen state branch similarly has a hidden Linear layer activated by ReLU and then another Linear that projects the hidden state into the pen-states. A final softmax activation is then used to transform the pen state logits into probabilities. We use the output of the offset branch as is and we pick the highest pen state probability as the pen state. To account for extra end-token introduced via padding, we performed masked versions of Mean Squared Error (MSE) and Categorical Cross Entropy that only consider pen-down and pen-lift points.

3.3. Training

We trained our models via self-supervised learning. Given a sequence of length n, we input the first $n-1$ points into the model and then compare that output to the last $n-1$ points. We train all our models over 20 epochs with a batch size of 256. We use an AdamW optimizer with an learning rate of 0.001, betas of $(0.99, 0.95)$, and weight decay calculated by $\sqrt{\frac{\text{batch size}}{\text{batches} * \text{epochs}}}$. We use a cosine learning rate scheduler with a minimum learning rate of $1e-5$. These hyperparameters were used in the Mamba paper [5]. We also implement an early stopping mechanism if the offset loss doesn't decrease by at least 30 throughout 3 epochs. All experiments were trained on a GTX 1080 with 8GB of VRAM, an i7-7700k, and 32 GB of RAM. Training the models used all the VRAM but all our data took a mere 3 GB of RAM space.

3.4. Real-time Demo

Our demo is comprised of an interactive Tkinter canvas where the user sketches on one canvas and a display of the finished sketch prediction is shown on another canvas. A user will use the mouse to draw lines on the canvas. We keep track of the lines drawn in memory as $L = [L_1, L_2..., L_n]$ where L_i is an array of pixels drawn black. On every mouse event, we simplify strokes using RDP (just as Google did for QuickDraw!) and then convert the lines to our 5 data point format [8]. This is then inputted into the model. The model generates for 60 strokes or until the pen end signal is seen. The model output is then drawn as lines on a separate display. Users will have the option to clear the interactive canvas and replace their drawing with the model's finished sketch and continue drawing.

The demo runs on the Intel UHD Graphics of an i9-11900H.

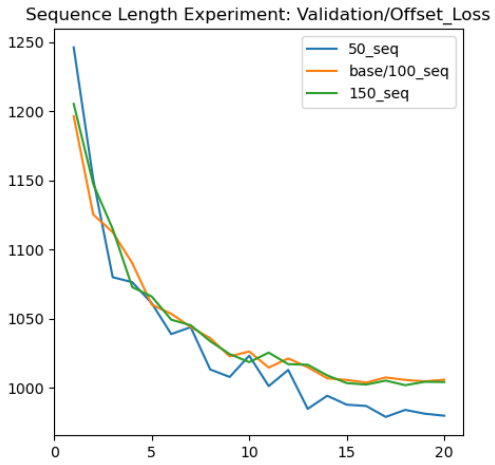

Figure 1. Val Offset Loss of the [50, 100, 150] max sequence models

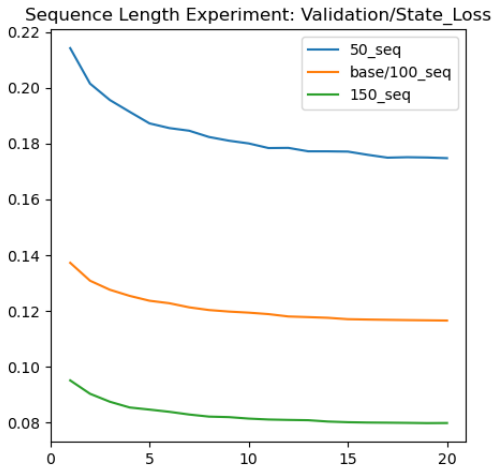

Figure 2. Val State Loss of [50, 100, 150] max sequence models

4. Experiments

Due to computational limitations, for each of our experiments, we train on a subset of 5 classes of the QuickDraw! dataset. For each of our experiments, we graph the MSE loss and Categorical cross-entropy loss against the number of epochs. The base model we train has a main branch size of 4 blocks with a branch size of 2 blocks in both the offset and pen state branches.

