SUNMASK: Appendix and Supplementary Material

A Generative Co-Creation and Possible Ethical Concerns

The application areas shown in SUNMASK have clear areas of concern (in terms of ethics), we describe them in detail as well as possible mitigation strategies.

A.1 Music

A chief concern in generative co-creation as demonstrated by our music sampling, is direct plagiarism through the training corpus or indirect plagiarism of outside work. Direct plagiarism has a number of mitigation strategies, either with exact matches, approximate dataset matches [10] (note sequences across voices, regardless of duration), or secondary tools such as automated copyright matchers. The latter category (automated copyright matchers) can also be used for the most difficult plagiarism - indirect plagiarism.

Due to most Western music sharing similar underlying rules and structure, it is possible to accidentally stumble upon a copyrighted work without a version of that work ever appearing in the training corpus. This goes especially for models trained on foundational scores, for example the underlying harmonic rules J.S. Bach followed and popularized underwrite a vast swath of the classical canon. Direct debut of any generative co-creation tool should have at least some consideration for tagging or labeling possible matches and conflicts, letting creators inspect the relevant matches to decide for themselves if there is an issue which warrants modification. Given the complexities of musical copyright, there is no clear cut automated solution but using recognition tools to provide information can help mitigate surprise issues for end users [4].

The music dataset used in this work (JSB) is fully in the public domain, as such the ethical concerns listed above are minimized, and any secondary issues related to music copyright are unlikely to be a problem with the direct output of this model. The short, quantized MIDI-style output from SUNMASK is a not a suitable format for general listening, needing substantial post-processing, combination, and interpretation in order to form a musical piece [12].

A.2 Text

There are many complexities around generative modeling of language, and especially with generative co-creation of text. The particular datasets and schemes used in this work are relatively limited compared to more direct and large scale applications of language models, however it is always a key concern to think about the limitations and biases of the underlying datasets used to train these models.

Given the propensity for using SUNMASK and related methods to infill given context, certain applications should have strong investigations into the biases and private information present in the underlying data. Imputing missing information in order to strengthen downstream classification (not directly shown in this work, but certainly possible) can be problematic with respect to imputed features amplifying underlying biases in the training corpus, or violating user privacy. Many mitigation and detection strategies proposed by researchers in fairness, bias, and privacy in machine learning [11] should be directly applicable to SUNMASK if deemed necessary, given the commonality of the modeling and training schemes to other well studied methods such as AR transformers, and deep learning more generally.

The text datasets used in our experiments are a common benchmark, chosen to enable comparison to existing work. Any underlying biases or issues with models trained on these particular datasets will be shared across *many* generative language models, and any proposed corrections specific to these datasets should be directly applicable to SUNMASK.

B Convolutional SUNMASK Model Hyperparameters and Training Information

Training	
Input channels	4
Pitch count	57
Sequence length	128
Training steps	50000
Batch size	1
Unrolled steps	2
Hidden size	64
Kernel size	(4, 57)
Block scales	(1, 1, 2, 4)
Residual layers per block	3
Optimizer	Adam
Learning rate	1E-4
Total parameter count	417M
Downweight multiplier	.75
Downweight learning rate steps	5000

B.1 Architecture Design

Attention is applied on the innermost U-Net block size as well as the middle block, with 1 attention head [9]. Convolutions are used in all resampling, and all resampling happens only on the time axis, making the Attentional U-Net effectively a 1-D architecture. However, rather than learning both instrument and pitch relations across channels, we isolate pitch relations and instrument relations into separate axes of the overall processing, the "width" and "channels" axes, respectively assuming $(N, C, H, W) == (N, I, T, P)$ axes. As is standard in many U-Net designs, we double the number of hidden values for layers every time the resolution is halved, with the reverse process being used when upsampling. Though the parameter count here is large, it is similar in spirit to other approaches to small datasets on text [1].

Sampling	
Temperature	.6
Steps	$2 \times I \times T = 2 \times 4 \times 128$
Mask dwell	1
Active balance	False
Final mask dwell	0
Keep prob	"triangular"
Sampling	typical
Top k	3
Top p	False
Intermediate noise	False
Mask max	.999
Mask min	.001

B.2 Sampling Details

All parameters and sampling designs were tuned based on generated sample quality, rather than direct tuning to the grading function used for final metric calculation. It is likely that these numbers could be greatly improved, but tuning directly against this metric may also result in less musical samples that exploit quirks in the metric calculation.

