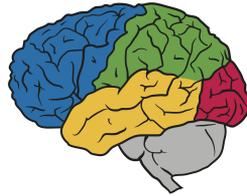


A Few Unusual Autoencoders

Colin Raffel



If you are confused about something, it's probably because I haven't explained it well and other people are probably confused too, so please feel free to stop me to ask questions. If something I'm describing seems like a bad idea or is not well-motivated, please let me know and I'll try to clarify.

A Few Unusual Autoencoders

Colin Raffel



David
Berthelot



Aurko
Roy



Ian
Goodfellow



Adam
Roberts



Jesse Engel



Fjord
Hawthorne



Douglas
Eck

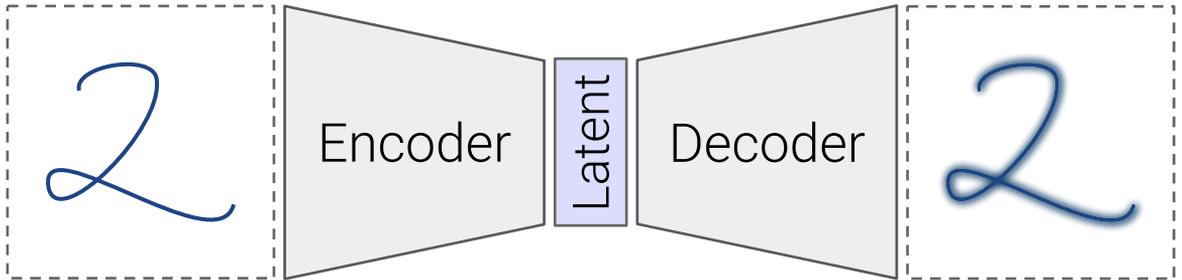


Ian
Simon

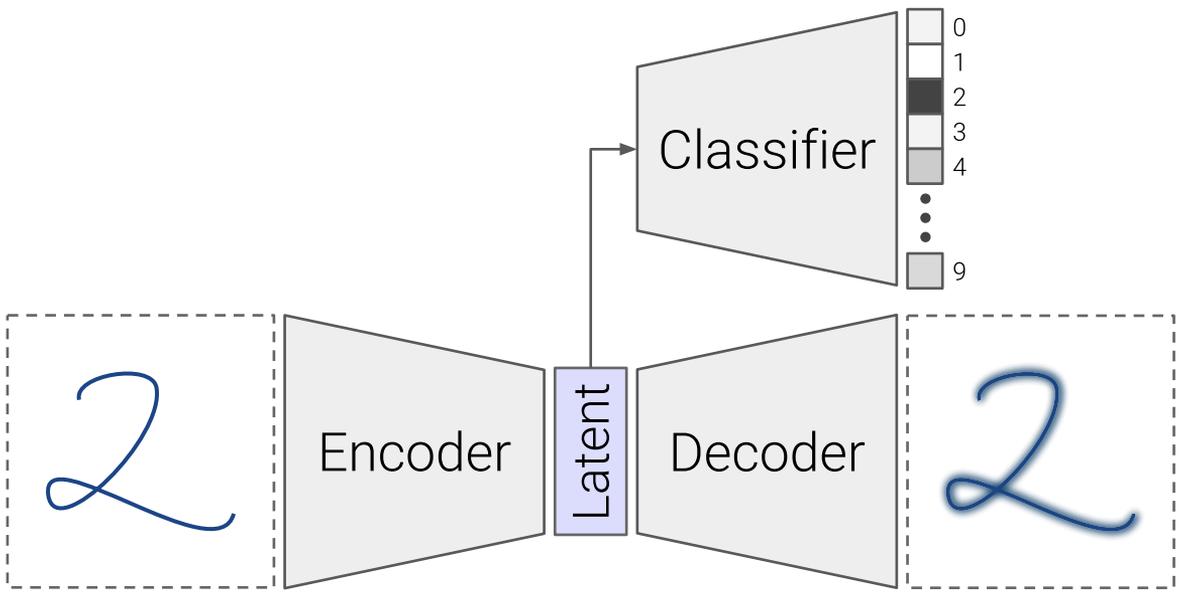


Rishabh
Agarwal

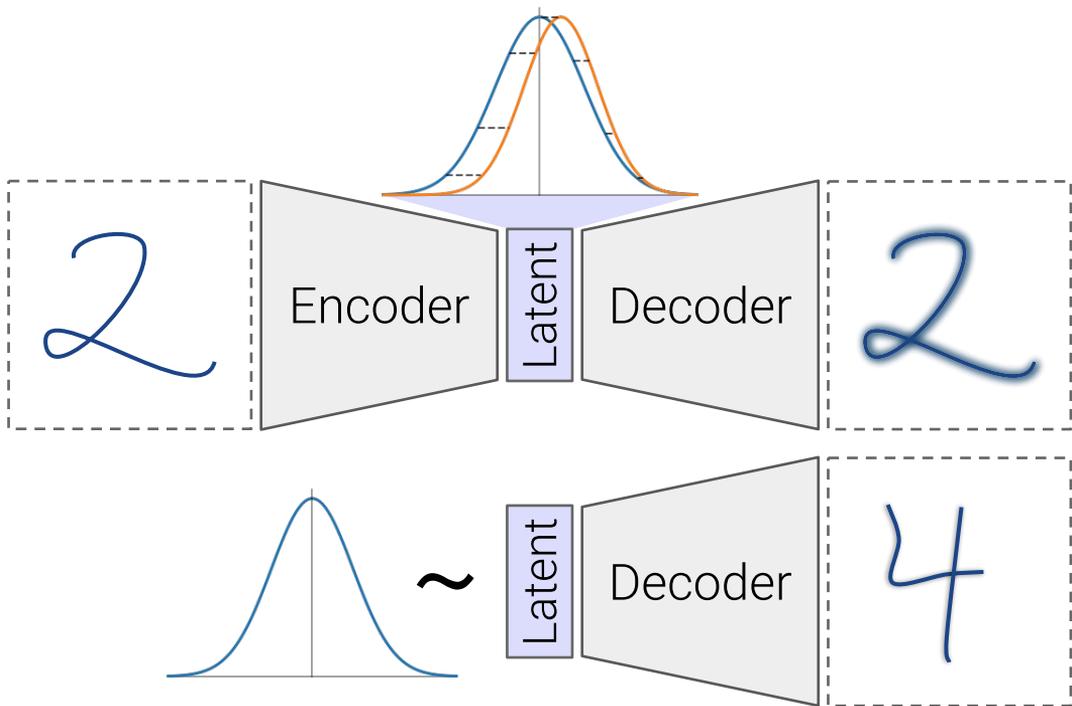
If you are confused about something, it's probably because I haven't explained it well and other people are probably confused too, so please feel free to stop me to ask questions. If something I'm describing seems like a bad idea or is not well-motivated, please let me know and I'll try to clarify.



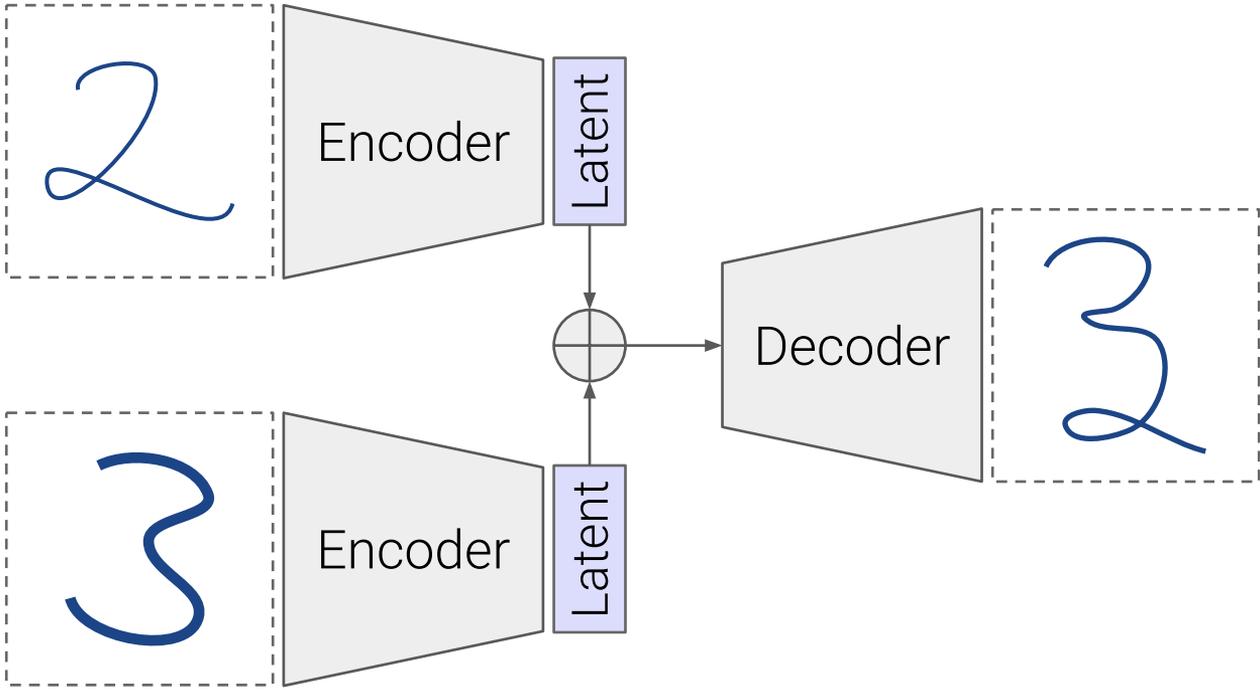
Today I'll be talking about autoencoders. The autoencoders is a simple model structure whose goal is to reconstruct its input. The input is first mapped to a latent code, which is typically lower-dimensional than the input. The encoder tries to encode all of the useful information about the input into into the latent code, and the decoder tries to reconstruct the input based only on the information in the latent. The model is typically trained against a reconstruction error cost such as mean squared error, which measures how closely the reconstruction matches the input.



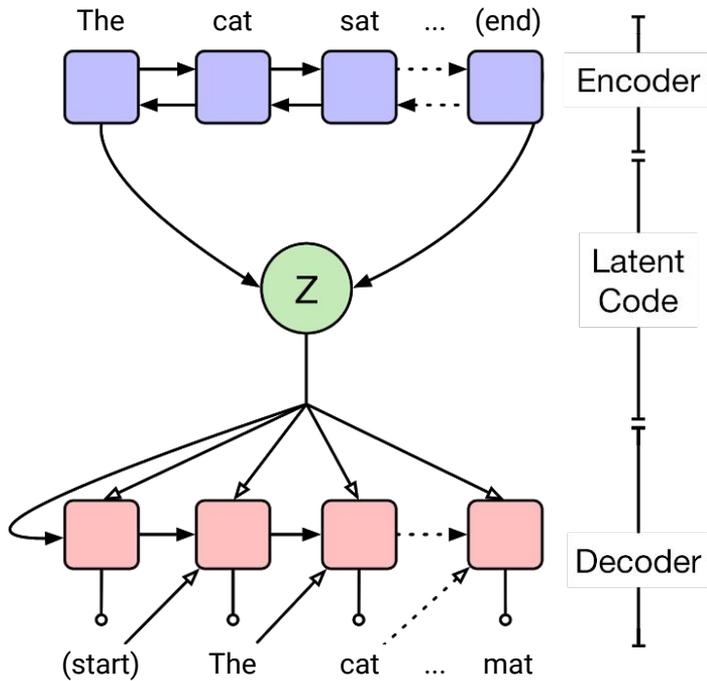
Since autoencoders are trained without labels, they are a popular unsupervised learning technique. One possible way to use them for unsupervised learning is to train them to reconstruct their input, and then use the latent code as a learned representation for a downstream task, like classification. Intuitively we might think this would work because the autoencoder learns a compressed representation of the input, where it has discarded noisy factors of variation which are not useful for reconstruction.



