Model Generation

ylouzoun, Shilo Avital, Sarel Lieberman

October 2024

Abstract

This paper explores a novel approach to simplifying the machine learning training process by utilizing generative models to produce optimized weights for classification tasks, specifically using the Iris dataset. We investigate the potential of a Variational Autoencoder framework to generate weight matrices for single-layer and two-layer neural networks without relying on the original training data. Our experiments reveal that the generated weights exhibit comparable performance to those obtained through conventional training methods, despite differences in their underlying structures. Our findings contribute to the emerging discourse on hypernetworks and highlight the feasibility of using generative models to enhance the machine learning pipeline by reducing the complexity associated with hyperparameter tuning and model architecture optimization.

1 Introduction

In recent years, the field of machine learning has witnessed tremendous advancements, primarily driven by the development of complex models capable of learning intricate patterns from vast datasets. However, these models often require extensive training, fine-tuning, and optimization of hyperparameters, which can be time-consuming and computationally expensive. This raises an intriguing question: can we develop a generative model that produces the weights for a machine learning model without relying on the original training data?

This research aims to explore this possibility by investigating a novel approach in which a generative model can generate the weights for a given model architecture. The ultimate goal of this endeavor is ambitious: to create a general model that takes a specific task as input and generates both the architecture and the optimized weights tailored for that task. If successful, this approach could significantly simplify the training process and alleviate the burdens associated with architecture optimization and hyperparameter tuning.

Recent studies in the domain of hypernetworks have explored innovative techniques for optimizing model architectures and weight generation. For example, the work titled *HyperNetworks* [1] delves into the concept of a hypernetwork that generates weights for another network, employing end-to-end training

with backpropagation to enhance performance while maintaining fewer learnable parameters. In a similar vein, *SMASH: One-Shot Model Architecture Search through HyperNetworks* [2] proposes a technique that utilizes an auxiliary hypernetwork to generate weights based on the model's architecture, enabling architecture selection through efficient single training runs.

In this context, our study focuses on employing a Variational Autoencoder (VAE) [3] framework to generate optimized weights specifically for classification tasks using the Iris dataset [4]. By examining how the generated weights perform on this well-established dataset, we aim to uncover insights into the behavior of weights produced by our generative model, emphasizing the simplification of the training process and the reduction of hyperparameter tuning complexities. This initial investigation will serve as a foundation for understanding the broader implications of our research, particularly in terms of robustness to overfitting and handling missing data.

Through this work, we seek to contribute to the ongoing discourse in the field of hypernetworks, highlighting the potential of generative models in simplifying and enhancing the machine learning pipeline.

2 Data

2.1 Iris Dataset

The Iris dataset contains 150 samples from three species: Setosa, Versicolor, and Virginica. Each sample is represented by a feature vector $\mathbf{x} \in \mathbb{R}^4$, consisting of sepal and petal lengths and widths. We split the dataset into 100 training samples and 50 test samples.

2.2 Weight Matrices Dataset

We trained 500 single-layer neural networks on the Iris dataset to create weight matrices as data for the generative model. Each model $\mathcal{M}_1$ consists of a single linear layer:

$$\mathcal{M}_1(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}, \quad \mathbf{W} \in \mathbb{R}^{3\times 4}, \quad \mathbf{b} \in \mathbb{R}^3 \tag{1}$$

These models, trained on the true labels, are referred to as "Real" models. We define the set of real models as:

$$\mathcal{R}_1 = \{\mathcal{M}_1^i \mid \mathcal{M}_1^i \text{ trained on Iris with true labels}\}_{i=1}^{500}$$

Similarly, we trained 500 additional models with the same architecture but used randomly shuffled labels. These are called "Scrambled" models:

$$\mathcal{S}_1 = \{\mathcal{M}_1^i \mid \mathcal{M}_1^i \text{ trained on Iris with scrambled labels}\}_{i=1}^{500}$$

Each model in the sets is saved as a single vector, where the bias term is appended to the flattened weight matrix: $\mathbf{w} \coloneqq \text{vec}(\mathbf{W}) \oplus \mathbf{b} \in \mathbb{R}^{15}$.

