Using Generative Models for Semi-Supervised Learning

Daniel De Freitas Adiwardana
Stanford SCPD / Google
dadiward@stanford.edu

Akihiro Matsukawa
Stanford SCPD / Google
akihiro@stanford.edu

Jay Whang
Stanford SCPD / Google
jaywhang@stanford.edu

Abstract

The availability of large amounts of quality labeled data is a fundamental challenge of modern supervised learning. In this work, we explore the use of generative models on unlabeled data to improved the performance of a supervised classifier trained on few labeled examples. We consider two families of generative models: PixelRNN and DCGAN. We successfully apply these models to achieve significantly higher performance in supervised image recognition while using small amounts of labeled training examples.

1. Introduction

It is well known that training deep neural networks requires an abundance of data, and its most notable successes have been in areas where this condition is met. One category of major success is supervised learning where large labeled datasets exist, such as ImageNet [1]. Another category of success includes unsupervised tasks where labels are not necessary. Image generation is an example of such task. To this end autoregressive generative models such as PixelRNN [2] and Generative Adversarial Networks (GANs) [3] have become an active area of research. We used the MNIST [4] dataset as the benchmark for our experiments. So given a picture of a handwritten digit, one should classify it into one of ten different classes (i.e. digits 0 through 9). We also initiated work on the CIFAR-10 dataset [5], but only got as far as training a PixelRNN on it. The challenges we faced are explained later in section 6.2. For CIFAR-10, given a picture, one should classify it into one of 10 possible classes (e.g. airplane, bird, horse).

In practice, however, arguably the most common setting is where large amounts of unlabeled data exists, but we wish to train some supervised predictor. In many cases, it is prohibitively expensive to label all the data, so the labeled dataset is often many orders of magnitude smaller.

There are two main approaches that are commonly used in dealing with the challenge of having a small labeled dataset: transfer learning and semi-supervised learning. In this work, we investigate both: semi-supervised techniques for transferring learned representations and jointly training supervised and unsupervised models concurrently.

1.1. Transfer Learning

Transfer learning tries to gain performance from a larger *labeled* dataset for a different, but related task. It has been empirically observed that given enough data, features learned by deep learning models can generalize to another problem and thus can be used in, for example, a new classification task. Especially in computer vision, it has become a very common practice to reuse layers from large pre-trained networks such as Inception[6] or VGG [7].

1.2. Semi-Supervised Learning

Semi-supervised learning tries to take advantage of *unlabeled* data, ideally from the same distribution as the labeled data. One common class of techniques is based on label propagation, which attempts to “smear” labels from the labeled dataset to the unlabeled one based on some similarity heuristic [8]. Another class of techniques attempts to use the unlabeled data directly in the training objective, for example to first train an autoregressive model to initialize good weights, learn useful compressed representations of the data, or to train for some joint objective. It is this second class that we focus on in this paper, and we describe relevant prior works in section 2.

1.3. Learning From Unlabeled Data

We conjecture that like features in supervised models, features learned by generative models that are able to generate images from some distribution $P(X)$ must also be transferable to a supervised task on $P(Y|X)$ where X is a data point and Y is some output variable that we want to map X to. Intuitively, those features should be more representative than transferring from another supervised task, since the data comes from the same distribution.

We investigate two different classes of models for this purpose. The first is PixelRNN [2], an autoregressive model of the distribution over image pixels. The second is a *semi-supervised* Deep Convolutional GAN (DCGAN) [9], which frames the objective as a game between two neural networks while concurrently optimizing for the supervised objective.

1.4. Problem Formulation

Formally speaking, let $D = \{X, Y, X'\}$ be a dataset where (X, Y) are the labeled points, and X' is the rest of the unlabeled data, which is often orders of magnitude larger than X.

Our PixelRNN experiments involve first training a generative model $U(\{X, X'\})$ and then transferring its features to a subsequent supervised task $S(X, Y; U)$. *To our knowledge leveraging PixelRNNs for this type of semi-supervised learning has not been attempted yet.*

On the other hand, our semi-supervised DCGAN experiment involves a generative model being trained along with a discriminator using all of $\{X, Y, X'\}$ at the same time. The discriminator is used to compute both the adversarial loss and the classification loss. This approach was based on the work of Salimans et al. [10].