4.1. Max Sequence Length Experiments

Using our base model, we train along three different max sequence lengths: [50, 100, 150].

Looking at offset loss in Figure 1, we notice that the sequence does not play a major factor in improving the offset regression but does show significant improvement in calculating the pen state. The best model in our sequence length experiment for offset regression is the one trained on a max sequence length of 50. As part of our data loading step, we cull any sketches larger than the max sequence length. Thus, this model was trained on simpler sketches and didn't need to validate against the longer sketches the other models had to. However, even though the max sequence length increases, the loss differential is not much higher. Thus, we can conclude that Mamba blocks do work well with larger sequence lengths and scales to accommodate them.

Looking at Figure 2, we can notice that the higher the max sequence length, the lower the cross entropy loss of the predicted pen state is. We believe one of the reasons for this is that, across this experiment, each model would get the same samples wrong, but because the larger sequence length models had more samples, it got more samples right in comparison. Thus, the loss improves as the sequence length grows.

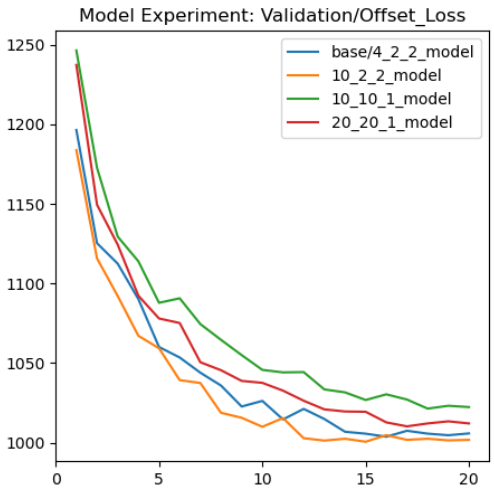

Figure 3. Val Offset Loss of different sized models

4.2. Branch Size Experiments

Using our base model, we train different branch sizes: $[4, 2, 2]$, $[10, 2, 2]$, $[10, 10, 1]$, $[20, 20, 1]$. Again, as a note, we were unable to train the larger models specifically $[10, 10, 1]$ and $[20, 20, 1]$ due to VRAM limitations, so these were trained with an internal embedding dimension of 64 not 256 3.3.

From our model depth experiments in Figure 3, we can see that model depth plays a slight role in improving the offset loss. However, the internal embedding dimension remains the key factor in determining which model performs the best at offset regression. In Figure 4, model depth seems to have little effect on the loss curves. Instead, the curves group by internal embedding dimension.

We believe this relationship exists because increasing the embedding dimension increases the number of trainable pa-