During sampling we have a number of additional parameters to set. Crucially, the use of typical sampling [8] and strong filtering (either small top-k, small top-p, or both) resulted in generally stronger samples than both the equivalent, or typical sampling with looser settings. We note that the

importance of this setting is demonstrated in the paper introducing typical sampling, in the difference between settings for summarization and story generation. These settings also interact heavily with the temperature setting.

When typical sampling top-k values or top-p values are too large, we see samples end up with the same result for either typical or standard sampling, so setting these filters is critical to see the full impact of typical sampling.

Triangular keep probability is described in SUNDAE, and we utilize it here as well, linearly ramping accept probability from 0 to 1 at $\frac{steps}{2}$, ramping back down to 0 at $\frac{steps}{}$. The keep probability schedule excludes the optional "final mask dwell", which we only utilize alongside intermediate noise. We also found fixed keep probabilities (such as $.33$, $.5$, and even 1.0) also performed well.

Mask proposals follow the scheme proposed by Coconet, sampling bernoulli masks with probability p, ramping from the mask min to mask max, over the total range of $steps$. These masks are further combined with pre-specified masks, specifically we allow two types of secondary specification. Focus masks hold the input value fixed at every diffusion step, and always specify a mask value of 1 in the model input. Keep masks allow a mask value of 0 or 1 (depending on the bernoulli random sampled mask) in the model input, but the value will be reset to the specified input value at each step of diffusion. The allowance of these two different mask types is unique to SUNMASK, and seems to have a large impact on the sampled outcome based on our testing.

When using intermediate noise, it is beneficial to set a final mask dwell, which holds the last mask (which is set to all 1) constant and then samples repeatedly to form final corrections. During final mask dwell, we set accept probability to 1, as triangular sampling would by default set accept probability to near 0. Intermediate noise is effectively disabled everywhere the mask is 1, so this is similar in spirit to noise tapering in other applications using gaussian style noise, and allows the SUNMASK learned improvement operator to make small changes and fixes to the general "skeleton" of the proposed sample.

C Transformer SUNMASK Model Hyperparameters and Training Information

Training	
Pitch count	57
Training steps	120000
Sequence length (JSB)	256
Batch size (JSB)	20
Unrolled steps	2
Transformer layers	16
Embedding dim	512
Transformer input dim	512
Transformer hidden dim	2048
Transformer attention heads	8
Transformer head dim	64
Optimizer	Adam
Learning rate max	5E-5
Learning rate max	5E-6
Ramp up steps (min to max)	5000
Ramp down steps (max to min)	100000
Gradient clip (value)	3
Total parameter count	50M

There is a large discrepancy in model parameter count between our best performing convolutional models for JSB, and our best transformers. Training larger transformers can work well for generation [1], but our large parameter transformers (on the order of 400M parameters) had poor generative performance on JSB. Given the foundational work in SUNDAE, it is clear that it should be possible to train these larger transformer models well, and finding the correct recipe may drastically improve the quality of the transformer generated musical examples.

Pitch size / vocabulary size, sequence length, and batch size changed for the transformers used in the text experiments. Notably, we use vocabulary size $5.7k$, sequence length 52, batch size 48 for EMNLP2017 News and vocabulary size 27, sequence length 64, batch size 20, and a slightly extended training step length of 150000 for text8.

D Pseudocode Loss for Convolutional SUNMASK, Training Loop, Example Model API

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
losses = []

N = batch_size
data_random_state = np.random.RandomState(2122)
gumbel_sampling_random_state = np.random.RandomState(3434)
corruption_sampling_random_state = np.random.RandomState(1122)

def gumbel_sample(logits, temperature=1.):
    noise = gumbel_sampling_random_state.uniform(1E-5, 1. - 1E-5, logits.shape)
    torch_noise = torch.tensor(noise).contiguous().to(device)

