Sign Language Generation with Adversarial Styling

Varshith Sreeramdass

A report submitted in fulfillment of
the requirements for the Internship

at the

Honda Research Institute Japan Co., Ltd.
8-1 Honcho, Wako
Saitama Prefecture 351-0188

under the mentorship of

Heike Brock, Ph. D.

in the period between

14th May 2018
and
13th July 2018

CONTENTS

ABSTRACT

The work attempts to create and train a seq2seq model to generate motion sequences from sign annotations so that the generated motion sequences can be rigged onto a virtual avatar and made to sign the sentence.

To incorporate a human-like variability and naturalness in the generated sequences, the generation process is coupled with style parameters that are learnt in an unsupervised manner using a GAN approach.

The work however fails in generating reliable sequences usable for communication, but does explore a few representations for the motion capture data and invalidates using certain approaches.

INTRODUCTION

There have been very few attempts to translate spoken or written language into sign language because of the minority of the population that uses sign language and the lack in capital returns. However the problem is quite a challenging one, involving disciplines of linguistics, vision and signal processing.

Human Motion Sequences involve a large number of degrees of freedom. So many that it becomes impractical to be able to process all of them using statistical or hand engineered methods. Having something like a language built out of the motion sequence is even more nuanced with a large number of parameters to take into account.
Deep learning is aptly suited for the task of generating data of such complex structure.

Though a lot of headway has been made into generating images, expecially using GANs, (Goodfellow et al. (2014)), similar progress is yet to be made in generating motion sequences, like that of HPGAN, (Barsoum et al. (2017)) but on a larger scale.
An end to end synthesis system like that of Tacotron, (Wang et al. (2017)), suffers from lack of variation and tends to produce the mean of the training samples. This was tackled with the introduction of the idea of style tokens, (Wang et al. (2018)), and having the output correspond with these styles. That could be difficult to have if the data is not sorted according to the different styles. Moreover it may be difficult to directly observe interpretable styles like in the case of motion sequences for sign language gestures.
Using the ideas developed in the study of GANs, InfoGAN (Chen et al. (2016)), the idea of performing a semantic interpretation of the latent space

learnt by the adversarial process itself could be extended into this task. The latent space could have a semantic interpretation with the features corresponding to the style of signing, extent of overlap, hand slips and other such characteristics.

This work makes this proposal and attempts to setup a framework for performing this kind of a generative process in a data scarce setup. While the work is left in an intermediate stage, it establishes certain approaches that could be taken the way forward in representation of the data.
The particular approaches it explores for the representation of orientations are euler angles, quaternions and discrete classes of the angle space.
This work combines studies on human activity recognition, signal-data generation, hierarchical modelling and language translation.

PRELIMINARIES

Kindly refer to the cited papers for reference to the below mentioned preliminaries relevant to understanding the work done here.

Data

Brock and Nakadai (2018)

RNNs

Colah's Blog [1]

GANs

Goodfellow et al. (2014)
Mirza and Osindero (2014)

[1]http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Architectures

The overall architecture takes the form of a straight forward Conditional GAN setup. The difference lies in the conditional input here being not a one hot vector description of the condition but is of a variable length that is to be summarised using an encoder network.

The generator reads in a sequence of annotations which is input into a generator, along with a sample from the latent space. The generator outputs a produced sequence.

The produced sequence and the target sequence are input into the discriminator and are supervised with the real/fake labels.