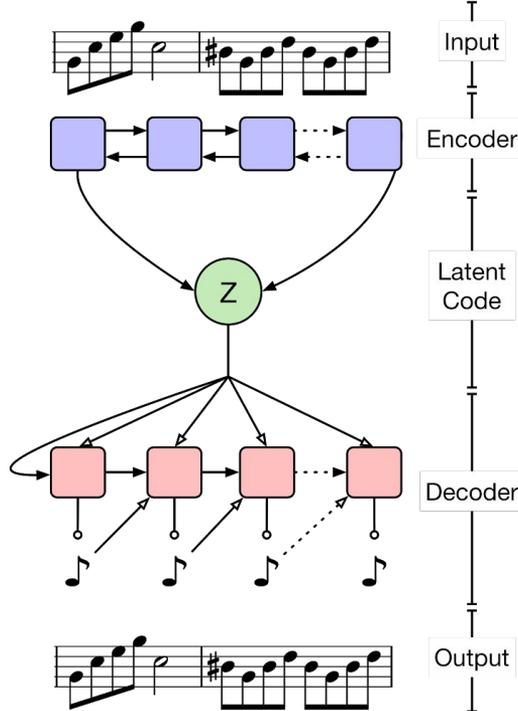
Another popular way to use autoencoders is to enforce specific structure on their latent code. Probably the most common framework for this is the variational autoencoder, which treats the encoder as an inference model which infers the parameters of a posterior distribution over latent codes, and treats the decoder as a generative model which maps samples from the distribution back to the data. The posterior is encouraged to fit a predefined prior distribution via an additional loss term. Taken together, this loss term with the reconstruction loss forms a lower bound on the log-likelihood of the data, which we maximize. In this diagram we're showing the prior distribution as a Gaussian in dark blue, and the predicted posterior distribution as orange on the top. The additional loss term encourages the orange distribution to match the blue distribution. When this approach is done successfully, we can draw samples from the prior distribution and decode them to sample new points in data space, as shown in the bottom.



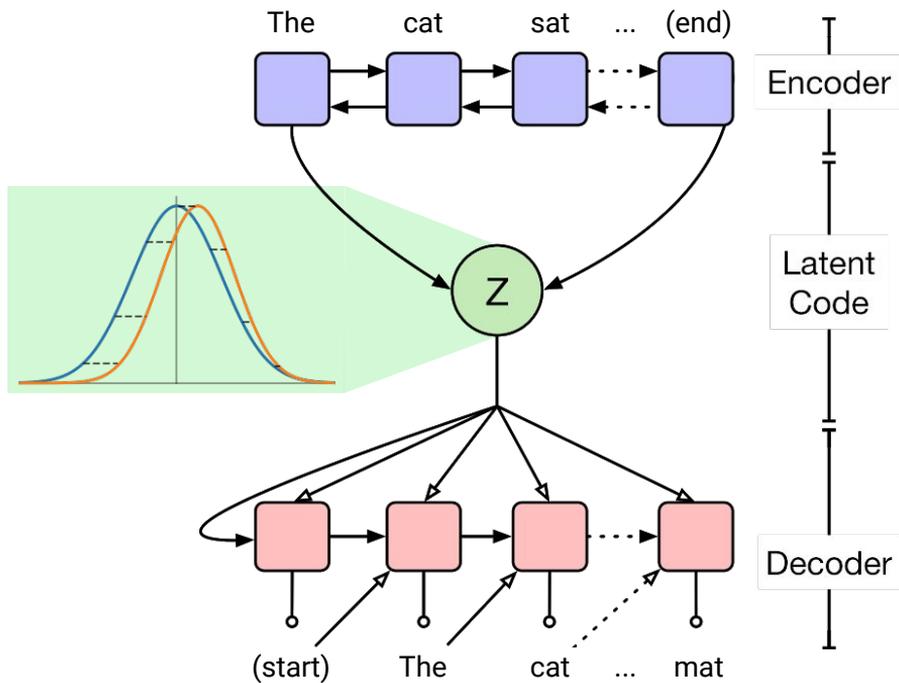
In some cases, autoencoders can facilitate semantic manipulation of data. One possible means for doing this is by combining latent vectors for different datapoints and decoding the result. Here, for example, by mixing the latent vectors for a two and a three, we get a two-three hybrid. People often call this interpolation, because we are interpolating between latent codes and decoding the result. This behavior is often only possible when the latent space has some structure, for example that nearby latent codes decode to similar datapoints.



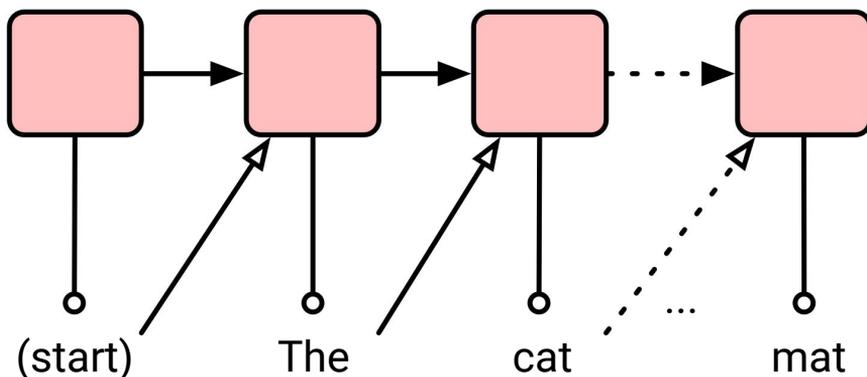
So far I've shown autoencoders which operate on images. We can also autoencode things like sequences, like text as shown here. In this case, the encoders and decoders are architected specifically to deal with sequential data; in the most common case, this means that they are recurrent neural networks. In this setup, the encoder RNN processes a sequence until it produces a fixed-length vector representation of it. The fixed-length vector is then used to compute the latent code. The code is then used to initialize the decoder RNN which generates the sequence autoregressively. Note that in this case we can make the encoder RNN a bidirectional RNN, since we can assume we have access to the entire sequence as we encode it.



For the first model I'm discussing today, we're actually going to be autoencoding music. Music in this case is just represented as a sequence of one-hot vectors, which signify the current note being played at a given timestep, in addition to special events corresponding to "hold the last played note" or a rest. So, in this setup, we can only represent melodies, baselines, or drums - no chords. We'll be considering autoencoding 16-bar snippets of music, with each sequence element corresponding to a sixteenth note. Otherwise, the setup is pretty similar to any sequential autoencoder.

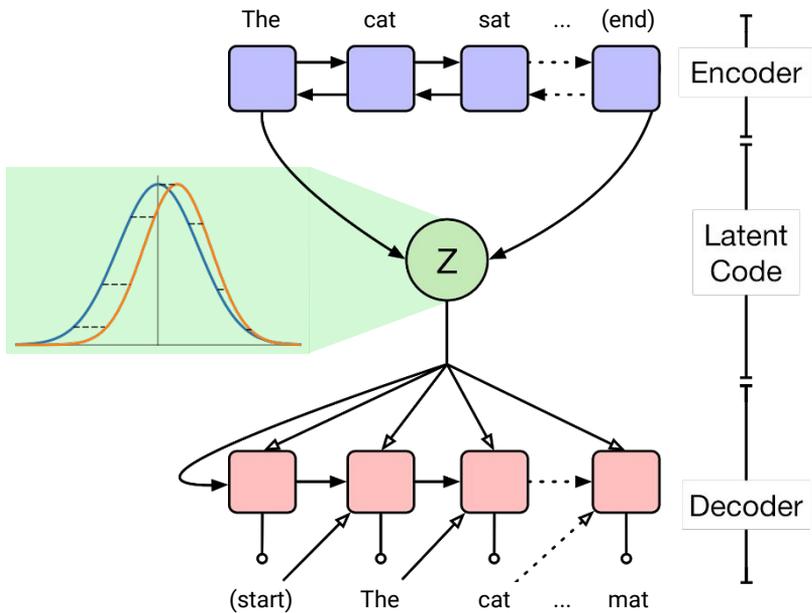


As described earlier, it can be useful to enforce specific structure on the latent code. We can do this using the variational autoencoder framework. As before, we'll treat the encoder as parametrizing a posterior distribution over latent codes, and introduce an additional loss term which measures the divergence between the posterior and a pre-determined prior. In doing so, we might hope to be able to do use latent-space manipulations like interpolations.



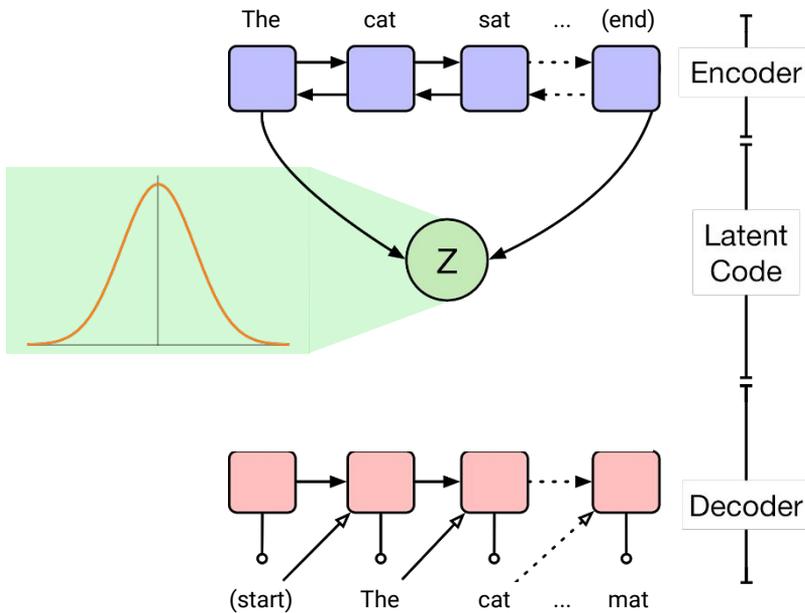
$$p(x_n | x_{n-1}, \dots, x_1)$$

Note that we will still be training the recurrent "decoder" as we would in the standard autoregressive maximum-likelihood formulation: We estimate the distribution over the possible values of the n 'th token based on the identities of all of the previous tokens in the sequence. When training this type of model, we feed in the correct input at each timestep, so the model's goal is to predict the distribution over possible values given the correct input sequence. This approach is called teacher forcing. This is a very powerful density estimation method - it's used in standard text language models, as well as powerful models of images like PixelCNN and audio like WaveNet.



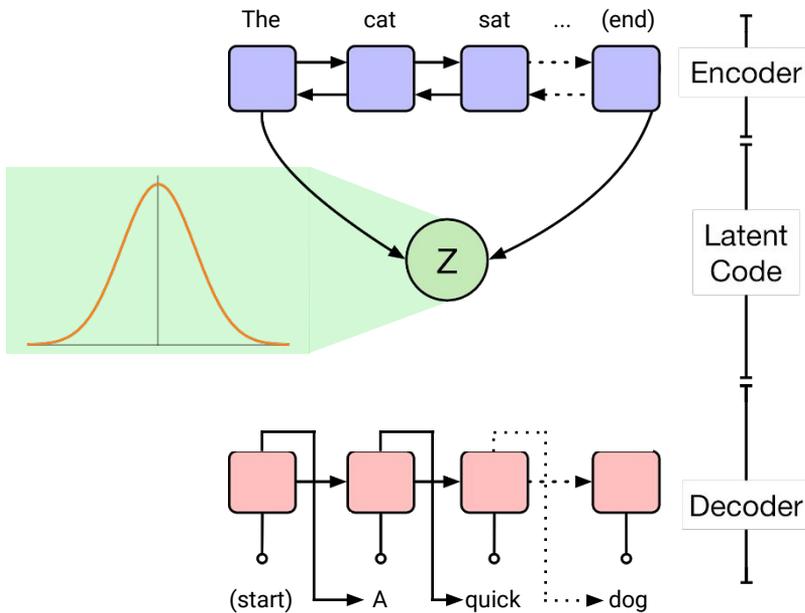
$$\mathbb{E} [\log p_{\theta}(x_n | x_{n-1}, \dots, x_1, z)] - \text{KL} [q_{\lambda}(z|x) || p(z)]$$