Finally the baseline model is trained only on the available labeled data $S'(X, Y)$.

The models are evaluated in their accuracy and the amount of labeled data required to converge to good results.

2. Related Work

There have been numerous studies that examine the use of generative models in a semi-supervised setting.

Salimans, et al. [10] report a way to utilize GANs for a classification task with K classes. Specifically, they propose an extension to the vanilla GAN where the labeled dataset is augmented with samples from the generator. The discriminator is also modified to predict $K + 1$ classes: the original K classes and a new class for fake (generated) data. In a sense this helps the discriminative model by augmenting a smaller labeled dataset with larger unlabeled set of real examples and generated samples. In [10], a fully connected generator network was used as noted from their published implementation [11]. *In the present work we replace it with a DCGAN and obtain a superior performance.*

Donahue *et al.* [12] describe an adversarial formulation with a third component, which they call the "encoder". While the generator maps a simple latent distribution to data space, the encoder attempts to encode real data to some latent space. They show that this encoder is capable of learning to invert the generator, and can be used as a featurizer for a supervised training.

On the autoregressive side, Dai and Le [13] explored the idea of first "pretraining" a sequence model to perform a task on unlabeled text data. These pretrained weights are then used to train supervised models for text classification. Their results show improved learning stability and model generalization.

Radford *et al.* [14] trained an mLSTM RNN on Amazon reviews to learn a language model and then used its internal cell state from the last time step as features for the subsequent supervised task of sentiment analysis of Amazon reviews. This enabled the authors to match the state of the art in their sentiment analysis dataset with significantly less labeled data and to surpass it with the full training set.

3. Methods

3.1. PixelRNN

PixelRNN [2] uses the chain rule of probability to decompose the distribution of pixel values and directly models that distribution with a 2-dimensional LSTM [15]. It uses masking to ensure that only the pixels previously seen by the model are used to predict the next one. Masking also ensures that the receptive field does not contain the value being predicted, or any "future" pixels. [2] presents three separate architectures: PixelCNN, which employs a fully convolutional network with visibility over local pixels; RowLSTM, which has a conal receptive field above the pixel; and DiagonalBiLSTM, which has full visibility over all previous pixel values. The receptive field of the DiagonalBiLSTM used is shown in figure 2, reproduced from [2]. We only use DiagonalBiLSTM, as the authors report that it achieves the best performance.

Note that for computational efficiency, PixelRNN uses convolutions in place of the standard matrix product inside each LSTM cell.

$$\begin{aligned}[o_i, f_i, i_i, g_i] &= \sigma(K^{ss} \star h_{i-1} + K^{is} \star x_i)\\ c_i &= f_i \odot c_{i-1} + i_i \odot g_i\\ h_i &= o_i \odot c_i\end{aligned}$$

In the equation above, $\star$ represents the convolution operator, while $\odot$ represents element-wise product. σ is a non-linearity, according to the standard definitions.

For efficiency, $K^{is} \star x_i$ is pre-computed over the entire image. First, we use a "mask A" convolution (shown in figure 1) to ensure the correct "visibility" for the pixel, then we use a 1×1 convolution to arrive at the correct dimensions. We use $K^{is} \star x_i$ to represent this entire operation.

For hidden states, we would like to derive values for a pixel given hidden states from the pixels above it, and to the left of it. A skew operation (show in figure 2) ensures that the desired receptive field forms a column in the skewed image. The hidden state operation is computed column-wise on this skewed image. That is, each $K^{ss} \star h_{i-1}$ computes the hidden state components along an entire column i.

The PixelRNN first learns a $W \times H \times Z$ representation of the image where Z is the LSTM hidden dimension size, then uses output layers to produce a distribution $W \times H \times C \times D$ for each image, where D is the domain of each pixel channel. Note that to ensure correct masking, we need to predict each channel separately. We use these intermediate $W \times H \times Z$ representation for semi-supervised learning.