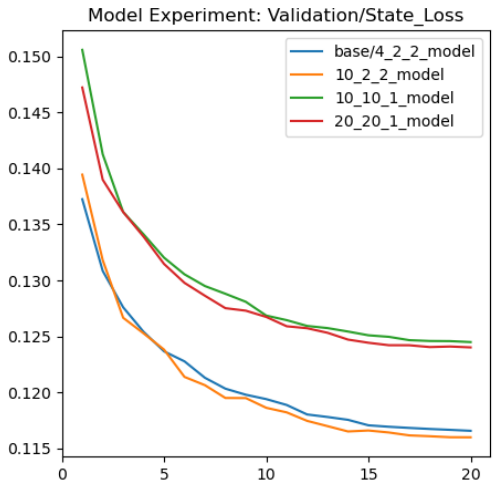

Figure 4. Val State Loss of different sized models

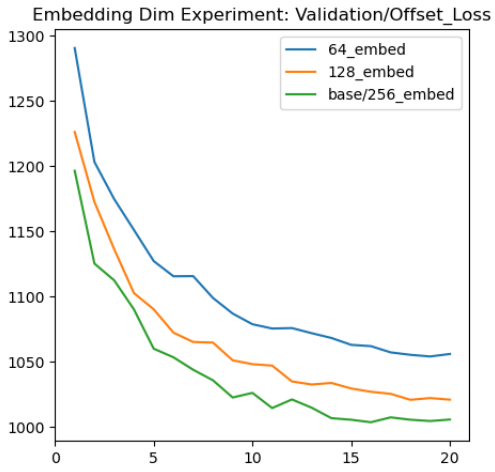

Figure 5. Val Offset Loss of models with different internal embedding sizes

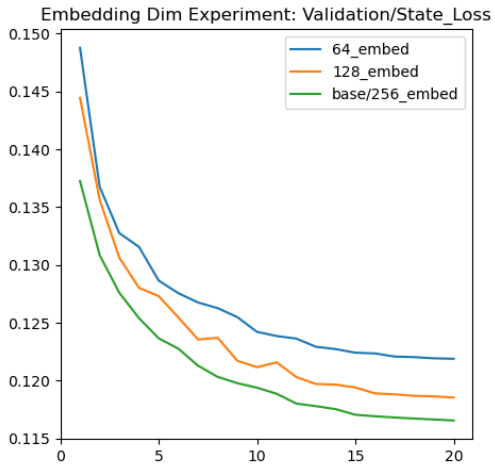

Figure 6. Val State Loss of models with different internal embedding sizes

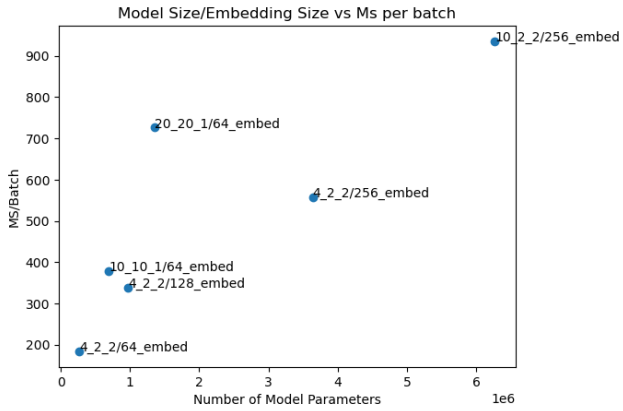

Figure 7. Scatter Plot of Model Parameters vs MS/batch

rameters far more than adding additional Mamba layers. Thus, with more parameters, the models can learn more specific patterns and predict future points more accurately [9].

4.3. Embedding Dimension Experiment

Using our base model, we train different internal embedding dimensions: $64, 128, 256$.

In both Figure 5 and Figure 6, we notice the relationship that as the internal embedding size increases, the loss curves across both the offset regression and pen state classification decreases. However, we notice a slight decrease in the spacing between the 64_embed and the 128_embed curves compared to the 128_embed and base curves. This possibly hints at a cap to the size of this internal embedding dimension. However, it is unclear whether this small difference is statistically significant. If there was a cap to the embedding dimension, it's certainly somewhat related to the number of layers because, in the Mamba paper, the authors were able to train a 24-layer model with an internal embedding dimension of 2048 [5]. Perhaps, in future work, we can explore using larger embedding sizes with shallower models.

4.4. Size vs Training Speed Experiment

We compare each of the above experiments based on parameter size and ms/batch. We also compare training on different sequence lengths and their corresponding ms/batch. As explained before, all our experiments were trained with a batch size of 256.