So, when we use this autoregressive density estimation technique in combination with the variational autoencoder, we get the lower bound on the log-likelihood of the data that I'm showing at the bottom here. The first term tries to maximize the log-probability of each token in the input sequence, conditioned on previous tokens and the latent code. The second term measures the divergence between the posterior distribution produced by the encoder and the prior distribution. Note that the divergence is minimized when it's zero and the posterior and prior are equivalent.



$$\mathbb{E} [\log p_{\theta}(x_n | x_{n-1}, \dots, x_1, z)] - \text{KL} [q_{\lambda}(z|x) || p(z)]$$

Unfortunately, what tends to happen in this kind of model is that the solution found is one where the second term goes to zero. This means that the posterior distribution predicted by the encoder is always the same, and it's always equal to the prior. In this case, the latent code can't contain any information about the input, because it's always the same distribution regardless of the input. People call this situation "posterior collapse" because the posterior has collapsed to the prior. One way of looking at why this happens is because the decoder is too powerful - it is sufficiently powerful on its own to model the data distribution, so the solution found is one where it can ignore the latent code and just model the data using the decoder alone. Alternatively, this suggests that the cost of incurring an additional penalty from encoding information in the latent code outweighs just not using the latent code at all.



$$\mathbb{E} [\log p_{\theta}(x_n | x_{n-1}, \dots, x_1, z)] - \text{KL} [q_{\lambda}(z|x) || p(z)]$$

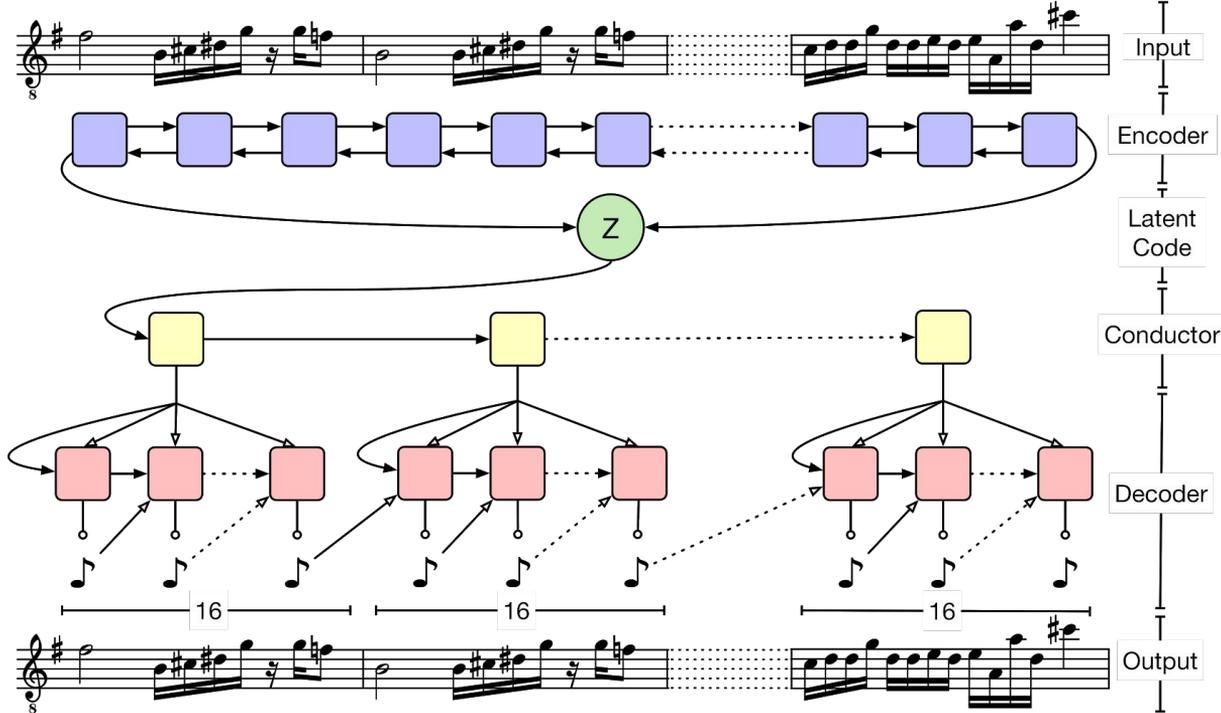
The problem with this scenario is that, as I mentioned, during training we feed the decoder the correct input (that is, the input as it was fed into the autoencoder) at each timestep. When we actually go to use the model to do things like sample new points and try to perform latent space manipulation, and we can't teacher force the input, the model can totally produce useful samples but they will be completely independent of the latent code. As a result, if we do latent space manipulations like interpolations the output will not change as intended. This is reflected in this diagram - since the decoder ignores the latent code, it ends up producing a totally plausible sample which is unrelated to the input when we don't teacher-force it.

$$\mathbb{E}[\log p_{\theta}(x|z)] - \text{KL}(q_{\lambda}(z|x)||p(z))$$

$$\mathbb{E}[\log p_{\theta}(x|z)] - \beta \text{KL}(q_{\lambda}(z|x)||p(z))$$

$$\mathbb{E}[\log p_{\theta}(x|z)] - \max(\text{KL}(q_{\lambda}(z|x)||p(z)) - \tau, 0)$$

The most common way to deal with this issue is to modify our loss function. The standard lower bound is shown at the top. If we add a scalar hyperparameter beta to scale the divergence term, we can make the cost of storing information in the latent code smaller by making beta small. Note that if you set beta to 0, you stop enforcing any structure on the latent code and we just get back to the standard autoencoder. Another way to avoid posterior collapse is shown on the bottom, where we basically don't enforce the divergence as long as it is less than tau. This gives the model an explicit "information budget" before we start penalizing it. The drawback of these techniques is that they each introduce an additional hyperparameter which can be hard to tune. We spent a lot of time trying out both approaches, and we weren't able to get the model to effectively model longer sequences of music.

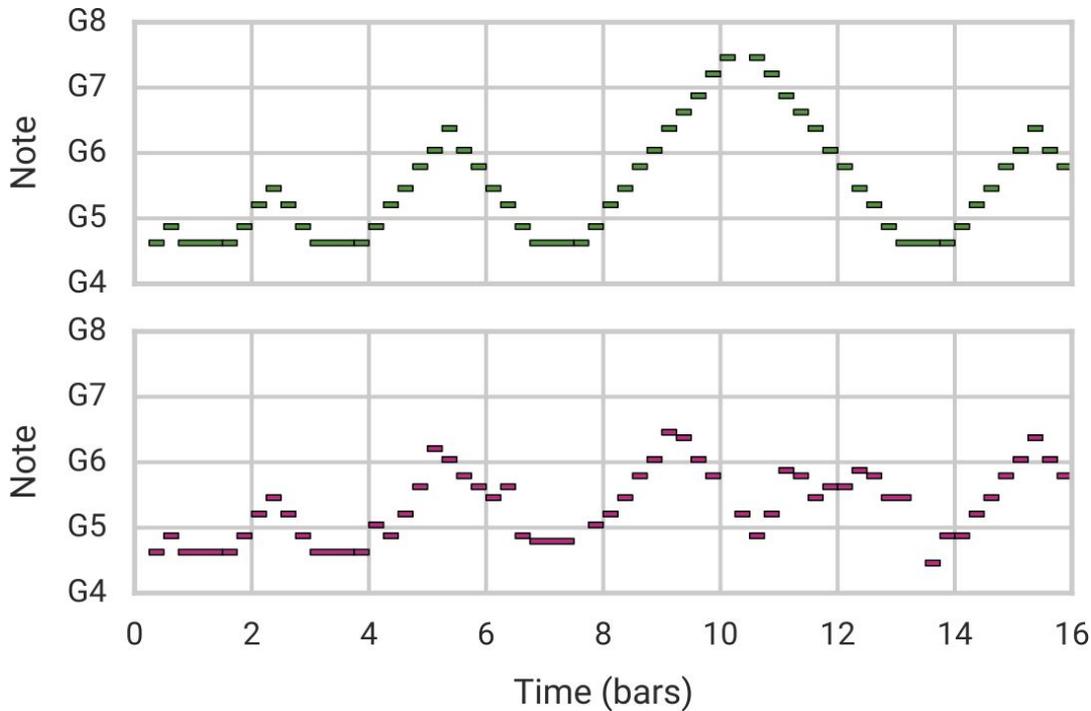


Another way to mitigate this issue is to change the model we are optimizing over. This approach is what allowed us to model long sequences of music, and produced the model that we call "MusicVAE", which is shown here. The main innovation is that we explicitly model the hierarchical structure of the data. Instead of a single RNN decoder, we first decode the latent code to a sequence of continuous latents using an RNN "conductor". Each of these states corresponds to one measure of music, or 16 output tokens. For each of these latents, we separately decode the corresponding portion of the output sequence. The output decoder does not share state across each chunk of the output, so the only way for it to gain context is to utilize the conductor's state. This gives it a stronger dependence on the latent code - if it did not use this state, it would have to attempt to predict each measure independently. Decomposing the output sequence in this way also provides a natural way of capturing its global structure in the latent code, as this global context is what the decoder will need when predicting each chunk of the output.

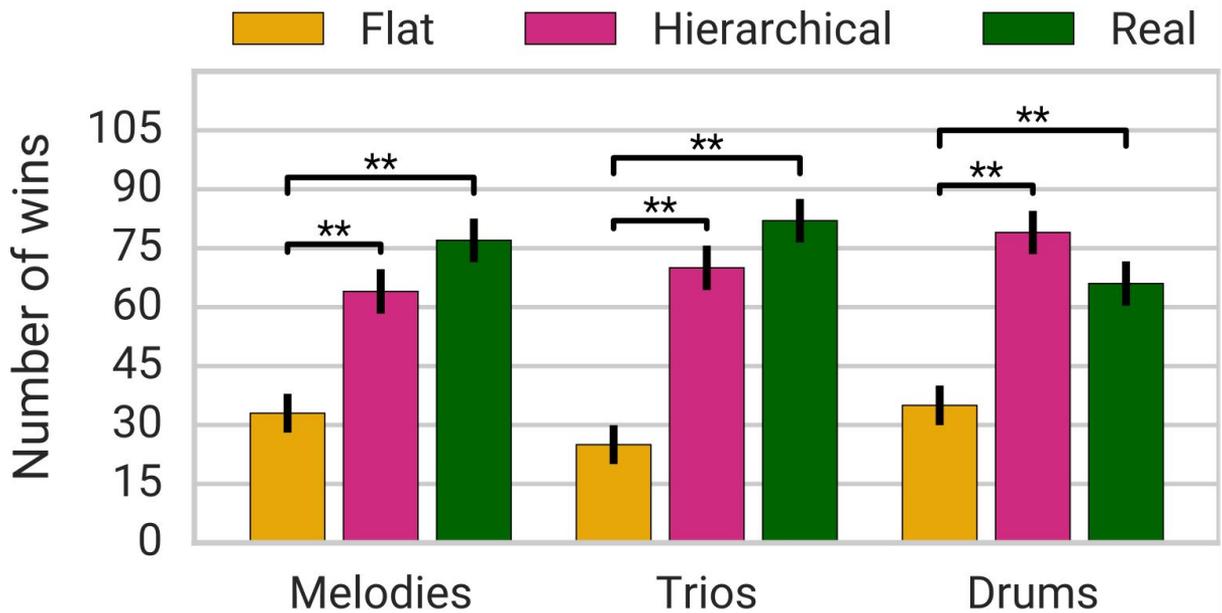
Reconstruction Accuracy

Model	Teacher-Forcing		Sampling	
	Flat	Hierarchical	Flat	Hierarchical
16-bar Melody	0.883	0.919	0.620	0.812
16-bar Drum	0.884	0.928	0.549	0.879
Trio (Melody)	0.796	0.848	0.579	0.753
Trio (Bass)	0.829	0.880	0.565	0.773
Trio (Drums)	0.903	0.912	0.641	0.863

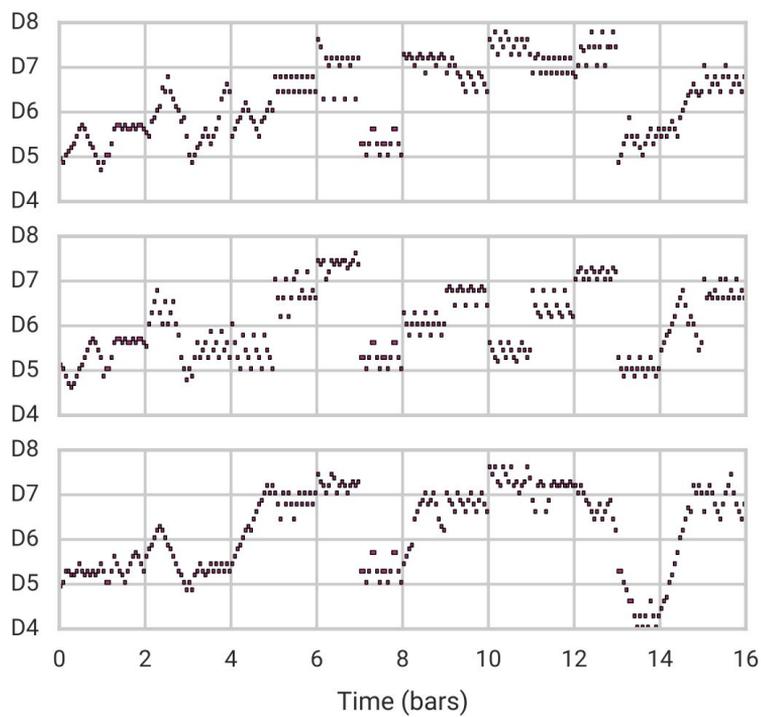
In practice, we found this procedure to be quite effective. If we measure the token-level reconstruction accuracy of our hierarchical MusicVAE against a non-hierarchical baseline, we found that it could more accurately reconstruct the input both when we teacher-force the input and when we don't. The latter column is really what we care about if we want to do latent-space manipulation on samples.