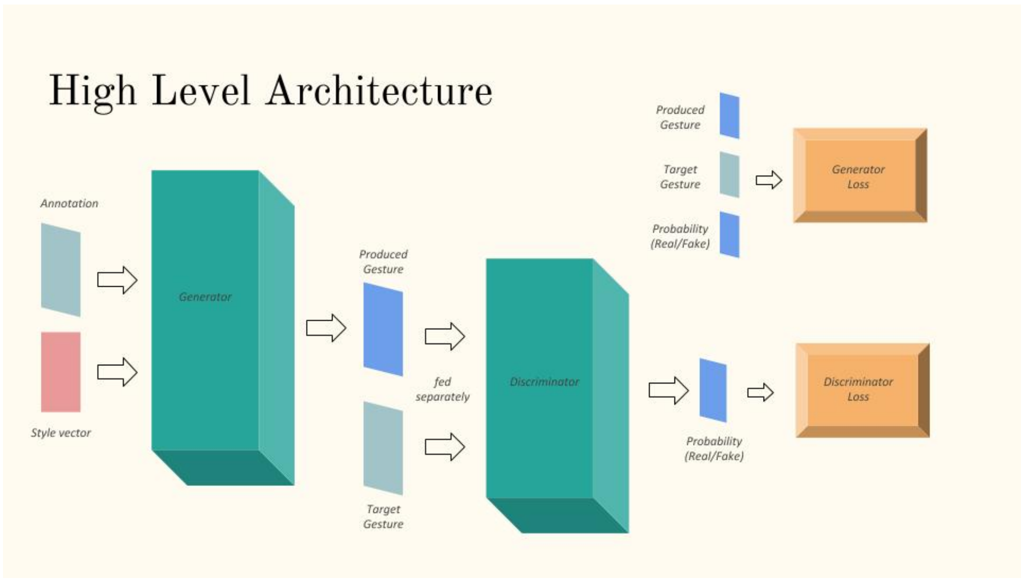

Figure 0.1: An eagle eye view of the architecture

The generator being a sequence to sequence model has an encoder-decoder architecture. The respective architectures are explained in the following sections. The structure of the discriminator has not been documented as it was implemented but not tested. In a nutshell, it is a convolutional network with filters among which each filter performs a temporal convolution over the angles of a joint over time.

Encoder

The encoder network involves the components described below.

- The annotation is projected into a learnable embedding space
- Sinusoidal position encodings as described in (Vaswani et al. (2017)) are added to the embed annotation. This is done to provide position information to the encoder RNN Network
- A residual convolution is applied over these embeddings so that the context information is accounted for. However this is optional as only the immediate context is essential considering the transitions that are to be captured involve only the adjacent annotations
- The output of the convolutions is processed by a bidirectional encoder and the encoder states are passed on to the decoder

Decoder

- The latent vector is tiled and added to each of the encoder outputs. This is to force the style space to act as a noise parameter that is additive
- The attention weights of the encoder outputs are determined by the state of the first layer of the RNN layer (to facilitate better parallelization) and output token in the previous timestep using a multihead

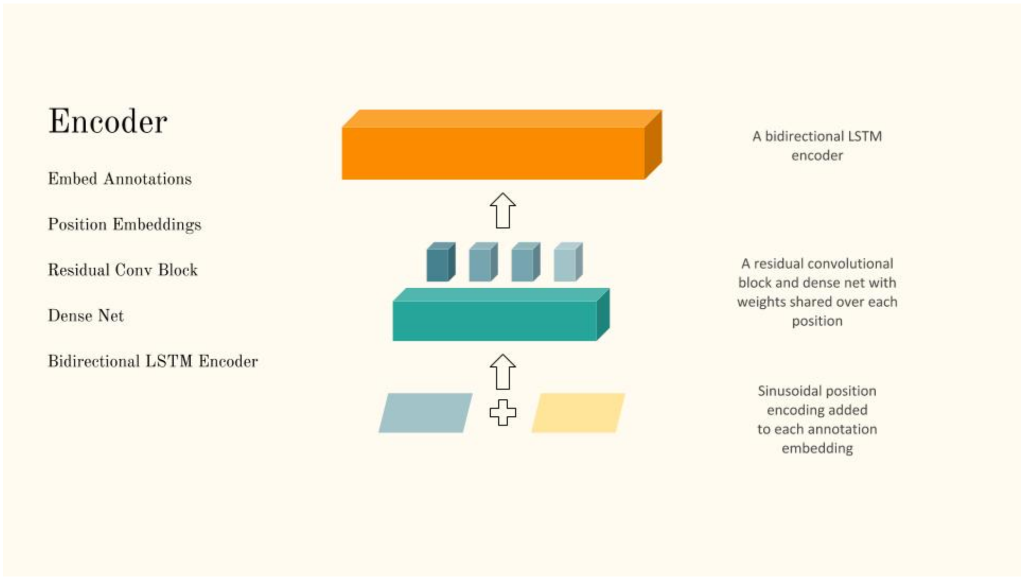
Encoder
Embed Annotations
Position Embeddings
Residual Conv Block
Dense Net
Bidirectional LSTM Encoder
A bidirectional LSTM encoder
A residual convolutional block and dense net with weights shared over each position
Sinusoidal position encoding added to each annotation embedding

Figure 0.2: Encoder architecture

attention network. The attention is multihead so that the transitions can be asynchronous across joints