    # max indices
    maxes = torch.argmax(logits / float(temperature) - torch.log(-torch.log(torch_noise)),
                         axis=-1, keepdim=True)
    one_hot = 0. * logits
    one_hot.scatter_(-1, maxes, 1)
    return one_hot

def get_random_pitches(shape, vocab_size, low=0):
    random_pitch = torch.tensor(corruption_sampling_random_state.randint(low=low,
                                high=vocab_size, size=shape)).type(torch.LongTensor).to(device)
    return random_pitch

def corrupt_pitch_mask(batch, mask, vocab_size):
    reduced_batch = torch.max(batch, dim=-1)[1]
    random_pitches = get_random_pitches(reduced_batch.shape, vocab_size)
    one_hot_random_pitches = 0. * batch
    one_hot_random_pitches.scatter_(-1, random_pitches[..., None], 1)
    corrupted = (1 - mask[..., None]) * one_hot_random_pitches + (1 * mask[..., None]) * batch
    return corrupted

def build_logits_fn(vocab_size, n_unrolled_steps, enable_sampling):
    def logits_fn(input_batch, input_mask, input_noise_x_based_on_mask=False):
        def fn(batch, mask, noise_x_based_on_mask=False):
            logits = model(batch, mask, noise_x_based_on_mask)
            return logits

        def unroll_fn(batch, mask, noise_x_based_on_mask=False):
            samples = corrupt_pitch_mask(batch, mask, vocab_size)
            all_logits = []
            for _ in range(n_unrolled_steps):
                logits = fn(samples, mask, noise_x_based_on_mask)
                samples = gumbel_sample(logits).detach()
                # sanity check to be sure stacked outputs are correct
                assert samples.shape[0] == batch.shape[0]
                all_logits += [logits[None]]
            final_logits = torch.cat(all_logits, dim=0)
            return final_logits
```

```
        if enable_sampling:
            return fn(input_batch, input_mask, input_noise_x_based_on_mask)
        else:
            return unroll_fn(input_batch, input_mask, input_noise_x_based_on_mask)
    return logits_fn

def build_loss_fn(vocab_size, n_unrolled_steps=4):
    logits_fn = build_logits_fn(vocab_size, n_unrolled_steps, enable_sampling=False)

    def local_loss_fn(batch, mask):
        # repeated targets are now n_unrolled_steps
        repeated_targets = torch.cat([batch] * n_unrolled_steps, dim=0)

        # passing True noises internally, used at inference
        logits = logits_fn(batch, mask, False)

        logits = logits.reshape(n_unrolled_steps * batch.shape[0], logits.shape[2],
                                logits.shape[3], logits.shape[4])

        raw_loss = -1. * (nn.functional.log_softmax(logits, dim=-1) * repeated_targets)
        raw_masked_loss = raw_loss * torch.cat([(1. - mask[..., None])] * n_unrolled_steps,
                                       dim=0)

        reduced_mask_active = torch.cat([1. / ((1. - mask).sum(dim=1).sum(dim=1) + 1)] *
                                    n_unrolled_steps, dim=0)

        reduced_loss = reduced_mask_active * raw_masked_loss.view(n_unrolled_steps * N,
                                                                  I * T * P).mean(dim=1)
        loss = torch.sum(reduced_loss)
        # upweight by unrolling * time. since longer sequences will tend to have more
        # active mask elements, the loss gets downweighted more
        loss = n_unrolled_steps * T * loss
        return loss
    return local_loss_fn

u_loss_fn = build_loss_fn(P, n_unrolled_steps=n_unrolled_steps)
for i in range(n_train_steps):
    if not DO_TRAIN:
        break
    # mask drawn with random probability
    C_prob = data_random_state.rand(N)
    C_mask_base = data_random_state.rand(N, I, T)
    C = 1 * (C_mask_base < C_prob[:, None, None])
    C = (1. - C) # convert to 0 drop format
    # technically doesn't matter here with uniform prob
    C = C.astype(np.int32)
    # mask convention is set for 0 == drop (or noise) 1 == keep
    # shape is N, I, T

    # batch is an np array of shape (N, I, T), entries are integers in [0, P)
    indices = data_random_state.choice(train_tracks.shape[0], size=N)
    batch = train_tracks[indices]

    # x is of shape (N, I, T, P)
    batch = batch.reshape(N*I*T)
    x = np.zeros((N*I*T, P))
    r = np.arange(N*I*T)
    x[r, batch] = 1
    x = x.reshape(N, I, T, P)
```