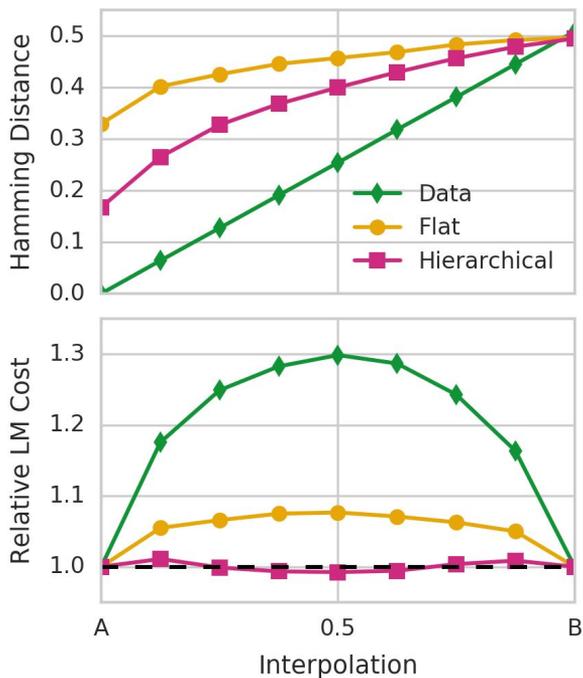
As you probably saw in the previous slide, our model isn't perfectly accurate. The interesting thing is that it tends to make "musical" mistakes - maintaining the overall structure and key of the piece, but changing some local characteristics. These piano rolls (explain?) show a music sequence on the top and its reconstruction on the bottom.



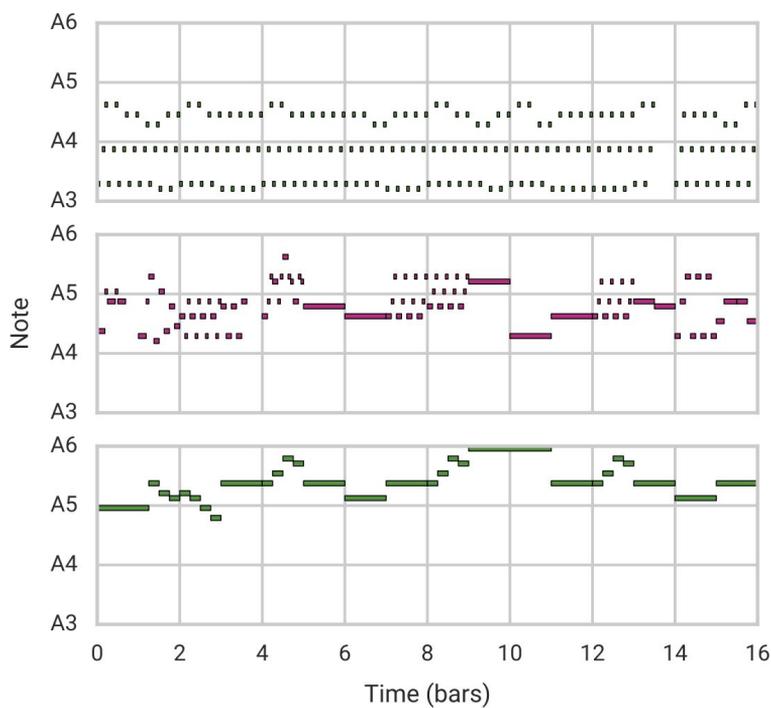
Note that reconstruction accuracy is not really enough to measure whether our model is useful. We could get perfect reconstruction just by using a standard autoencoder with no structure imposed on the latent code, that is ignoring the divergence term or setting $\beta=0$. It's also important to measure whether our samples are realistic. To measure this, we performed a listening study where we asked listeners which of two music clips were "more musical". Here we show the number of times each model was considered more musical - the flat baseline, the hierarchical model, or real data. In all cases the hierarchical model won much more than the flat model. For $N = 200$ per setting, the difference between the number of wins for the hierarchical model and real data was not significant under a Kruskal-Wallis H test.



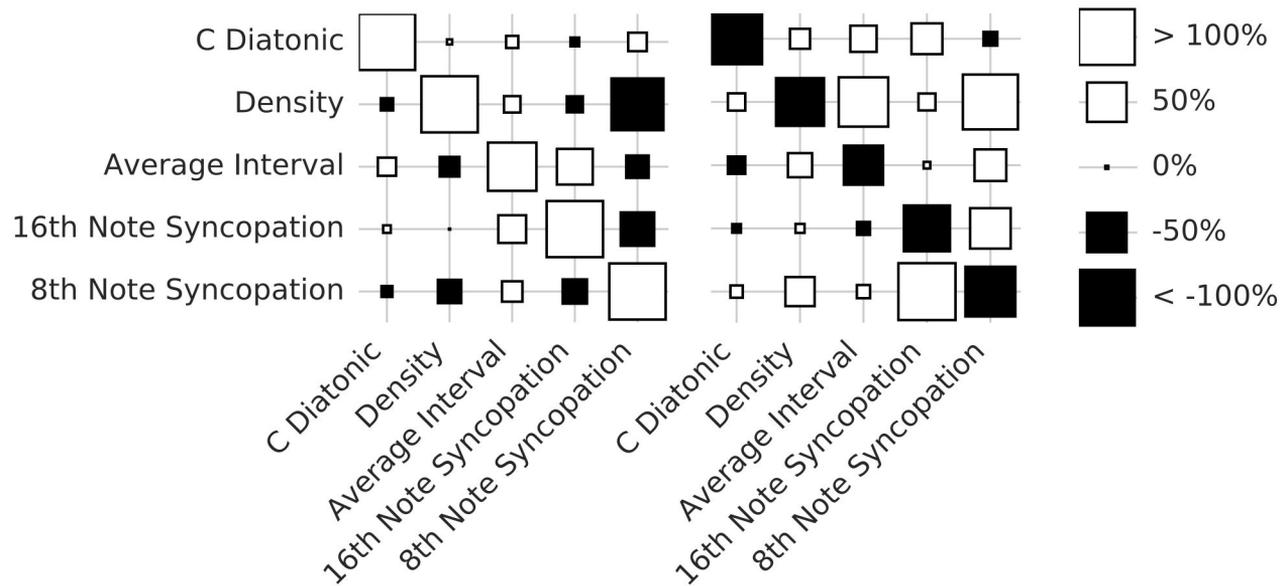
Now, remember that we could also have gotten good samples just by training an autoregressive model of the data, without any global latent code. One way to check that the model is using the latent code is to produce many samples from the model for the same fixed latent code. Here are three examples of that. Note that the overall structure and key is similar, but the exact realization of the music varies.



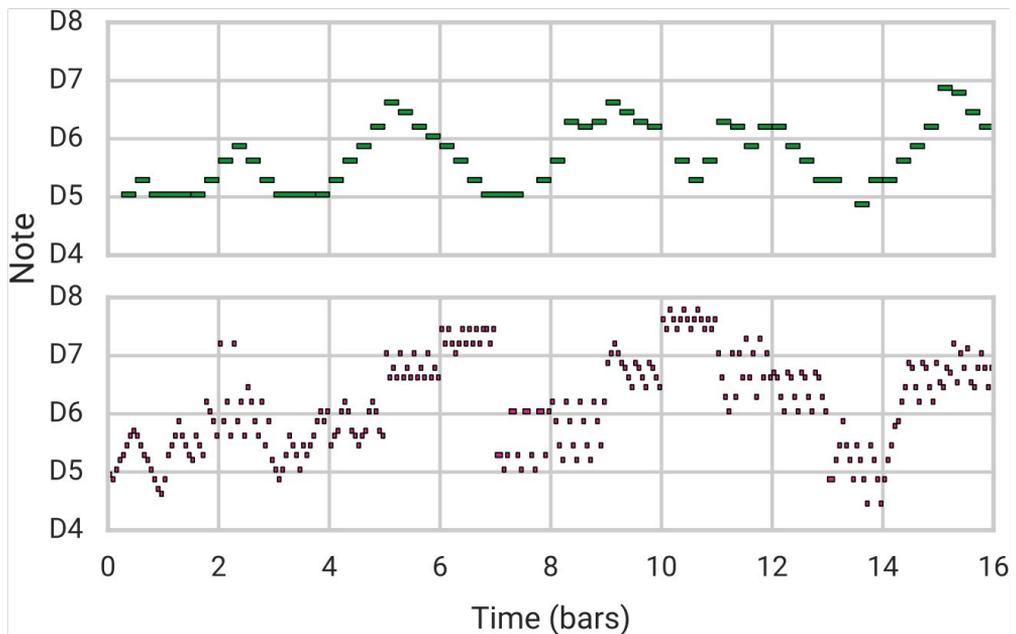
We can also check that the model is using its latent code by performing latent-space interpolation. To measure whether the model is interpolating successfully, we measured two things about the intermediate points along the interpolation. Say we are interpolating between two sequences, A and B. First, we measure the Hamming distance between each interpolated point and sequence A. We want this hamming distance to increase monotonically, so that as we interpolate towards B the sequence gets less similar to A. Note that we can do this perfectly by just sampling a proportion of notes from A or B directly instead of interpolating in latent space, which is what we're calling "data" in this diagram. The second thing we measure is the cost assigned to each interpolated data point by a simple 5-gram language model as we interpolate from A to B, normalized by the cost assigned by the language model to sequence A and B. We want this to stay near 1. So, on the top, you can see that the simple data baseline interpolation does what we want; both the flat and hierarchical models also smoothly morph from sequence A to B although the hamming distances for the "flat" model are higher because it reconstructs the input (A) imperfectly. On the bottom, you can see that our synthetic data interpolation has intermediate points which are assigned higher cost by the LM, suggesting they are less realistic. Only the hierarchical model generates interpolated points which are uniformly considered as realistic as the original datapoints. All of these curves were averaged over 1024 interpolations.



So, what does an interpolation sound like? In this figure we show on the top and bottom two simple melodies. The top is a quick arpeggio in a lower register in one key; the bottom is a slower melody in a higher register in another key. The interpolated point, shown in the middle, manages to successfully mix attributes (like the arpeggios and long notes) of both sequences in a middle register and still remain musical.