As seen in Figure 7, we can see a linear relationship between the number of trainable parameters a model has and the time it takes for a forward and backward pass. This relationship is self-evident as more trainable parameters mean

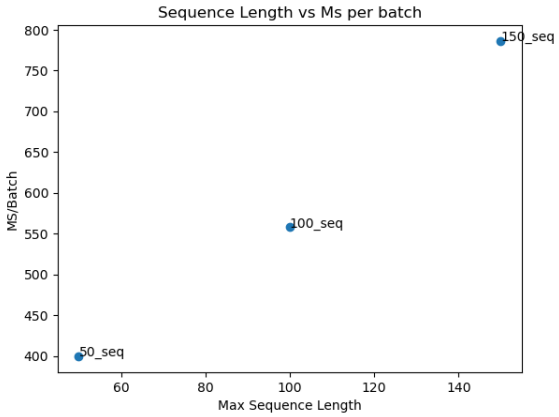

Figure 8. Scatter plot of max sequence length vs MS/batch

more calculations in both the forward and passes. However, the $[20, 20, 1]$ and $[10, 10, 1]$ models are outliers in this relationship. We believe they show increased time per batch because of the increased number of Mamba blocks they train.

We see another linear relationship in Figure 8. Again, it's self-evident that the more data passed through the model, the longer the model will take to train and output. The linear relationship also validates Gu et. al.'s claim of the $O(n)$ scaling of Mamba concerning sequence length [5].

5. Demo Image Analysis

We successfully developed a demo to showcase our model's real-time performance. In practice, we see instantaneous completion of user sketches on every mouse event. On average, the time the model generation took, running on the integrated graphics of an i9-11900H, was 0.2 milliseconds, which is considered instantaneous [13].

Turning to real-world performance, our model had significant limitations. First, it was only trained in a handful of classes and could not extrapolate any semantic embeddings to complete drawings it had never seen before. Second, while the model excels at general shapes, it struggles with the semantic placement of class-specific lines such as the whiskers of a cat or the location of the sail of a sailboat. Finally, once the sequence length was exceeded or the model had finished drawing a class, the rest of the image were short curves or spirals indicating that the model could not extrapolate any further than what it was trained on.

We did notice some interesting generations such as generating sunglasses around the eyes of a cat (unfortunately, the image was cleared before it could be recorded). We also observed the model complete an image of a skateboard when not trained specifically in the skateboard class. However, we believe these generations are unintended mistakes that happen to mirror real-world sketches. The vast majority of the "mistakes" were unintelligible lines.

We believe these errors can be attributed to a lack of training classes. With less variety in our training samples, our models cannot extrapolate to different classes or classes it's never seen again [9]. Thus, in future works, we aim to train with a larger set of classes or all samples in the Quick-Draw! Dataset to see the greatest generality. We also believe some of the instability within our model's generations is a result of our method of offset regression. As seen in our related works 2, we mainly picked a branch model with offset regression because George et. al. saw decent performance with their transformer-based approach [3]. However, other papers exploring Mamba often output softmax probabilities and pick the next token from a set of tokens [2, 18]. In future work, we would explore outputting the probability of integer offsets for the x and y dimensions and selecting the offsets with the highest probability. Using this method, most similar to PixelRNN, would utilize Mamba in a more standardized use rather than our current implementation [17].

5.1. Example Demo Images

Below are some example image completions from our demo during a poster fair. In each image, the bottom canvas represents the user input (drawn with a mouse), and the top canvas represents the completed image (generated by our 10_2_2_model).

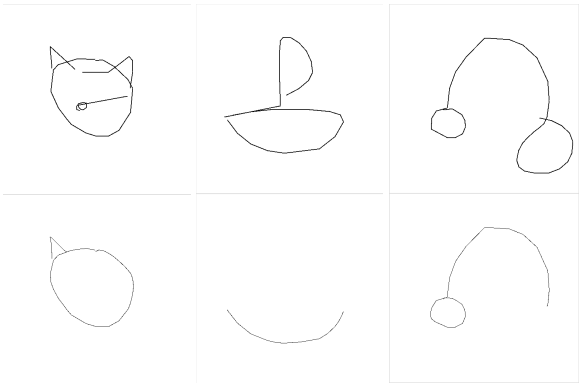

Figure 9. Example Demo Images

6. Discussion

In this paper, we explored using the new Mamba architecture in a real-time context. To research this subject, we proposed building a real-time doodler based on Mamba blocks. We cited their similar performance to other sequence-to-sequence architectures like Transformers and $O(1)$ inference speed as reasons why Mamba may succeed. To evaluate our models, we calculated the loss in two separate contexts: MSE of the x and y offsets and the categorical cross-entropy of the pen state. To analyze Mamba in various contexts, we conducted experiments regarding max se-