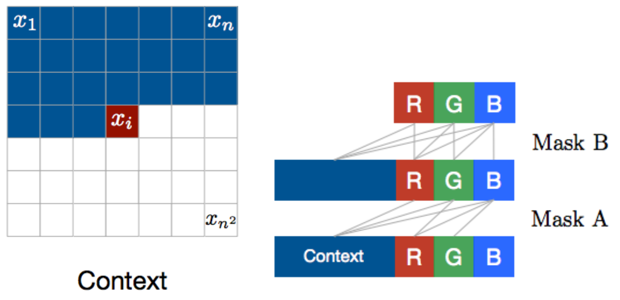

Figure 1. **Left:** To generate pixel x_i one conditions on all the previously generated pixels left of and above x_i. **Right:** Diagram of the connectivity inside a masked convolution. In the first layer, each of the RGB channels is connected to previous channels and to the context, but is not connected to itself. In subsequent layers, the channels are also connected among themselves. This figure and caption were extracted from Oord *et al.* [2].

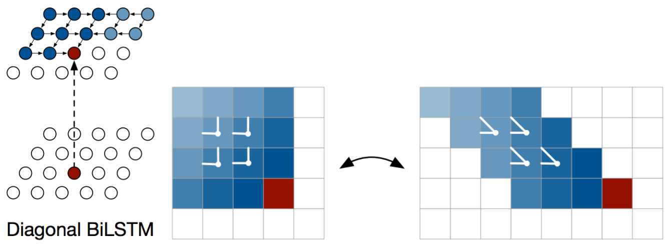

Figure 2. Visualization of the input-to-state and state-to-state mappings for the three proposed architectures. This figure and caption were extracted from Oord *et al.* [2].

3.2. Semi-Supervised DCGAN

By combining the DCGAN architecture and the work of Salimans et al. [10], we created a semi-supervised DCGAN model where the discriminator loss consists of three components:

- Supervised loss: the cross-entropy loss from the predicted distribution over K $(= 10)$ digit classes: $-\mathbb{E}_{\boldsymbol{x},y\sim p_{data}}[\log p_{model}(y|\boldsymbol{x}, y < K + 1)]$
- Unsupervised loss: the loss from classifying unlabeled data points as real, i.e. class $\neq K + 1$: $-\mathbb{E}_{\boldsymbol{x}\sim p_{data}}[\log(1 - p_{model}(y = K + 1|\boldsymbol{x}))]$
- GAN sample loss: the loss from classifying generated images as fake, i.e. class $= K + 1$: $-\mathbb{E}_{\boldsymbol{x}\sim G}[\log p_{model}(y = K + 1|\boldsymbol{x})]$

Notice that an artificial "fake" class is added, corresponding to the class $K + 1$. This approach allows us to jointly train the discriminative network to serve two functions: as a supervised classifier over K classes, and as a discriminator between real and fake images (K real classes vs. the "fake" class).

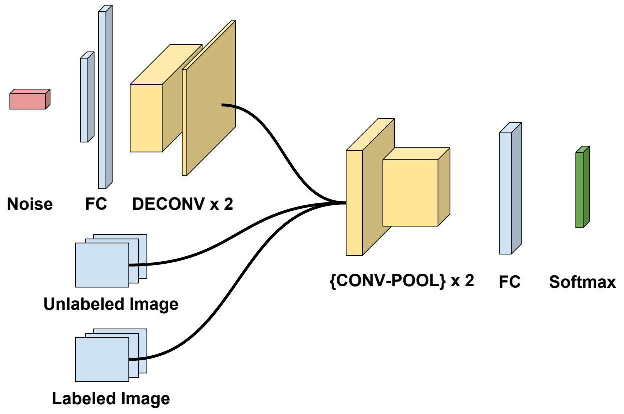

Figure 3. Architecture of the semi-supervised DCGAN model. Notice that the discriminator network receives three inputs and now serves as a classifier for target classes.