How else can we test that our model has discovered useful semantic attributes in the dataset? One behavior that VAEs have been shown to exhibit is the ability to perform attribute vector arithmetic. This operation involves collecting the latent codes corresponding to many datapoints which all share a particular attribute and averaging their corresponding latent representation. By adding or subtracting this average latent from the latent code corresponding to some datapoint and decoding the result, we can often increase or decrease the extent to which the attribute is present in the input. Note that this is done in a totally unsupervised way - we never explicitly tell the model during training what each attribute is, we only compute the attribute vectors after it has been trained. So, we defined about 5 attributes, corresponding to simple-to-compute characteristics, such as whether the sequence was in that C diatonic scale, how "dense" or frequent notes were, the average change in pitch or interval between notes, and so on. We found that when we added (left) or subtracted (right) these attributes from existing sequences, the corresponding attribute increased or decreased respectively. These attributes were also modeled relatively independent, however possible; however note that for example if we increase or decrease 8th or 16th note syncopation the other kind of syncopation will respectively decrease or increase, due to how the attributes are defined.



Here's an example of adding the note density attribute to a simple melody shown on the top. Note that a trivial way to increase note density is just to turn, for example, quarter notes into four repeated sixteenth notes; our model instead arpeggiates notes in a musically meaningful way, staying in key and so on.

Take A Step With Me.

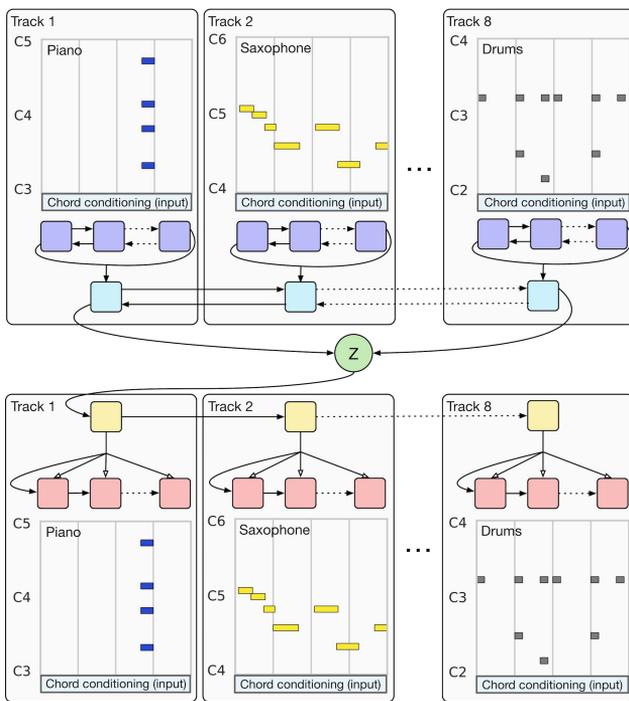
Words by
HARRY B. SMITH.Music by
JEROME D. KERN.

Allegro moderato.

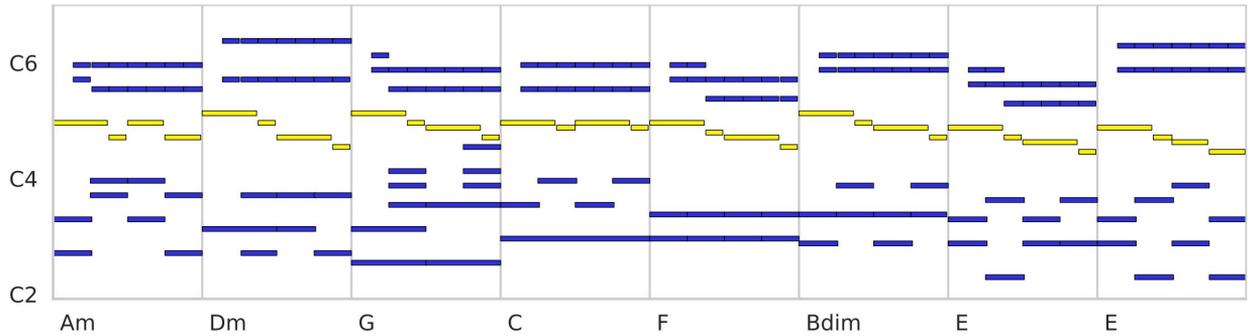
Piano.

The musical score is presented in three systems. The first system shows the piano introduction with a treble and bass clef, a key signature of one sharp (F#), and a 2/4 time signature. The tempo is marked 'Allegro moderato.' and the dynamics range from 'f' (forte) to 'p' (piano). The second system begins the vocal entry with the lyrics 'You A' and features a piano accompaniment with a 'p' dynamic. The third system continues the vocal line with the lyrics 'see her ev-ry-where, Pe-tite and de-bo-cer-tain mar-ried dame, I will not give her' and includes a piano accompaniment with a 'p' dynamic.

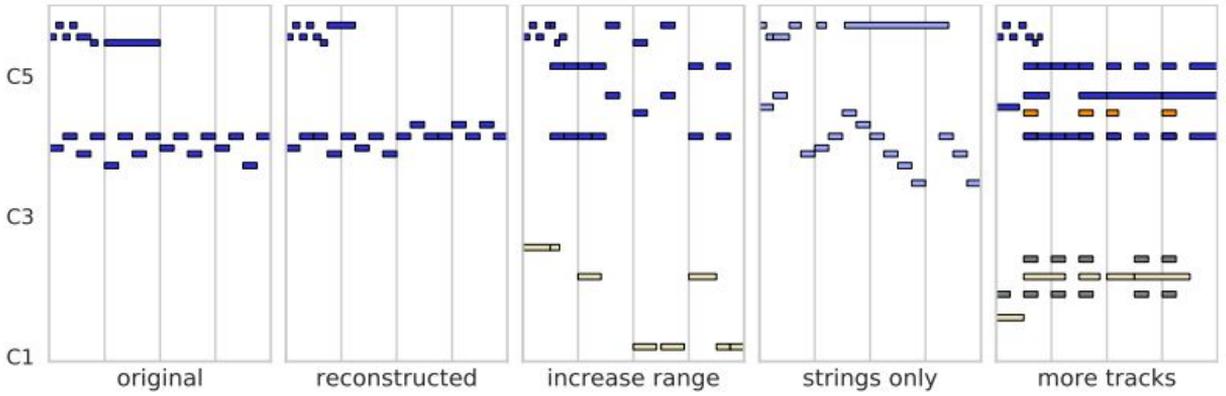
So, hopefully I have convinced you that MusicVAE can successfully model the global structure of music in its latent code and facilitate useful semantically meaningful manipulations. The main drawback of the model is really the way we've represented data - we're requiring that only one note is active at a time, that it only models one instrument at a time, and that all notes happen on sixteenth note intervals. Real music isn't like that.



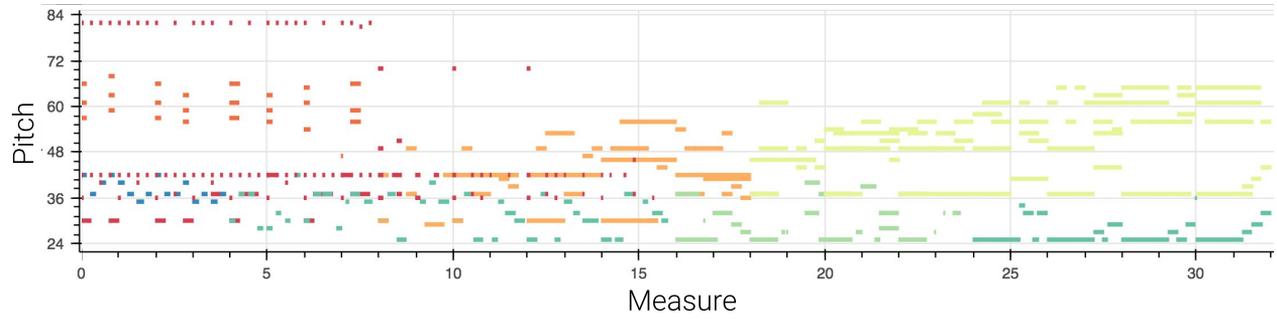
To address this, we extended MusicVAE with a new input representation and different architecture. Here, we model one-bar snippets of music with up to eight instruments, or tracks. Each track is encoded separately by a token-level encoder, and the result is fed to a "composer" RNN which produces a single encoding of all of the tracks by reading their encodings sequentially. The same process is repeated for the decoder - a composer produces embeddings for each track, and then a track-level encoder independently produces the notes for each track. We encode the instrument identity as a separate token, and separately model "note on", "note off", and "progress time" events. This allows chords to be played. Finally, we also optionally pass in a chord conditioning token as an auxiliary input, which allows us to control the chord being played in the 1-bar snippet. Note that the bar can have any number from 1 to eight tracks playing, because we can model an empty track by just not producing any notes for the track.



So how does this version of the model work? Since we are explicitly passing in the chord to play around, we can take a single latent code and decode it conditioned on different chords. This provides a natural way to produce a little snippet of music with a given chord progression. Note that we are only modeling short, one-bar snippets here, but by allowing us to control the chord progression we can naturally create music which appears to have long-term structure.



We can also perform attribute vector arithmetic here. Since our data representation is more expressive, we can model new attributes, like "strings only" meaning all tracks are playing on string instruments and "number of tracks" corresponding to how many tracks are active.



We can also still perform interpolations, except now we can interpolate between real music - like, real songs!



Radford et al., "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks"

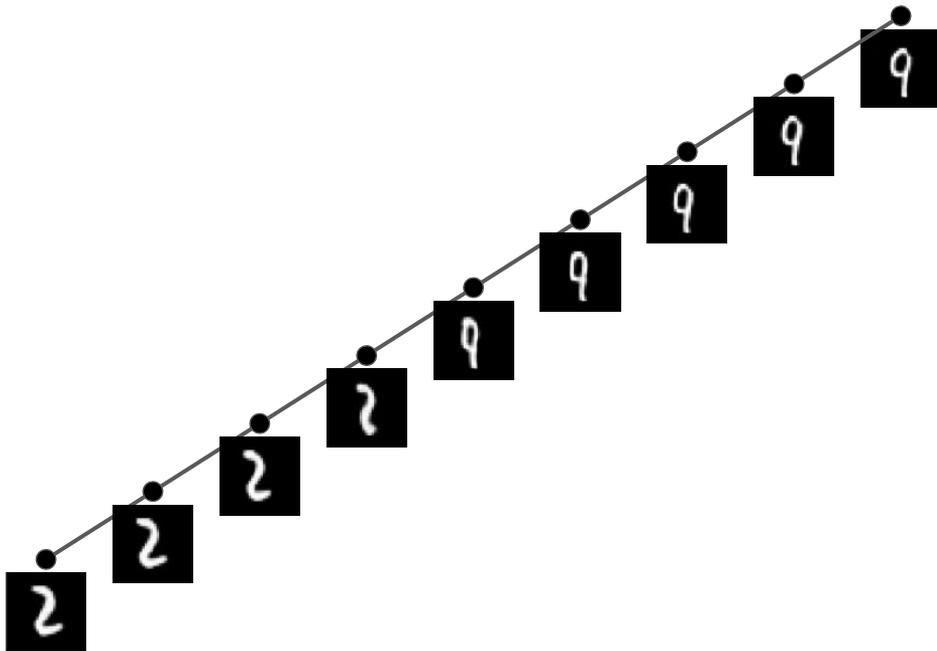


Ha & Eck, "A Neural Representation of Sketch Drawings"