- This weighted context vector is combined with the previous output and fed to the RNN Network
- The RNN network uses dropouts and residual connections to avoid exploding gradients
- The output of the RNN is passed through a three layer network, the probabilities for the angles are generated with a softmax layer and the angles for each of the joints are sampled

Data Representation

In a data rich scenario, even if the target output is constrained to be in a certain space, the model need not be restricted as such, as it can be trained

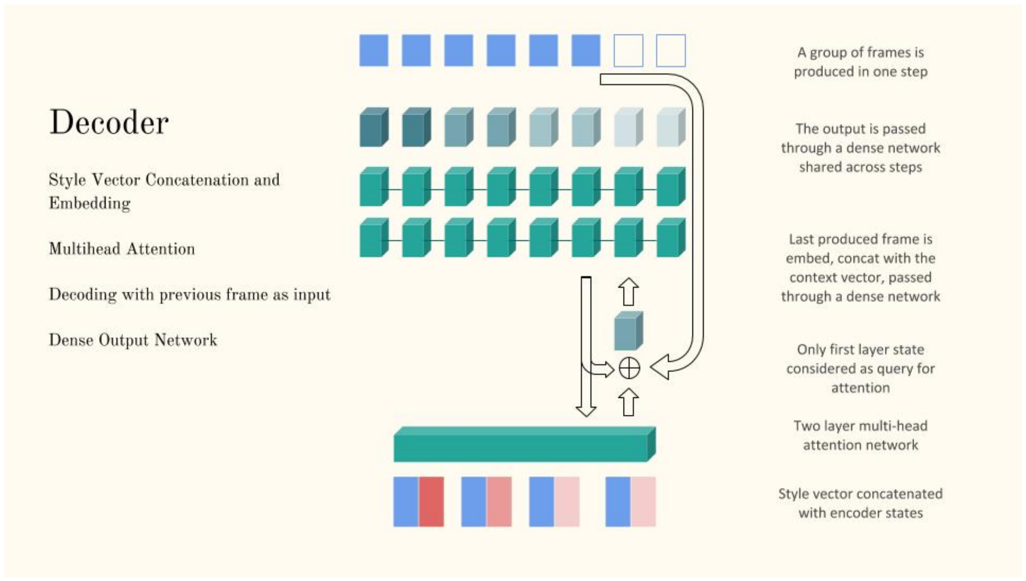
Decoder
Style Vector Concatenation and Embedding
Multihead Attention
Decoding with previous frame as input
Dense Output Network
A group of frames is produced in one step
The output is passed through a dense network shared across steps
Last produced frame is embed, concat with the context vector, passed through a dense network
Only first layer state considered as query for attention
Two layer multi-head attention network
Style vector concatenated with encoder states

Figure 0.3: Decoder architecture

to learn the constraints on the output.
But the same cannot be expected if the amount and quality of the data is poor. In these scenarios, the output of the model needs to be inherently constrained.

For this purpose, the choice of data representation also influences the activation functions used, the architecture itself and the training strategy.

Three representations are chosen and studied.

Regression with Euler Angles

The euler angle space is cyclic. The angles -180° and 180° are exactly the same with a dissimilarity of zero. This cannot be directly captured with a euclidean distance measure in the cost function.
The gimbal lock problem could potentially create flat cost spaces with non

informative gradients as well.

Regression with Unit Quaternion Half Spaces

Euler angles being a three dimensional cyclic representation of orientation can be converted to a constrained four dimensional representation called a quaternion.

For a four vector of real values to represent a representation, it has to have a unit norm. On top of this, quaternion spaces are redundant in that a quaternion represents the exact same orientation as its negative counterpart.

To tackle this, the space considered is a unit semi hypersphere in the half space where the first component is positive. The activation of the outputs of the network is designed to be of the form:

$$q(x, y, z, w) = \frac{(f(x), g(y), g(z), g(w))}{\sqrt{f(x)^2 + g(y)^2 + g(z)^2 + g(w)^2}}$$

f is chosen to be a function that outputs a positive value while g is chosen to be a function that is unconstrained as far as positivity is concerned.
The activation function is highly non differentiable. Initialization of the weights tends to be a difficult task as well.
The half space introduced disturbs the distance measure used for the cost function as opposite points on the diameter of the hypersphere on the boundary of the half plane are identical while measure says otherwise.
Better distance measures could be explored.