The generator uses batch-normalized [16] transposed convolutional layers as originally suggested in the DCGAN paper [9], with fully-connected layers in the beginning. Unlike Salimans, et al., we used the regular GAN generator loss instead of feature matching.

4. Dataset

We measure the performance of the semi-supervised techniques on the MNIST digit classification dataset [4] and the CIFAR10 dataset [5]. For both, the (semi-)supervised model classifies a given picture into one of the possible classes. Our baseline is a convolutional neural network (CNN). We use the same architecture for the discriminator for GAN experiments to provide a fair comparison on classification performance. One key metric of interest is the number of labeled data points required to reach similar level of performance as the baseline model because this is a strong indicator that we are benefiting from unlabeled data.

5. Implementation

We have used TensorFlow [17] to implement the following: our own implementation of PixelRNN (with Diagonal BiLSTM) to produce image embeddings, a supervised CNN classifier that takes either PixelRNN embeddings or raw image pixels as input, and our own DCGAN-based model that uses the semi-supervised learning framework suggested by Salimans et al. [10]. We would like to acknowledge that the open source PixelRNN implementation [18] written in Theano [19] which is limited to MNIST and that we used it for initial explorations until our own implementation was complete.

6. Experiments

6.1. Baseline CNN Architecture and Hyperparameters

We use the following architecture which achieves 99.31% test accuracy on the full MNIST training set:

- Layer 1: Convolutional layer

  32 5x5 filters, padding: same, activation: relu

- Layer 2: Max pooling layer

  2x2, stride: 2

- Layer 3: Convolutional layer

  64 5x5 filters, padding: same, activation: relu

- Layer 4: Max pooling layer

  2x2, stride: 2

- Layer 5: Dense layer

  output: 1024, activation: relu

  dropout: 40% drop rate

- Layer 6: Dense layer (logits)

  output: 10, activation: softmax

We use mini-batches of 100 examples and the Adam optimizer with learning rate of 0.001.

For the rest of this paper, we will focus on measuring the performance of our semi-supervised models at different amounts of labeled examples in comparison with the baseline CNN. The expectation is that through semi-supervised learning techniques the use of unlabeled data should allow us to perform well even with much fewer labeled examples.

6.2. PixelRNN Model Details

We trained a DiagonalBiLSTM PixelRNN model with the following hyperparameters for CIFAR-10. Like the authors of [2], we used a batch size of 16, an input (mask A) kernel size of 7×7, LSTM hidden size of 128, and hidden output layers of size 1024. We used an exponentially decaying learning rate schedule, starting with 1e-4 and decaying by a factor of 0.95 every 1000 batches.

However, we found our access to infrastructure and time constraints to be limiting. Unlike [2], we trained only 5 residual layers instead of 12, which is all we could fit on the memory of a single GTX1080 GPU. While no timing information is given in [2] in the README of the PixelCNN++ [20], it is claimed that it took 10 hours on 8 TitanX GPUs to reach a negative log-likelihood (NLL) of 3.0. Since the PixelRNN trains even slower than PixelCNN, we would expect this time to be longer. Note that we only cite [20] because it provides a data point on required training time.

For results presented here, we trained the model for 75K iterations, which equates to approximately 2 days, and reached a test set NLL of 3.59. Despite a higher NLL, however, we found samples generated to be visually similar to those in [2]. A example can be found in Figure 4.

Albeit somewhat subjective, we observed that the training curve was still trending downwards. Aside from training longer, we also saw a large amount of fluctuations in the NLL evaluated over our batch size of 16, and we believe that evaluating NLL periodically more accurately on a larger validation set and selecting a good checkpoint would have helped us discover a better model. However, such an evaluation would have extended our training period significantly, so we opted to take the last checkpoint of the model.

6.3. Samples from Generative Models

The generative models PixelRNN and DCGAN were both trained on MNIST training data (without the labels). The PixelRNN was additionally also trained on CIFAR-10. As a sanity check to make sure our generative are working after having trained them we generate samples. Figure 4 shows samples of images generated by the models we implemented. The generated digit images for MNIST resemble the images from the training data distribution, look fairly crisp and are diverse. The CIFAR-10 based image samples do not look as good, although some local patterns similar to the ones in the real images did emerge.