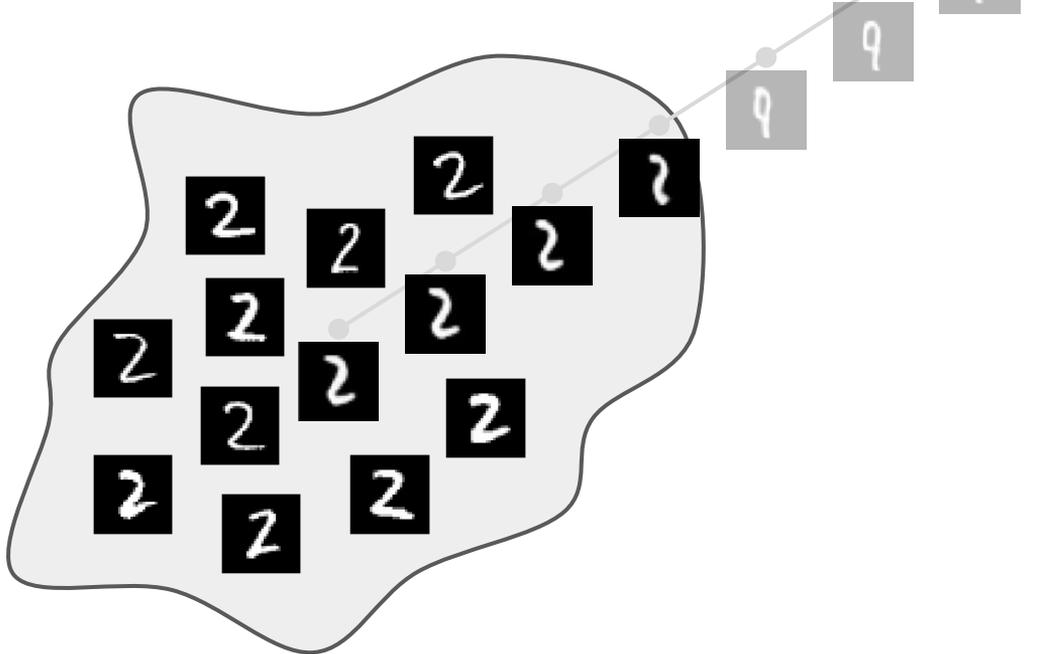


Roberts et al., "A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music"

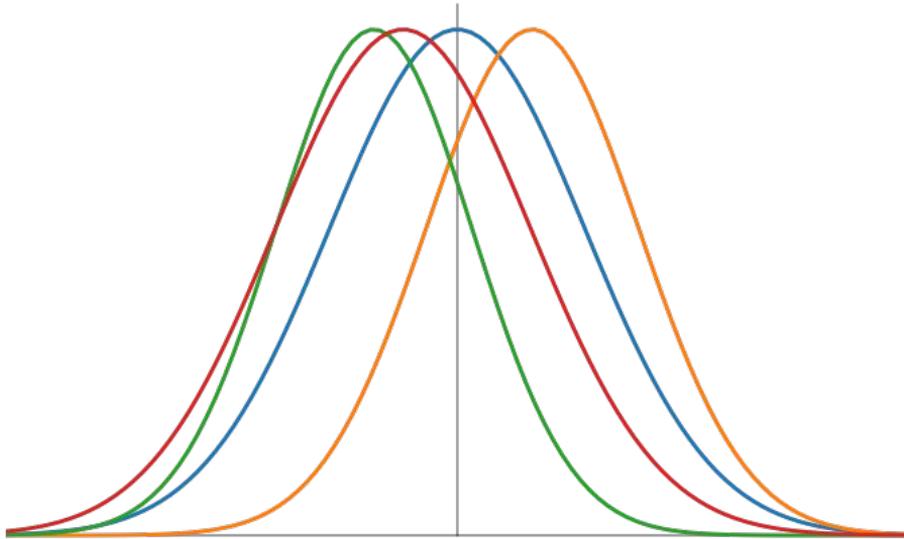
Ok, so one of the things I've been stressing so far is the ability for autoencoders to interpolate. This is in fact a pretty common thing that people discuss when talking about autoencoders, or latent-variable generative models in general. Here we see interpolations generated by the DCGAN, sketch-RNN, and our MusicVAE model. It can, as I've shown, be useful for creative applications. But now, I'll turn to a broader question: Does the ability to interpolate suggest that the autoencoder has learned a useful representation of this data?



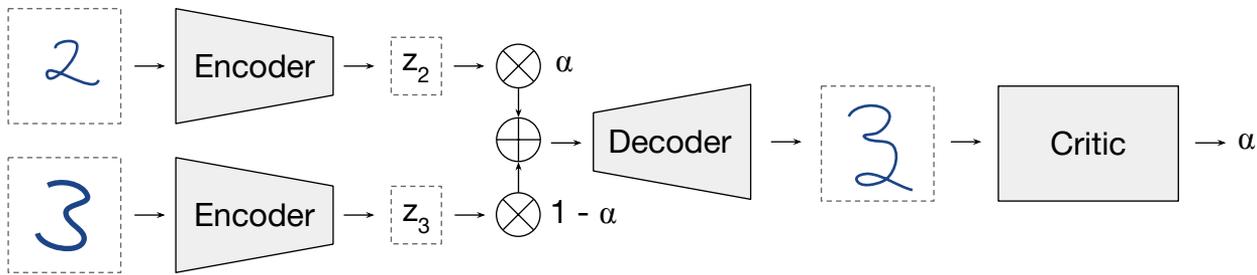
When we interpolate between two points in latent space, we're traversing the line in latent space between the encoding of point A and point B and decoding the result along the line. We say an interpolation is good if it semantically smoothly varies from point A to point B, and if it remains realistic across the interpolation. The first point requires that the latent space is "smooth", the second requires that it has no "holes" between data points where data becomes unrealistic.



So what does it mean that the latent space is smooth? It means that nearby each latent code are the latent codes corresponding to semantically similar datapoints. By extension this suggests some structure in the latent space - namely that the latent space is in some loose way organized around semantic similarity. So in this example, all of the similar twos are clustered near each other, because if we move a small amount away from one of the twos we should get a two-like symbol.



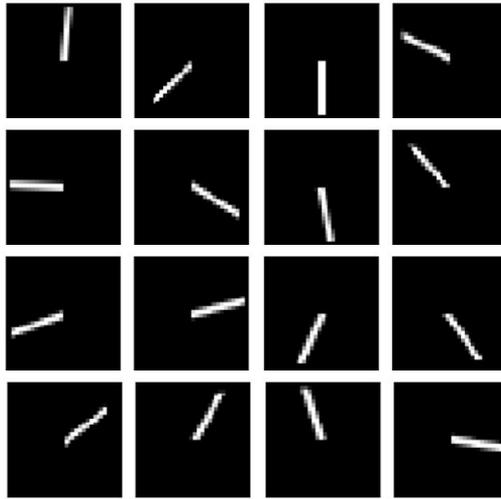
Why might interpolation appear in VAEs? Well, typically in a VAE the prior and posterior are modeled as diagonal-covariance Gaussians, with the prior being the standard spherical Gaussian with zero mean and identity covariance. In this setting, the KL term is effectively encouraging each dimension of the posterior to be concentrated around the origin, but with a nonzero variance. Naturally this will result in the distribution of latent codes for different datapoints to overlap. This means the model can't be certain whether a given encoding will be mapped to its intended output, or the output for another datapoint. The best way to mitigate this is to put similar points close together, so that if the model samples a latent code corresponding to the "wrong" data point it won't incur a high reconstruction cost. In other words, one possible explanation for why we can interpolate with a VAE is that it's an unintentional side-effect of the particular posterior/prior we chose.



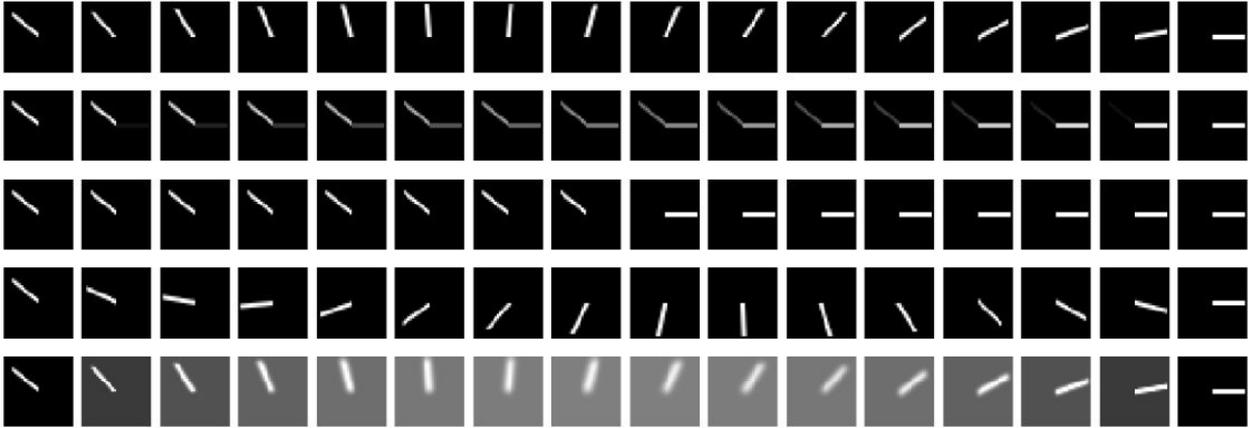
Adversarial Constraint for Autoencoder Interpolation



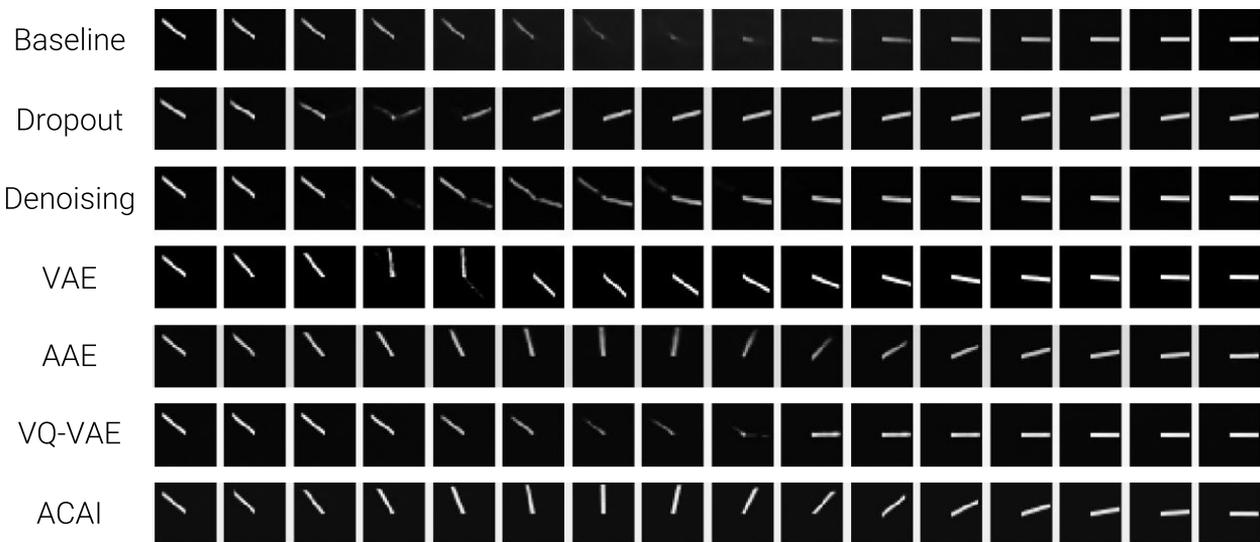
But now I'm going to talk about a more explicit way of encouraging good interpolations. This will allow us to test, in isolation, whether encouraging high-quality interpolations also produces useful representations. What is our goal when we interpolate? As I mentioned, one goal is that intermediate points are realistic, or in other words, are indistinguishable from "real" datapoints. We propose a regularizer for autoencoders which explicitly enforces this goal. We take two datapoints, encode them, interpolate their latent codes with some mixing coefficient α , then decode the result. Then, we train a critic to try to distinguish between reconstructions of real datapoints, and reconstructions of interpolated latent codes. It does this by trying to predict the mixing coefficient α . The autoencoder, in turn, is trained to force the critic to output $\alpha = 0$ for interpolated points, which would correspond to not mixing at all - in other words, producing real datapoints. The critic implicitly learns to distinguish between the distribution of interpolated reconstructions and real reconstructions, which gives the autoencoder a useful objective to use to make its interpolations more realistic. We call this approach "adversarial constraint for autoencoder interpolation", or ACAI.



To test this idea, we'll start with a toy task where we are autoencoding black-and-white images of lines. The lines are radii of a circle inscribed in the border of the image. In this toy setting, we know the underlying manifold of the data - it's one dimensional, corresponding to the angle of the line.