```
x = torch.tensor(x).type(torch.FloatTensor).to(device)
C2 = torch.tensor(C).type(torch.FloatTensor).to(device)

loss = u_loss_fn(x, C2)
losses.append(loss.item())
loss.backward()
optimizer.step()
optimizer.zero_grad()
# adjust learning rate
if i % 5000 == 0:
    for g in optimizer.param_groups:
        g['lr'] *= .75
```

We use this same loss framework for both SUNMASK and SUNDAE experiments. The primary difference between SUNMASK and SUNDAE models is inside the model forward function - in SUNDAE we ignore the mask (unless doing internal noising at inference), passing the input to a layer which expects input channels of 4. In SUNDAE we concatenate the mask to the input along the channel axis, and pass to an input layer which expects input channels of 8.

Transformer versions of this setup are similar, with the primary differences being in the axes of computation - our default transformer model assumptions are shaped (T, B, P) so the internal logic is changed to match. The learning rate reduction scheme is also changed to the ramp-up, ramp-down learning rate schedule that is common when training transformer models.

E Inference Pseudocode for Convolutional SUNMASK

```
def torch_infer(y, C, model,
                keep_mask=None,
                n_steps=I * T,
                n_reps_per_mask=1,
                n_reps_final_mask_dwell=0,
                sundae_keep_prob=0.33,
                initial_corrupt=True,
                intermediate_corrupt=False,
                frozen_mask=False,
                use_active_balance=False,
                top_k=0, top_p=0.0,
                use_typical_sampling=False,
                temperature=1.0, o_nade_eta=3./4, seed=12, verbose=True):
    assert len(y.shape) >= 3
    if len(y.shape) == 3:
        assert y.shape[-1] != 1

    model.eval()
    rs = np.random.RandomState(seed)
    trsg = torch.Generator(device=device)
    trsg.manual_seed(seed)

    def lcl_gumbel_sample(logits):
        torch_noise = torch.rand(logits.shape, generator=trsg, device=device) *
             ((1 - 1E-5) - 1E-5) + 1E-5

        maxes = torch.argmax(logits - torch.log(-torch.log(torch_noise)),
                axis=-1, keepdim=True)
        return maxes

    def lcl_get_random_pitches(shape, vocab_size):
        random_pitch = torch.randint(low=0, high=vocab_size, size=shape,
                device=device, generator=trsg)
```

```
    return random_pitch

with torch.no_grad():
    x = torch.tensor(y).float().to(device)
    C = torch.tensor(C).long().to(device)
    if keep_mask is not None:
        keep_C = torch.tensor(keep_mask).long().to(device)

    C2 = torch.clone(C)
    alpha_max = .999
    alpha_min = .001
    eta = o_nade_eta

    x_cache = torch.clone(x)
    if initial_corrupt:
        x = lcl_get_random_pitches(x.shape, P).float()
        x[C2==1] = x_cache[C2==1]
        if keep_mask is not None:
            x[keep_C==1] = x_cache[keep_C==1]

    n_steps = max(1, int(n_steps))
    if sundae_keep_prob == "triangular":
        sundae_keep_tokens_per_step = [2 * x.shape[2] *
                        min((t + 1) / float(n_steps), 1 - (t + 1) / float(n_steps))
                         for t in range(int(n_steps))] +
                    [1.0 * x.shape[2] for t in range(int(n_reps_final_mask_dwell))]
    else:
        sundae_keep_tokens_per_step = [sundae_keep_prob * x.shape[2]
                                       for t in range(int(n_steps))] +
                [1.0 * x.shape[2] for t in range(int(n_reps_final_mask_dwell))]


    has_been_kept = 1. + 0. * x
    has_been_kept_torch = torch.tensor(has_been_kept).to(device)

    sampled_binaries = None
    for n in range(int(n_steps + n_reps_final_mask_dwell)):
        k = int(sundae_keep_tokens_per_step[n])
        if k == 0:
            # skip zero keep scheduled steps to speed things up
            # do it this way because very long schedules need small k values
            # which necessarily causes 0 to be more frequent
            continue
        fwd_step = n
        if n_reps_per_mask > 1:
            # roll mask forward
            fwd_step = int(fwd_step + n_reps_per_mask)
        p = np.maximum(alpha_min,
            alpha_max - fwd_step*(alpha_max-alpha_min)/(eta*int(n_steps)))

        if not frozen_mask:
            if n % n_reps_per_mask == 0:
                sampled_binaries = torch.bernoulli(1. - (0 * C + p), generator=trsg).long()
                C2 += sampled_binaries
            if n > n_steps:
                # set final mask to all ones
                C2[:] = 1
        C2[C==1] = 1
```

```
        # expand things to one-hot representation
        x_e, C2_e = model.expand(x, C2, is_torch=True)
        # passing true will noise things
        logits_x = model(x_e, C2_e, intermediate_corrupt)