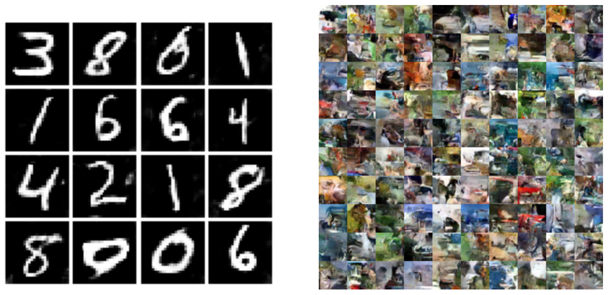
Figure 4. Samples from DCGAN trained on MNIST (left) and PixelRNN trained on CIFAR-10 (right).

6.4. Obtaining PixelRNN Embeddings

After training our PixelRNN model on MNIST training data, we used it to generate embeddings for MNIST images by taking the hidden states at every pixel of each image. This effectively transforms a single example into a 28x28x64 input map, which was then fed as input features for supervised classification. In other words, for each MNIST image we obtained a 28x28x64 embedding containing features learned via the unsupervised training of the PixelRNN model. Note that the PixelRNN's hidden state is 64 dimensional and that there are 28x28 RNN time steps involved, one for each pixel in the input image.

6.5. Using PixelRNN Embeddings as Features

We experimented replacing image pixels with the PixelRNN embeddings as input features to the same baseline supervised architecture CNN outlined in section 6.1. Note that the embedding for each example is a 28x28x64 volume. An embedding is fed to the first CNN layer as if it was a 28x28 image with 64 channels. We then trained the supervised model with varying amounts of labeled data by taking subsets of the original MNIST training dataset. Figure 5 shows how the MNIST test accuracy varies with the log of the number of training examples when using the embeddings vs using the image pixels. We also tried including the image pixels as a 65th channel in the input features, but that did not yield any performance gains.

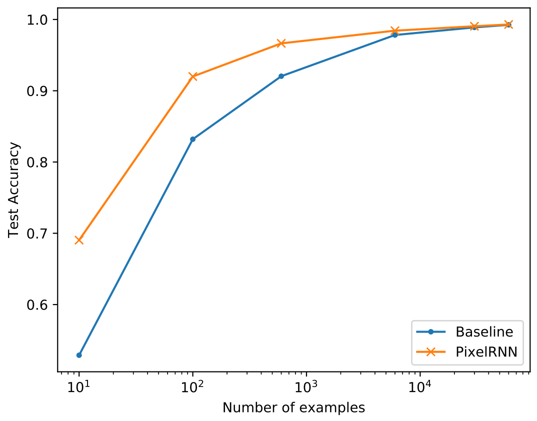

Figure 5. MNIST test set accuracy of using PixelRNN embeddings vs. baseline usage of pixels. The classifier is a CNN for both.

In figure 5 one can see that using embeddings makes much better use of smaller amounts of labels. In the extreme low end with only 1 example of each digit (10 training examples in total) using embeddings yields a test accuracy of 69.04% versus 52.89% when directly using pixels. That is a 30.54% increase in performance by using embeddings alone when training with just 10 examples. As the number of examples increases the gap decreases as expected due to the diminishing return from embedding learned from the unlabeled model. Better quality embeddings could potentially increase that gap between using embeddings and image pixels.

6.6. The "Golden 10" Features

Each image PixelRNN embedding comprises 50,176 individual unsupervised features/real numbers (i.e. 28x28x64).

Using all of these features with just a linear classifier yields **99%** test accuracy.

We then trained a linear model with just the **ten** features that maximize the Pearson linear correlation scores (instead of 50,176) for each of the 10 digit classes in the training set. With this setup we obtained test accuracy of **84.16%** indicating the existence of unsupervised features that are strongly correlated with the high-level concept of digit types in the embeddings. This is surprising in the sense that these features were learned without any involvement of human provided labels. Notice that this is in the similar vein as [14], where a specific neuron trained without labels corresponded directly to the sentiment of an Amazon review.