Classification with discretized Euler Angles

The space of 360° is divided into 120 classes with a resolution of 3°. The output activation is a softmax that outputs the probability of the angle

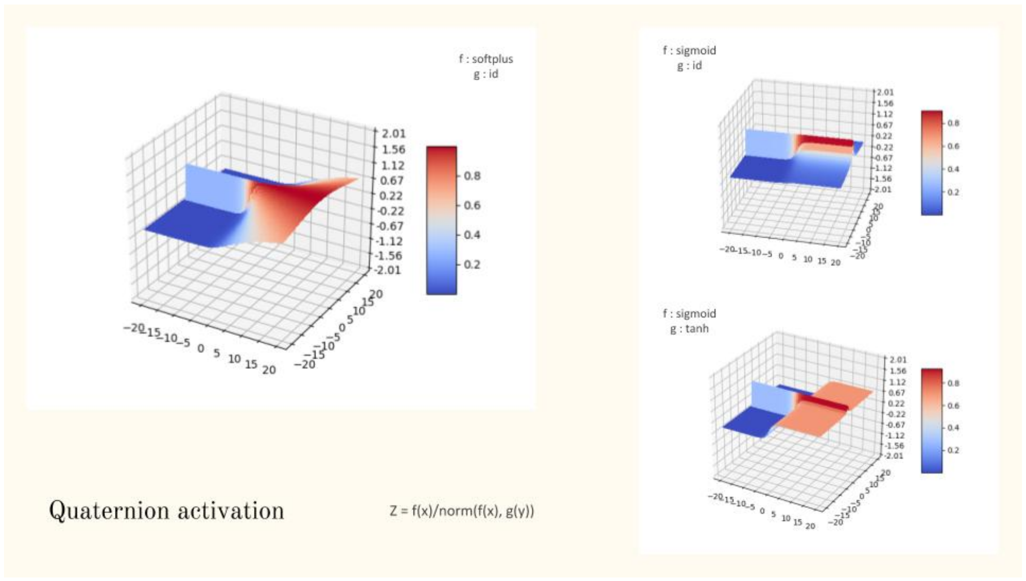

Figure 0.4: Quaternion Activation Spaces. The space shown is the x component of the activation of a two dimensional output

being a particular class.
To enforce some correlation among nearby classes, the target label is smoothed while considering cross entropy for the cost. This is only a workaround and better methods could be found.

Annotations and Paddings

An end-of-sentence token is added to the dictionary of annotations.
The sequence of gestures are edge-padded so that the batch contains sequences of the same length.
The expectation in doing this is that the model learns to output a constant gesture once it observes the <eos> token.

Training

The training process is adversarial. This involves a generator cost and a discriminator cost.

Costs

The generator cost involves the following;

- Smoothness cost so that that the output of the network is smooth and gradual over time. $sin^2\frac{\theta}{2}$ where θ is the difference between two consecutive angles
- The segment lengths are expected to be constant. This is captured by $(x - l)^2$ where x is the predicted length and l is the actual length. The experiments however consider the segment lengths to be fixed
- Distance between the target orientation and the predicted orientation. Euclidean distance in the regression cases and cross entropy with smoothened labels in case of classification
- Distance across time to be (soft/differentiable) Dynamic Time Warping distance (to be tried)
- WGAN loss for the generator

The discriminator cost is a standard WGAN loss involving a gradient penalty term as in (Gulrajani et al. (2017)).

The style vector is sampled from a normal distribution.

Sequence Masks

Even though the gestures are edge-padded, as an alternative approach, sequence masks are used. The differences are computed for each point

along the entire length of the padded sequence but the cost for the sequence only considers the differences up to its actual length and ignores the padding. This is done considering the nature of the dataset where the lengths of the sequences are highly variant and the majority of the lengths of the sequences is just padding.
The mask ensures that due importance is given to the original sequence over padding.

Due to complications in the pre-training phase of the generator (using the generator cost without the discriminator component), adversarial training process did not commence. The below observations are in reference to the pretraining of the generator.