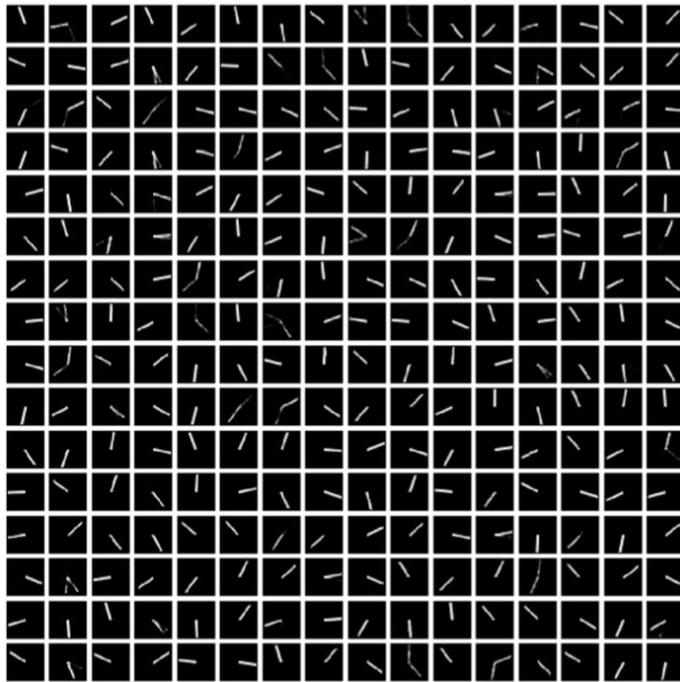


Here are some made-up examples of what good and bad interpolation on this dataset could look like. In this simple toy setting, we know what interpolating linearly on the data manifold should look like, and that's shown on the top. We are simply adjusting the angle of the line from the left endpoint to the right endpoint. The intermediate points look realistic. In the second row, we are interpolating in "data space", not along the data manifold. In the third row, we're abruptly moving along the data manifold, rather than smoothly. In the fourth row, we're interpolating smoothly but are not taking the shortest path along the manifold. In the final row, we are interpolating correctly but the intermediate points don't appear realistic.

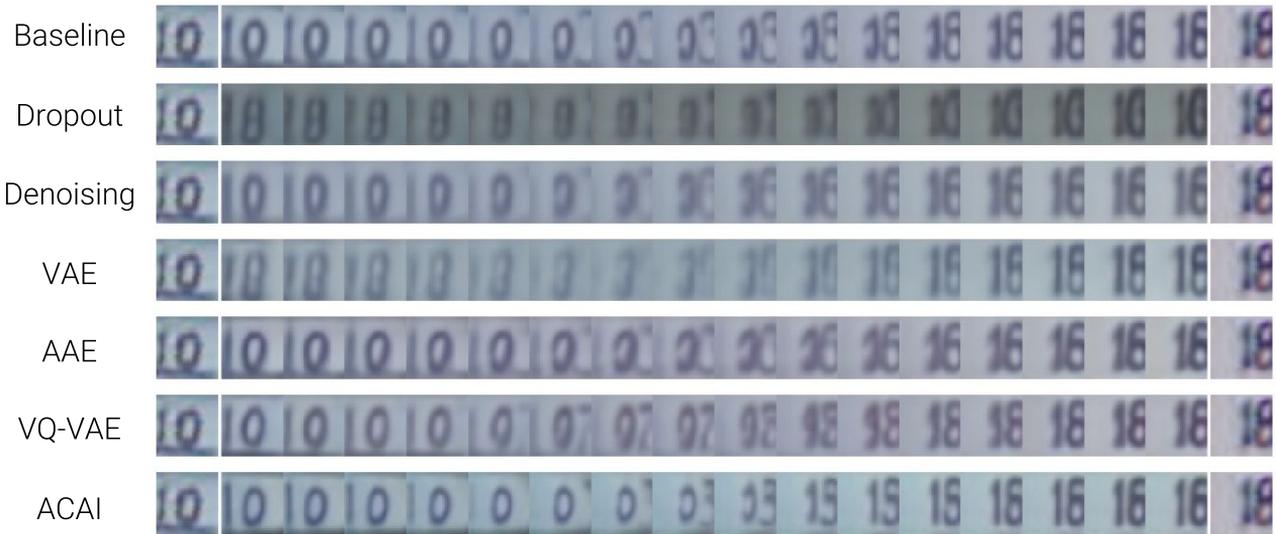


Metric	Baseline	Dropout	Denoising	VAE	AAE	VQ-VAE	ACAI
Mean Distance ($\times 10^{-3}$)	6.88 \pm 0.21	2.85 \pm 0.54	4.21 \pm 0.32	1.21 \pm 0.17	3.26 \pm 0.19	5.41 \pm 0.49	0.24\pm0.01
Smoothness	0.44 \pm 0.04	0.74 \pm 0.02	0.66 \pm 0.02	0.49 \pm 0.13	0.14 \pm 0.02	0.77 \pm 0.02	0.10\pm0.01

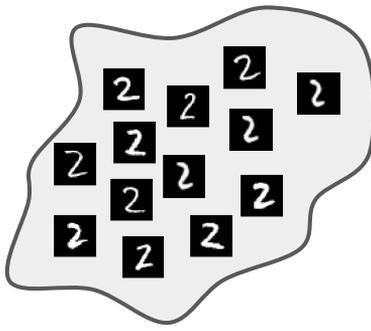
So, how do common autoencoders fare on this task? Here we show a whole slew of them, including our proposed regularizer. If we just train a normal autoencoder with no constraint on the latent code, the intermediate points stop looking realistic when we interpolate. If we apply dropout to the latent code, we see "abrupt" interpolation behavior. The denoising autoencoder does data-space interpolation. Surprisingly, the VAE exhibits abrupt interpolation behavior. The adversarial autoencoder seems to interpolate reasonably well but the intermediate lines stop looking as realistic. The Vector-quantized autoencoder stops appearing realistic, like the baseline. Finally, ACAI applied to the baseline, with no other constraints, interpolates exactly as we'd hope. We proposed some simple heuristic scores to measure interpolation quality and found that ACAI performed best.



So, back to the VAE. The surprising thing is that the VAE actually did learn the data distribution reasonably well. Here are 256 samples from the VAE, and you can see they all appear to be realistic lines. This suggests that having a VAE learn the distribution does not imply good interpolation - optimizing the VAE's loss function, the ELBO, optimizes a trade-off between the posterior fit and the reconstruction, and there's no reason to believe it will learn some particular representation.

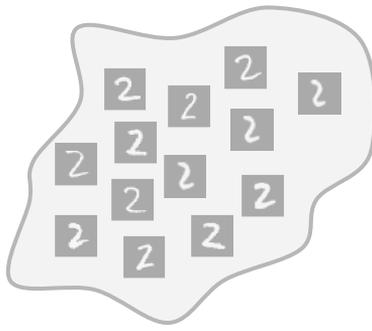


ACAI also interpolates well on real data. Other autoencoders are a bit more blurry, or intermediate datapoints aren't realistic like on the baseline, but really they all do reasonably well.



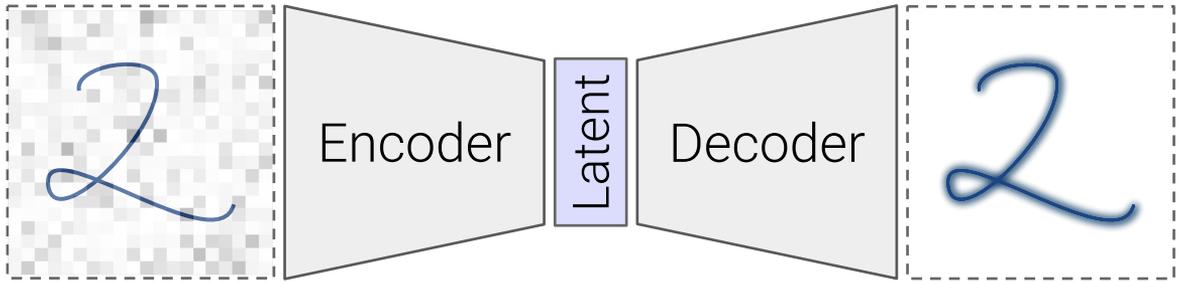
Dataset	d_z	Baseline	Dropout	Denosing	VAE	AAE	VQ-VAE	ACAI
MNIST	32	94.90±0.14	96.45±0.42	96.00±0.27	96.56±0.31	70.74±3.27	97.50±0.18	98.25±0.11
	256	93.94±0.13	94.50±0.29	98.51±0.04	98.74±0.14	90.03±0.54	97.25±1.42	99.00±0.08
SVHN	32	26.21±0.42	26.09±1.48	25.15±0.78	29.58±3.22	23.43±0.79	24.53±1.33	34.47±1.14
	256	22.74±0.05	25.12±1.05	77.89±0.35	66.30±1.06	22.81±0.24	44.94±20.42	85.14±0.20
CIFAR-10	256	47.92±0.20	40.99±0.41	53.78±0.36	47.49±0.22	40.65±1.45	42.80±0.44	52.77±0.45
	1024	51.62±0.25	49.38±0.77	60.65±0.14	51.39±0.46	42.86±0.88	16.22±12.44	63.99±0.47

So, remember that we motivated this idea as a test of whether improved interpolation also improves representation learning performance by mapping similar datapoints close together in latent space. One way we can test this is by training a single-layer classifier, a logistic regressor, on the latent codes learned by each autoencoder on various common datasets. Note that we do not optimize the autoencoder's weights with respect to the classification objective - we treat the weights as fixed, produce the latent codes, and only optimize the classifier's parameters to improve classification performance given this fixed feature representation. We found that the simple autoencoder combined with ACAI gave the best results in nearly every case, across three datasets and two latent dimensionalities. This suggests that indeed there may be some link between representation learning performance and interpolation ability, that interpolation suggests some useful structure in the latent space apart from potentially being useful for creative applications.

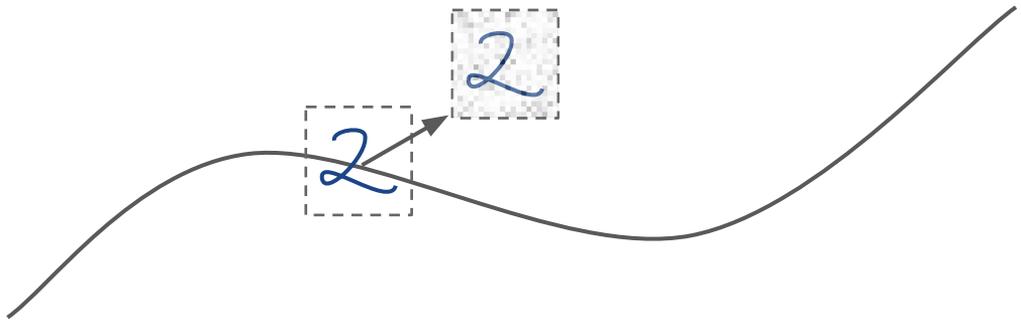


Dataset	d_z	Baseline	Dropout	Denoising	VAE	AAE	VQ-VAE	ACAI
MNIST	32	94.90±0.14	96.45±0.42	96.00±0.27	96.56±0.31	70.74±3.27	97.50±0.18	98.25±0.11
	256	93.94±0.13	94.50±0.29	98.51±0.04	98.74±0.14	90.03±0.54	97.25±1.42	99.00±0.08
SVHN	32	26.21±0.42	26.09±1.48	25.15±0.78	29.58±3.22	23.43±0.79	24.53±1.33	34.47±1.14
	256	22.74±0.05	25.12±1.05	77.89±0.35	66.30±1.06	22.81±0.24	44.94±20.42	85.14±0.20
CIFAR-10	256	47.92±0.20	40.99±0.41	53.78±0.36	47.49±0.22	40.65±1.45	42.80±0.44	52.77±0.45
	1024	51.62±0.25	49.38±0.77	60.65±0.14	51.39±0.46	42.86±0.88	16.22±12.44	63.99±0.47

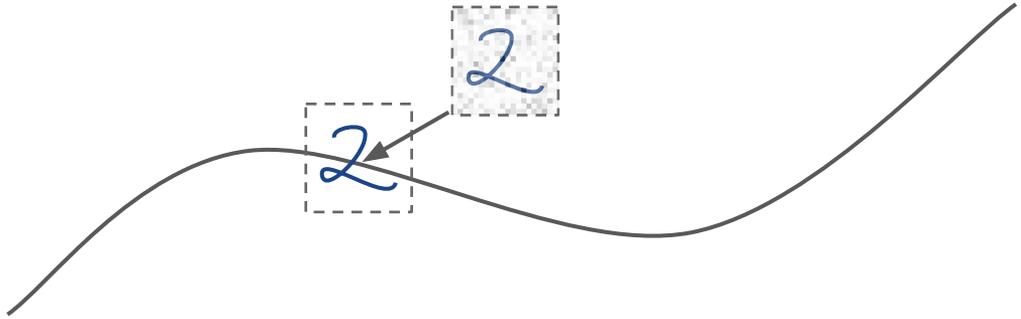
So one interesting thing about this results table is that the denoising autoencoder also fared pretty well. This was surprising to us - it's a sort of old-fashioned technique which has not gotten a lot of attention recently. It also is not enforcing good interpolation, or any other structure on the latent space.