Table 1 shows the Pearson linear correlation scores for the top feature for each digit class.

Table 1. Pearson correlation for top feature for each class

Digit class	Pearson correlation
0	0.72
1	-0.79
2	-0.60
3	0.52
4	0.61
5	-0.57
6	0.79
7	0.62
8	-0.57
9	0.56

6.7. Visualizing the Influence of PixelRNN Time Steps in the Quality of the Embedding Features

We plot the weights of a linear classifier trained using embeddings, MNIST labels and $L1$ regularization to find which of the embedding time steps were the most predictive. The $L1$ regularization enforces weight sparsity such that less predictive weights tend to go to zero. This helps increase the contrast between our most useful weights and the least useful ones for visualization.

The full weight matrix for the linear classifier with 10 output classes and using embeddings as input is 28x28x64x10. For each class we have 28x28x64 weights. By reducing on the last axis by taking the maximum absolute value we obtain a 28x28 image of the largest weights in absolute value. Figure 6 shows such weights for the digit class 3.

We observe that embeddings from earlier time steps are not very useful, the middle ones are very useful and the very last ones (bottom-rightmost) are also not very useful. This matches the following intuition: earlier steps/pixels have accrued less information, the very last steps might have "forgotten" the earlier information as they ingested mostly dark pixels towards the end and the middle steps strike a balance.

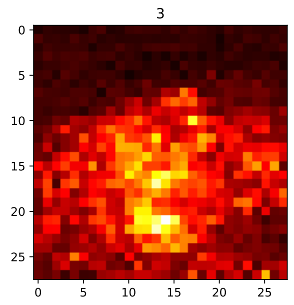

Figure 6. Weights for digit '3' of linear classifier trained with PixelRNN embeddings and $L1$ regularization. The warmer the color, the higher the absolute value of the weight.

6.8. Semi-Supervised DCGAN

We trained semi-supervised DCGAN models with varying amounts of labeled data by taking subsets of the original MNIST training dataset. Figure 7 shows how the MNIST test accuracy varies with the log of the number of training examples when using the semi-supervised DCGAN vs supervised CNN.

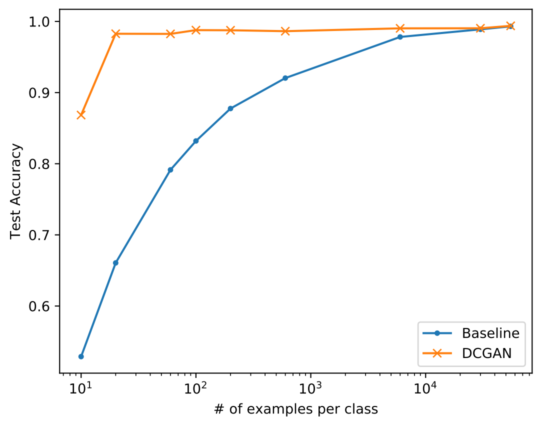

Figure 7. MNIST test accuracy, trained on subsets of training data: baseline CNN vs. DCGAN-based model. Notice that the DCGAN-based model performs much better than the baseline model, especially when there are very few examples. It achieves 84% accuracy when only single image is provided for each class for a total of 10 labeled training examples.

In figure 7 one can see that using a semi-supervised objective which basically multi-tasks between the supervised CNN objective and the standard DCGAN objective makes much better use of small amounts of labels. In the extreme low end with only 1 example of each digit (10 training examples in total) the semi-supervised model achieves a test accuracy of surprisingly high 84% vs 52.89% when only using the supervised objective. That is a 58.8% increase in performance when training with just 10 examples. Just as in the PixelRNN embeddings case, as the number of labeled examples increases the gap with the baseline decreases.