quence length, decoder depth, embedding dimension, and overall speed. These experiments revealed that in every context, Mamba-based models were able to run in a real-time context. In addition, they corroborated claims made by Gu et. al. about the speed of Mamba models as well as their performance given model depth and model embedding dimensions [5]. By developing a demo, we evaluated the real-world performance of our experiments. Our demo excelled in its real-time performance providing instantaneous inpainting from input. However, our models struggled with the semantic placement of certain lines and would often spiral when exceeding their trained sequence length. We believe that Mamba's struggle with drawing can be attributed to our usage of offset regression. Thus, we proposed encoding offsets as classes and predicting which offsets are most likely similar to PixelRNN which predicts which pixels are most likely [17]. We are excited to see other uses of Mamba in real-time contexts.

References

[1] EUROSIM 2019. ARGESIM, 2019. 1

[2] Raunaq Bhirangi, Chenyu Wang, Venkatesh Pattabiraman, Carmel Majidi, Abhinav Gupta, Tess Hellebrekers, and Lerrel Pinto. Hierarchical state space models for continuous sequence-to-sequence modeling, 2024. 2, 5

[3] Michael George and Kevin Smith. Sketch-transformer, 2021. 2, 5

[4] Alex Graves. Generating sequences with recurrent neural networks, 2014. 2

[5] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2023. 1, 2, 4, 5, 6

[6] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces, 2022. 1

[7] David Ha. Recurrent net dreams up fake chinese characters in vector format with tensorflow. *blog.otoro.net*, 2015. 2

[8] David Ha and Douglas Eck. A neural representation of sketch drawings, 2017. 2

[9] Ranjay Krishna and Sarah Pratt. Lecture 4: Neural networks and backpropagation, 2024. 4, 5

[10] Guilin Liu, Fitsum A. Reda, Kevin J. Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image inpainting for irregular holes using partial convolutions, 2018. 1

[11] Hongyu Liu, Bin Jiang, Yibing Song, Wei Huang, and Chao Yang. Rethinking image inpainting via a mutual encoder-decoder with feature equalizations, 2020. 1

[12] Andreas Lugmayr, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. Repaint: Inpainting using denoising diffusion probabilistic models, 2022. 1

[13] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 267–277, New York, NY, USA, 1968. Association for Computing Machinery. 5

[14] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting, 2016. 1

[15] Jialun Peng, Dong Liu, Songcen Xu, and Houqiang Li. Generating diverse structure for image inpainting with hierarchical vq-vae, 2021. 1

[16] Saul Simhon and Gregory Dudek. Sketch Interpretation and Refinement Using Statistical Models. In Alexander Keller and Henrik Wann Jensen, editors, *Eurographics Workshop on Rendering*. The Eurographics Association, 2004. 2

[17] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks, 2016. 5, 6

[18] Junxiong Wang, Tushaar Gangavarapu, Jing Nathan Yan, and Alexander M Rush. Mambabyte: Token-free selective state space model, 2024. 2, 5

[19] Jian Wang, Chenhui Gou, Qiman Wu, Haocheng Feng, Junyu Han, Errui Ding, and Jingdong Wang. Rtformer: Efficient design for real-time semantic segmentation with transformer, 2022. 1

[20] Lei Zhao, Qihang Mo, Sihuan Lin, Zhizhong Wang, Zhiwen Zuo, Haibo Chen, Wei Xing, and Dongming Lu. Uctgan: Diverse image inpainting based on unsupervised cross-space translation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. 1

[21] Chuanxia Zheng, Tat-Jen Cham, and Jianfei Cai. Pluralistic image completion, 2019. 1