        # dont predict just logits anymore
        # top k top p gumbel

        if use_typical_sampling:
            logits_x = logits_x / float(temperature)
            filtered_logits_x = typical_top_k_filtering(logits_x, top_k=top_k, top_p=top_p,
                temperature=float(temperature))
        else:
            logits_x = logits_x / float(temperature)
            filtered_logits_x = top_k_top_p_filtering(logits_x, top_k=top_k, top_p=top_p)
        x_new = lcl_gumbel_sample(filtered_logits_x).float()

        # active balance
        p = has_been_kept_torch[:, :, :] / torch.sum(has_been_kept_torch[:, :, :],
            axis=2, keepdims=True)
        r_p = 1. - p
        r_p = r_p / torch.sum(r_p, axis=2, keepdims=True)

        if k > 0:
            shp = r_p.shape
            assert len(shp) == 3
            r_p = r_p.reshape(shp[0] * shp[1], shp[2])
            if use_active_balance:
                keep_inds_torch = torch.multinomial(r_p, num_samples=k,
                    replacement=False, generator=trsg)
            else:
                keep_inds_torch = torch.multinomial(0. * r_p + 1. / float(shp[2]),
                    num_samples=k, replacement=False, generator=trsg)

            keep_inds_torch = keep_inds_torch.reshape(shp[0], shp[1], -1)

            assert x_new.shape[-1] == 1

            for _ii in range(x.shape[0]):
                for _jj in range(x.shape[1]):
                    x[_ii, _jj, keep_inds_torch[_ii, _jj]] = x_new[_ii, _jj,
                                    keep_inds_torch[_ii, _jj], 0]
                    has_been_kept_torch[_ii, _jj, keep_inds_torch[_ii, _jj]] += 1
        else:
            pass
        x[C==1] = x_cache[C==1]
        if keep_mask is not None:
            x[keep_C==1] = x_cache[keep_C==1]