7. Conclusion

Generative models can learn representations that are highly predictive of a high-level concept (e.g. sentiment from a text, the digit of handwritten number). Often in supervised learning, only human reviewers can provide such information, which can be a scarce resource. PixelRNN embeddings promoted an increase in test accuracy of up to 30% when only training with 1 example of each digit. Augmenting the supervised loss with the DCGAN's unsupervised loss allowed us to obtain 96% test accuracy with only 50 examples. Further improvements to generative models and scaling up to larger amounts of unlabeled data could yield even more performance gains on supervised tasks and further promote the efficient use of very small amounts of labeled data. The success of such semi-supervised learning techniques shows that one can still train a performant model for supervised tasks in which collecting the necessary amount of labeled data is prohibitively expensive.

8. Appendix

8.1. Code Release

We intend to open source our TensorFlow PixelRNN implementation and our semi-supervised DCGAN. As far as we are aware, there is no available open source semi-supervised DCGAN implementation. Also, it will be the first open source implementation that supports fitting images with multiple channels, as we demonstrate on the CIFAR-10 [5] dataset. This is a non-trivial addition, as implementing the correct masking over multiple channels requires complex management of network connections. In addition, the implementation benefits from TensorFlow features such as model checkpointing and using Tensorboard to monitor loss and generated samples over the course of training. The DiagonalBiLSTM is also a complex architecture that required implementation of a custom masked convolutional kernel, a custom RNN cell, use of the `tf.contrib.rnn` API, and complex management of TensorFlow scopes via `tf.make_template`. We hope that this implementation will aid further research into similar architectures, and also in general illustrate the usage of these TensorFlow functionalities.

Since all three authors are employed by Google where [2] was original implemented, we are waiting for internal approval to open source this work.

References

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein,

A. C. Berg, and F. Li, “Imagenet large scale visual recognition challenge,” *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575

[2] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu, “Pixel recurrent neural networks,” *CoRR*, vol. abs/1601.06759, 2016. [Online]. Available: http://arxiv.org/abs/1601.06759

[3] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[4] Y. LeCun and C. Cortes, “MNIST handwritten digit database,” 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[5] A. Krizhevsky, V. Nair, and G. Hinton, “Cifar-10 (canadian institute for advanced research).” [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[6] C. Szegedy, S. Ioffe, and V. Vanhoucke, “Inception-v4, inception-resnet and the impact of residual connections on learning,” *CoRR*, vol. abs/1602.07261, 2016. [Online]. Available: http://arxiv.org/abs/1602.07261

[7] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[8] A. Kannan, K. Kurach, S. Ravi, T. Kaufmann, A. Tomkins, B. Miklos, G. Corrado, L. Lukács, M. Ganea, P. Young, and V. Ramavajjala, “Smart reply: Automated response suggestion for email,” *CoRR*, vol. abs/1606.04870, 2016. [Online]. Available: http://arxiv.org/abs/1606.04870

[9] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” *CoRR*, vol. abs/1511.06434, 2015. [Online]. Available: http://arxiv.org/abs/1511.06434

[10] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training gans,” *CoRR*, vol. abs/1606.03498, 2016. [Online]. Available: http://arxiv.org/abs/1606.03498

[11] OpenAI, “Code for the paper ”improved techniques for training gans.” [Online]. Available: https://github.com/openai/improved-gan

[12] J. Donahue, P. Krähenbühl, and T. Darrell, “Adversarial feature learning,” *CoRR*, vol. abs/1605.09782, 2016. [Online]. Available: http://arxiv.org/abs/1605.09782

[13] A. M. Dai and Q. V. Le, “Semi-supervised sequence learning,” *CoRR*, vol. abs/1511.01432, 2015. [Online]. Available: http://arxiv.org/abs/1511.01432

[14] A. Radford, R. Józefowicz, and I. Sutskever, “Learning to generate reviews and discovering sentiment,” *CoRR*, vol. abs/1704.01444, 2017. [Online]. Available: http://arxiv.org/abs/1704.01444

[15] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[16] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[18] I. Gulrajani, “Working theano implementation of pixel rnn on mnist.” [Online]. Available: https://github.com/igul222/pixel_rnn

[19] Theano Development Team, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[20] OpenAI, “Python3 / tensorflow implementation of pixelcnn++.” [Online]. Available: https://github.com/openai/pixel-cnn