The euler angles representation led to an extremely noisy output where the inherent constraints on the data were hardly learnt.
The quaternion representation, thanks to the complex non-differentiable space, failed to learn anything at all. Trivial initialization strategies tended to place the initial cost closer to the non differentiable regions. More complex designed initialization strategies could be explored.
The discretized cost with the cross entropy loss seemed to be the most stable among all, with the training behaviour not very dependent on the

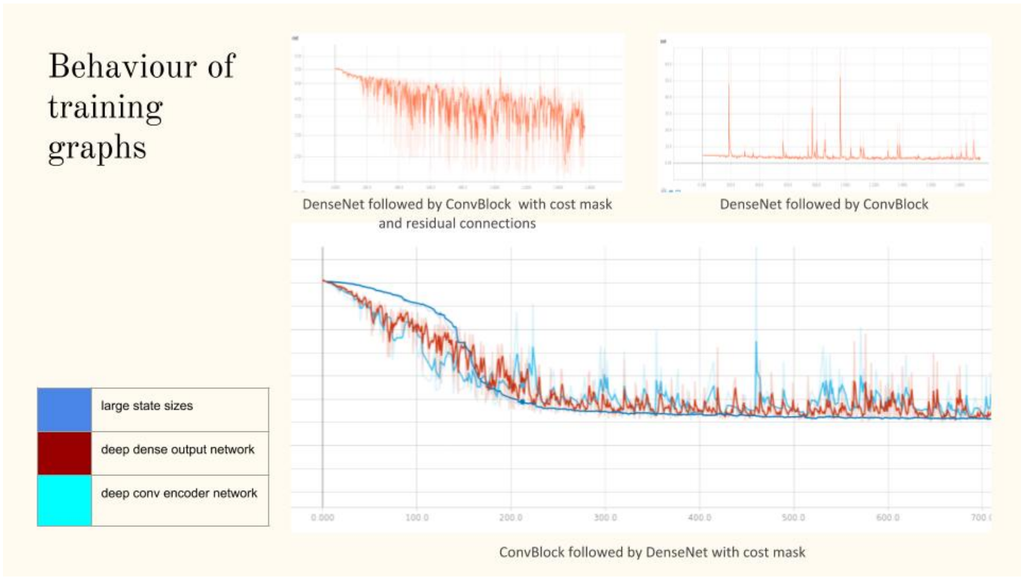

Figure 0.5: Training graphs of various configurations of the model

initial values of the weights involved.

The cost space and consequently the training process varied significantly with changing architectural elements.
The use of residual connections in the encoder and the decoder significantly reduced the problem of exploding gradients.
Two architectural variants that were tried were; convolutions performed on annotation embeddings followed by a fully connected network, a fully connected network on embeddings followed by convolutions. The training graphs are shown in figure 0.5.
While the second approach led to a very unstable training graph, the first approach was relatively stable. Convolutions are designed to exploit spatial correlations in the input. When the convolutions are performed on the output of a fully connected net, they work with inputs that are not necessarily correlated outright. The correlations need to be learnt by the fully connected network through the convolutions. In the first approach, the convolutions are performed on embeddings. The embedding space being constrained, there are spatial correlations present outright. This is presumably why the training graphs behave as they do.

The use of cost mask played a major role in the stabilizing the training process. The dataset being highly skewed in the lengths of the gestures was a problem. In some of the gestures, the edge-paddings occupied a majority of the duration of the gesture. While the model was expected to learn the paddings and when to freeze the output, this confused the model into trying to figure out the variations in the padding.

The model ultimately converged to a local minima where it outputs a constant posture which is presumably the mean of all the postures in the dataset. This could be because of the inability of the model in memorizing the gestures corresponding to the annotations. Possible direction forward is to explore larger encoder architectures and the importance of the presence of residual connections if and how they could affect the learning process.

Code

The code base is structured into modules and scripts.

Main

The main executable file is `main.py`. The code in the file processes arguments, builds config objects, constructs the `Data`, `Model` and the `Train` objects according to the arguments and performs train and evaluate actions on the models and data.

The list of parameters in a file pointed to by the `config` argument is parsed through and a dictionary (bundle) object is created. This config object is passed to the various Data objects, Models and Train objects as a set of hyperparameters based on which the objects are to be built.
Based on the `dataset` and the `architecture` arguments, the corresponding `Data`, `Model` and the appropriate `Train` objects are created.
The `model` argument specifies loading of saved parameters from a set of checkpoint files.
The `mode` argument specifies if the model is to be tested or trained with `test_index` argument pointing to the appropriate test.

Data

The `Data` object handles the preprocessing of the data, saving of data on disk in the post-processed form, normalizing, denormalizing of the data and fetching batches from the data for training purposes.