        C2 = torch.clone(C)
    return x
```

This inference code supports the variety of different options used and tested in the body of the main paper. Transformer inference is largely similar, again with the primary difference being the use of (T, B, P) axis notation, and changes to match this axis convention.

F Sampling Runtime

One chief drawback of the currently implemented SUNMASK models, primarily the convolutional SUNMASK used in JSB, is the time to sample. In part due to the parameter count, as well as the non-standard details of the architecture (kernel size in particular), sampling runs at roughly $8\times$ slower than real-time, taking approximately 4 minutes to generate a 38 second sample (4 voices, each with 128 steps, at 16 steps per measure quantization) on V100 GPUs. On A100 GPUs, this goes directly to 66 seconds to generate the same size sample, which gives some indication that simply updating hardware may radically improve the runtime of convolutional SUNMASK. Increasing batch sizes improves the effective amortized sample speed, but at an increase of latency. In addition, the non-causal nature of diffusion sampling means that pipelined sampling to reduce the effective latency felt by end-users is not easily applicable, compared to standard AR methods.

However there are many direct optimizations available for these architectures from both the computer vision literature at large, and specifically for symbolic music modeling [6]. More exploration of computational improvements, toward fully interactive use remain a key research direction.

G Code repository and samples player

We attach several folders of samples (in midi format) from our model for music, as well as the evaluation sentences for BLEU / self-BLEU testing on EMNLP2017 News as part of the supplemental material.

Full reproduction code and sample listening page can be found at https://github.com/SUNMASK-web/SUNMASK

H Creating a "Greatest Hits"

Music demonstrations of the model labeled "BachMock" transformer can be heard at https://alisawuffles.github.io/post/grading-function/. We find SUNMASK generations are qualitatively on a similar level as these sample generations, though some SUNMASK samples do fare poorly by the grading metrics. However the best SUNMASK samples have remarkably good grades, on a similar level as the best "BachMock" samples shown in the linked post, and indeed to a similar level as the data itself.

Of particular note is the high variance in the grade of all SUNMASK models compared to either Coconet, or the baselines. Given the existence of the grader function, it is possible to prune generations from our SUNMASK diffusion models to improve the overall output. Generating 200 samples from the best SUNMASK method, and pruning to only the top 20 overall grades, we see that it is possible to produce high quality subsets which rival the "BachMock" transformer and the dataset itself on this metric.

Model	Note	Rhythm	Parallel Errors	Harmonic Quality	S Intervals	A Intervals	T Intervals	B Intervals	Repeated Sequence	**Overall**
Bach GT	0.24 (0.15)	0.23 (0.14)	0.0 (0.69)	0.41 (0.2)	0.47 (0.28)	0.49 (0.22)	0.53 (0.24)	0.69 (0.4)	1.29 (0.88)	4.91 (1.63)
BachMock	0.37 (0.22)	0.26 (0.14)	2.16 (3.22)	0.54 (0.31)	0.53 (0.35)	0.71 (0.34)	0.73 (0.38)	0.89 (0.68)	1.86 (2.81)	8.94 (4.64)
SMc-T	0.57 (1.79)	0.69 (0.35)	1.28 (3.73)	0.93 (0.49)	0.80 (4.51)	0.99 (4.01)	1.20 (4.68)	1.40 (3.91)	1.81 (0.83)	13.43 (19.27)
SMc-T-BEST20-200	**0.39 (0.16)**	**0.53 (0.26)**	**0.0 (0.81)**	**0.68 (0.27)**	**0.59 (0.25)**	**0.88 (0.42)**	**0.80 (0.20)**	**0.71 (0.27)**	**1.44 (0.52)**	**7.16 (0.97)**

While selecting directly against this metric makes it a less useful ranking scheme for the various methods, listening to the resulting samples also reveals that this ranked subset are also qualitatively among the best of this cohort. We stress that this simple generate-and-test method could be used with all available models, and potentially as part of training itself, as in published work on augmented generative training [7].

Given that the best scoring example from Coconet has an overall grade of 12.26, the best SUNDAE example (SD-AT in main table) grade 7.78, best SUNMASK example from the small set (SMc-T in main table) grade 7.14, and the best SUNMASK example grade from the larger 200 set 4.93 it seems SUNMASK may be a better candidate for this kind of scheme due to higher generated sample variance. Generating more samples and then curating a top performing subset should yield better scores for all methods tested. Comparing this approach against a broader swath of high-performance, template based infilling methods [5, 3, 2] remains an important future direction.

References

[1] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. Character-level language modeling with deeper self-attention. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 3159–3166, 2019.

[2] Mason Bretan, Sageev Oore, Doug Eck, and Larry Heck. Learning and evaluating musical features with deep autoencoders. *arXiv preprint arXiv:1706.04486*, 2017.

[3] Mason Bretan, Gil Weinberg, and Larry Heck. A unit selection methodology for music generation using deep neural networks. *arXiv preprint arXiv:1612.03789*, 2016.

[4] Jean-Pierre Briot and François Pachet. Deep learning for music generation: challenges and directions. *Neural Computing and Applications*, 32(4):981–993, 2020.

[5] Gaëtan Hadjeres and Léopold Crestel. Vector quantized contrastive predictive coding for template-based music generation. *arXiv preprint arXiv:2004.10120*, 2020.

[6] Cheng-Zhi Anna Huang, Curtis Hawthorne, Adam Roberts, Monica Dinculescu, James Wexler, Leon Hong, and Jacob Howcroft. The bach doodle: Approachable music composition with machine learning at scale.

[7] Alisa Liu, Alexander Fang, Gaëtan Hadjeres, Prem Seetharaman, and Bryan Pardo. Incorporating music knowledge in continual dataset augmentation for music generation. 2020.

[8] Clara Meister, Tiago Pimentel, Gian Wiher, and Ryan Cotterell. Typical decoding for natural language generation. *arXiv preprint arXiv:2202.00666*, 2022.

[9] Alex Nichol, Prafulla Dhariwal, Aditya Ramesh, Pranav Shyam, Pamela Mishkin, Bob McGrew, Ilya Sutskever, and Mark Chen. Glide: Towards photorealistic image generation and editing with text-guided diffusion models. *arXiv preprint arXiv:2112.10741*, 2021.

[10] Alexandre Papadopoulos, Pierre Roy, and François Pachet. Avoiding plagiarism in markov sequence generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[11] Eric Michael Smith, Melissa Hall Melanie Kambadur, Eleonora Presani, and Adina Williams. " i'm sorry to hear that": finding bias in language models with a holistic descriptor dataset. *arXiv preprint arXiv:2205.09209*, 2022.

[12] Bob L Sturm, Oded Ben-Tal, Úna Monaghan, Nick Collins, Dorien Herremans, Elaine Chew, Gaëtan Hadjeres, Emmanuel Deruty, and François Pachet. Machine learning research that matters for music creation: A case study. *Journal of New Music Research*, 48(1):36–55, 2019.