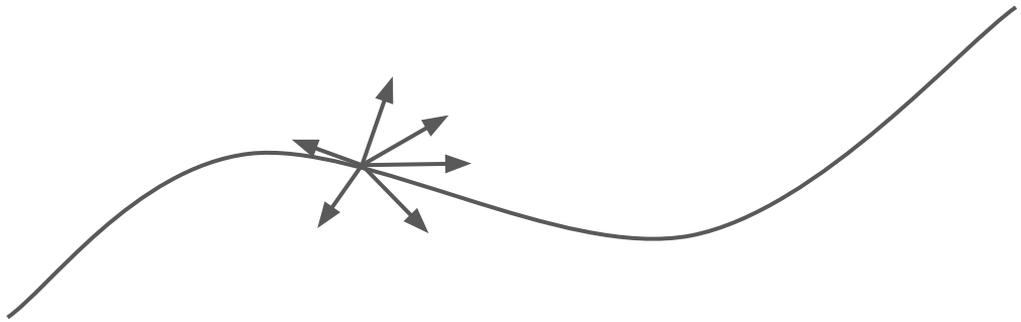
So what is it doing exactly? All it's doing is taking an input, adding noise to it (in this case Gaussian noise), and decoding the result. Its goal is to produce the original uncorrupted input given the corrupted version of it.



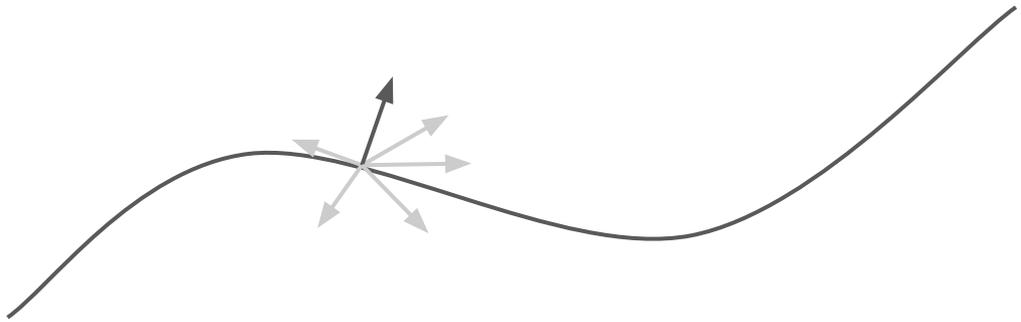
Why might this be a good way of doing representation learning? Well, imagine that our dataset falls on some underlying manifold, shown here as a squiggly line. Adding noise to the data moves the input away from the data manifold.



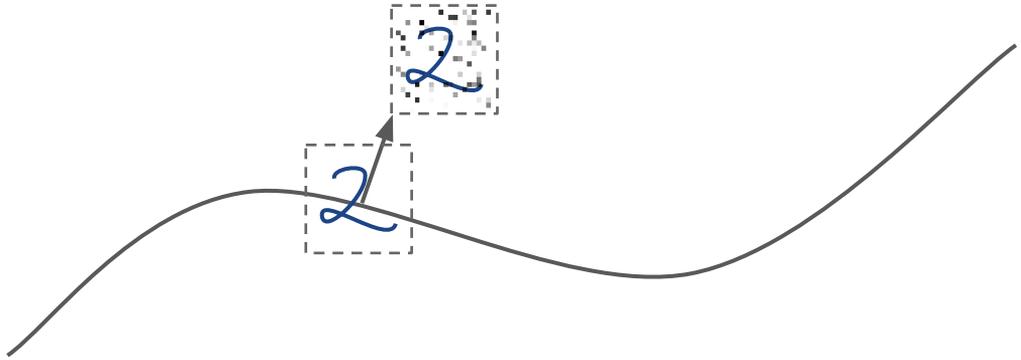
The denoising autoencoder's goal, then, is to move this corrupted input back onto the data manifold. In order to do so, you might imagine that it needs to internalize something about the data manifold.



So when we add noise to a given point on the data manifold, we are effectively moving the point around in a ball near a given true datapoint. There are many possible directions in high-dimensional space we could move, and we're just moving in one particular random direction.

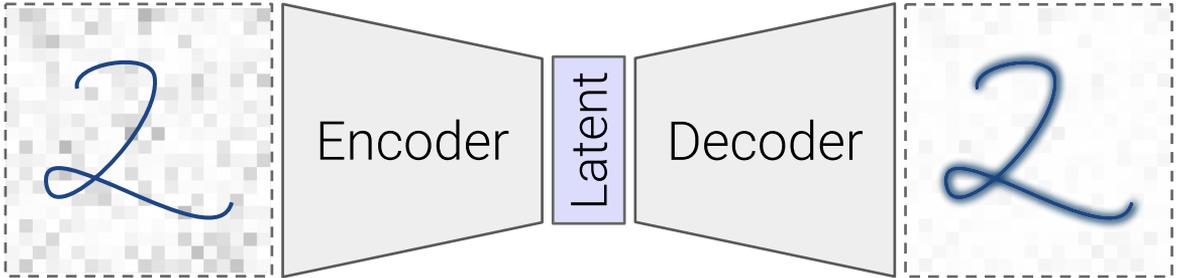


It may be that among these directions, the denoising autoencoder has already learned that most of them map away from the data manifold, but that it has not yet learned to be invariant to other directions. In other words, it may be more or less sensitive, in terms of its reconstruction error, to different perturbations, which suggests that in some directions it has not learned the shape of the data manifold correctly. It follows that it then may be more efficient to pick directions that the autoencoder is more sensitive to, instead of just randomly exploring the space around the data manifold.



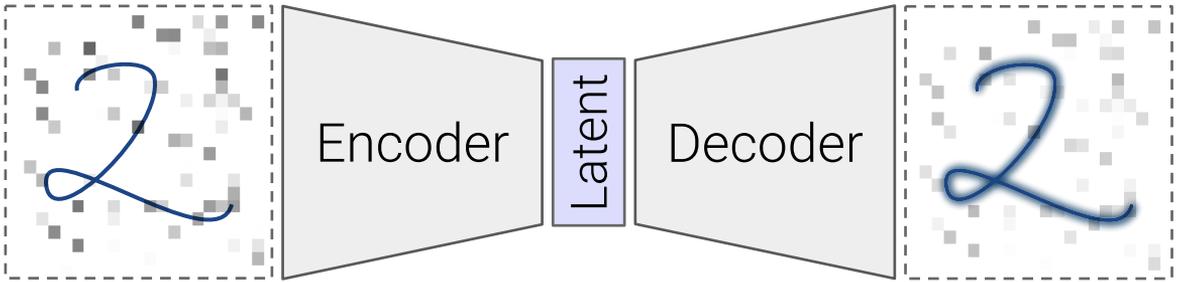
In this case, we would end up computing and applying a specific perturbation which is crafted specifically to maximize the reconstruction error, and then as usual train the denoising autoencoder to map back to the uncorrupted input.

$$r = \nabla_{\epsilon} \| \text{decoder}(\text{encoder}(x)) - \text{decoder}(\text{encoder}(x + \epsilon)) \|^2$$



How are we going to compute this perturbation? Well, what we are effectively doing is computing an "adversarial example" for the autoencoder. To compute this, we can add some noise to the input, feed it through the autoencoder, and then compute the error of the difference between the autoencoder output on the clean input and the perturbed input. We can then compute the gradient of the error with respect to the perturbation. This will give us the direction that maximally increases the reconstruction error.

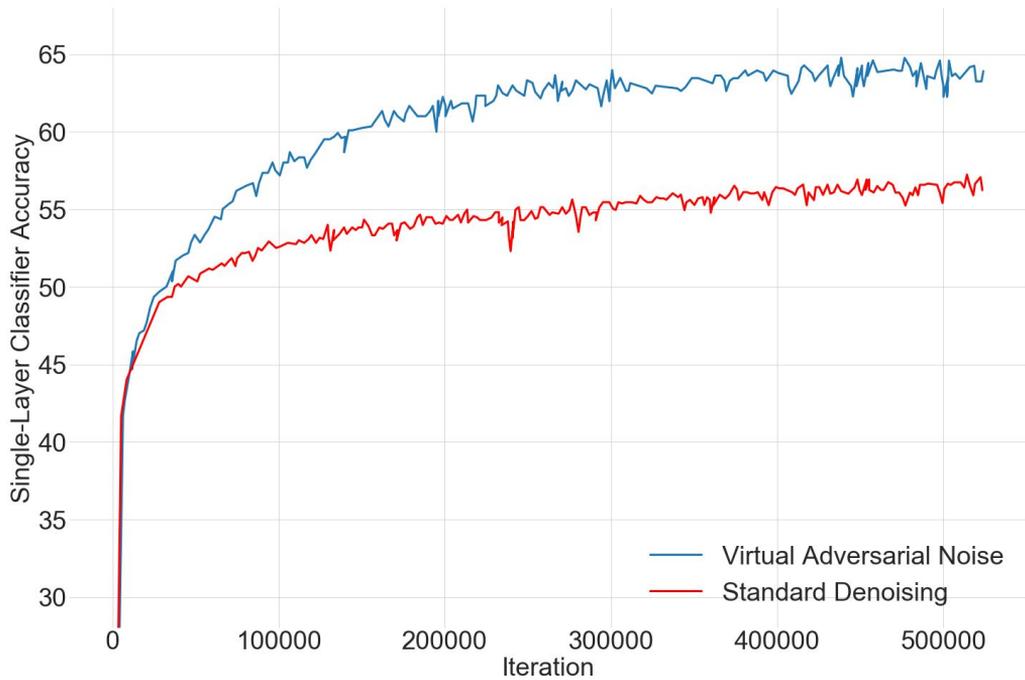
$$\|x - \text{decoder}(\text{encoder}(x + r))\|^2$$



Then, we can feed the perturbed input, using this adversarial perturbation, through the network. Our new loss function is to minimize the mean squared error of the network's output on the perturbed input against the original, uncorrupted input.



For example, for this dog, the adversarial direction changes the background color and jumbles the main object while also adding some random speckle noise.



So does this help? Some preliminary results suggest it might. Here's a graph showing the single-layer classifier accuracy, as we computed before, over the course of training. This is for CIFAR-10. In this case, using adversarial noise results in significantly better representation learning performance. This suggests it may be beneficial to use specially-crafted perturbations instead of random noise, but we need to do more experiments.

References

Roberts, Engel, Raffel, Hawthorne, and Eck, "*A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music.*"

Simon, Roberts, Raffel, Engel, Hawthorne, and Eck, "*Learning a Latent Space of Multitrack Measures*".

Berthelot*, Raffel*, Roy, and Goodfellow, "*Understanding and Improving Interpolation in Autoencoders via an Adversarial Regularizer.*"

Code

<https://github.com/tensorflow/magenta/>

<https://github.com/brain-research/acai/>

Thanks!

Here are some references for the papers I talked about today. All of the code for all of the papers is available online.