The files in the data/ folder correspond to the various datasets and sometimes the same dataset in different forms. Each file defines a `Data` class

with methods that process the dataset accordingly.
The $\_\_init\_\_()$ method loads from disk post processed data available in the location pointed to by the variable $self.datapath$. If not, the data is loaded from the location dir specified in the config object, processes it, splits into test/eval/train sets.
The $next\_batch()$ splits the train data into batches and serves when a function call is made.

Model

The $Model$ object handles the creation of the graph of the model. The code is divided into functions that apply/create the model on the placeholders, compute the cost functions, compute and apply the gradients to the weights according to the optimizer policy. The code is self explanatory. The $BaseModel$ class has the overall outline of the execution stream of all the models while the abstract functions are implemented in the specific model. The $BaseModel$ class also has the code for saving the models and loading them from checkpoint files.
One key feature of some of the models is that they are built for multi-gpu setups. This typically involves splitting the batch among the GPUs, computing the gradients according to the costs on each of the GPUs for the corresponding batch, averaging these updates and then applying them on the weights. The process is described in greater detail in the blog [2] and is reflected in the code in the implementation.

Trainer

The $Trainer$ object takes an instance of $Model$ and $Data$ and fetches batches from the $Data$ object while updating the weights of the $Model$.

[2]https://www.tensorflow.org/guide/using_gpu#using_multiple_gpus

The various trainers are built keeping the format of the batch in mind. The major types of $Trainers$ involved are classified into adversarial trainers and regular trainers and also depend on the type of the dataset.

The main one, $CSeqGANTrain$ first runs the pre-training epochs, and then performs the adversarial training for instance.

Execution

To run a fresh instance of training:

- Build an appropriate $Data$ class inheriting from the $BaseData$. Implement the $set\_data\_path()$ method to set the location of the npy files to be stored and read from, from the disk after post processing, all the post and preprocessing functions. Add the imports in the $\_\_init\_\_.py$ file in the data folder. Or use an existing $Data$ class. Prefer to end the name of the folder of the location of npy files with '_data' as then the folder will be ignored by the version control system.
- Inherit from the $BaseModel$ and follow the outline in the other models and implement $build\_model()$, $create\_placeholders()$ and other such functions. Add the imports to the $\_\_init\_\_.py$ file in the models folder. Or use an existing $Model$ class.
- Perform the same for the $Trainer$ class in the models folder though the general methods are implemented already.
- Add/update the appropriate config file to set the hyper parameters, the checkpoint and the data source directories.
- Check if the classes have been imported in the $main.py$ file and appropriately update the file to pick the right combination of $Data$, $Model$ and $Trainer$ classes.

- To run tests, add a function in one of the `tests[23].py` files or alternatively create a function in a new file and assign it a test index in the `main.py` file. The `test_index` argument serves as a pointer to the correct function.
- Refer to the sample `run.sh` file and adjust the parameters accordingly. Set the environment variable for the GPUs accordingly and execute `run.sh`.

The files located in the home folder 'user' on the system 'HPI-JP2262' involve scripts to convert csv to bvh files, copy of models, outputs to and from the server.

REFERENCES

Barsoum, E., J. Kender, and Z. Liu. 2017. HP-GAN: Probabilistic 3D human motion prediction via GAN. *ArXiv e-prints*. `1711.09561`.

Brock, Heike, and Kazuhiro Nakadai. 2018. Deep jslc: A multimodal corpus collection for data-driven generation of japanese sign language expressions.

Chen, Xi, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. 2016. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. *CoRR* abs/1606.03657. `1606.03657`.

Goodfellow, I. J., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. 2014. Generative Adversarial Networks. *ArXiv e-prints*. `1406.2661`.

Gulrajani, Ishaan, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. 2017. Improved training of wasserstein gans. *CoRR* abs/1704.00028. `1704.00028`.

Mirza, Mehdi, and Simon Osindero. 2014. Conditional generative adversarial nets. *CoRR* abs/1411.1784. `1411.1784`.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR* abs/1706.03762. `1706.03762`.

Wang, Yuxuan, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc V. Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. 2017. Tacotron: A fully end-to-end text-to-speech synthesis model. *CoRR* abs/1703.10135. `1703.10135`.

Wang, Yuxuan, Daisy Stanton, Yu Zhang, R. J. Skerry-Ryan, Eric Battenberg, Joel Shor, Ying Xiao, Fei Ren, Ye Jia, and Rif A. Saurous. 2018. Style tokens: Unsupervised style modeling, control and transfer in end-to-end speech synthesis. *CoRR* abs/1803.09017. `1803.09017`.