To extend the experiment, we trained 500 two-layer neural networks, denoted as $\mathcal{M}_2$. These models consist of two linear layers, with a ReLU[1] activation applied between the layers. The architecture can be described as:

$$\mathcal{M}_2(\mathbf{x}) = \mathbf{W}_2 \cdot \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{2}$$

$$\mathbf{W}_1 \in \mathbb{R}^{5\times4}, \quad \mathbf{b}_1 \in \mathbb{R}^5, \quad \mathbf{W}_2 \in \mathbb{R}^{3\times5}, \quad \mathbf{b}_2 \in \mathbb{R}^3$$

These models form the "Real-2" model set:

$$\mathcal{R}_2 = \{\mathcal{M}_2^i \mid \mathcal{M}_2^i \text{ trained on Iris with true labels}\}_{i=1}^{500}$$

Similarly, models in $\mathcal{R}_2$ are saved as tuples of vectors with bias terms appended to flattened matrices:

$$(\mathbf{w}_1, \mathbf{w}_2) \coloneqq (\text{vec}(\mathbf{W}_1) \oplus \mathbf{b}_1, \text{vec}(\mathbf{W}_2) \oplus \mathbf{b}_2) \in \mathbb{R}^{25} \times \mathbb{R}^{18}$$

All models were trained using the cross-entropy loss function $\mathcal{L}_{\text{CE}}$:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{C} y_i \log \hat{y}_i \tag{3}$$

where C represents the number of classes, y_i is the true label, and $\hat{y}_i$ is the predicted probability for each class computed using the softmax function:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}} \tag{4}$$

where z_i is the output of the model for class i. We used the Adam [5] optimizer with a learning rate of $\eta = 0.005$, and each model was trained for 200 epochs.

3 Architecture

In this section, we introduce the generative models for producing weight matrices of the single-layer models, $\mathcal{M}_1$, and two-layer models, $\mathcal{M}_2$, described in the data section. Our approach leverages the VAE framework, which we modify for each model type. For the single-layer model, we directly apply a VAE architecture, while for the two-layer model, we design a more complex architecture that builds on the VAE structure.

3.1 VAE Architecture for $\mathcal{M}_1$

The first generative model we discuss is applied to the single-layer models, $\mathcal{M}_1$. The input to this model is weight vector from the set of real models: $\mathbf{w} \in \mathcal{R}_1 \subseteq \mathbb{R}^{15}$. The goal of the VAE is to learn a latent representation of these weight vectors and to generate new, similar weight vectors from this learned space.

[1]The ReLU activation function is defined as $\text{ReLU}(x) = \max(0, x)$.

The architecture of the VAE consists of an encoder, which transforms the input $\mathbf{w}$ into a latent representation, and a decoder, which reconstructs the original $\mathbf{w}$ from the latent space. The encoder and decoder use separate sets of weight matrices, denoted here as $\mathbf{A}_i$, to avoid confusion with the classifier weights.

- **Encoder:**

$$\begin{aligned} \mathbf{h}_1 &= \mathrm{ReLU}(\mathbf{A}_1\mathbf{w} + \mathbf{b}_1), & \mathbf{h}_1 &\in \mathbb{R}^{128} \\ \mathbf{h}_2 &= \mathrm{ReLU}(\mathbf{A}_2\mathbf{h}_1 + \mathbf{b}_2), & \mathbf{h}_2 &\in \mathbb{R}^{64} \\ \mathbf{h}_3 &= \mathbf{A}_3\mathbf{h}_2 + \mathbf{b}_3, & \mathbf{h}_3 &\in \mathbb{R}^{32} \end{aligned}$$

- **Latent Variables:**

$$\begin{aligned} \boldsymbol{\mu} &= \mathbf{A}_4\mathbf{h}_3 + \mathbf{b}_4, & \boldsymbol{\mu} &\in \mathbb{R}^{32} \\ \log \boldsymbol{\sigma}^2 &= \mathbf{A}_5\mathbf{h}_3 + \mathbf{b}_5, & \boldsymbol{\sigma} &\in \mathbb{R}^{32} \\ \mathbf{z} &= \boldsymbol{\mu} + \boldsymbol{\epsilon} \odot \boldsymbol{\sigma}, & \mathbf{z} &\in \mathbb{R}^{32},\ \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \end{aligned}$$

- **Decoder:**[2]

$$\begin{aligned} \mathbf{h}_4 &= \mathrm{ReLU}(\mathbf{A}_6\mathbf{z} + \mathbf{b}_6), & \mathbf{h}_4 &\in \mathbb{R}^{64} \\ \mathbf{h}_5 &= \mathrm{ReLU}(\mathbf{A}_7\mathbf{h}_4 + \mathbf{b}_7), & \mathbf{h}_5 &\in \mathbb{R}^{128} \\ \hat{\mathbf{w}} &= \sigma(\mathbf{A}_8\mathbf{h}_5 + \mathbf{b}_8), & \hat{\mathbf{w}} &\in \mathbb{R}^{15} \end{aligned}$$

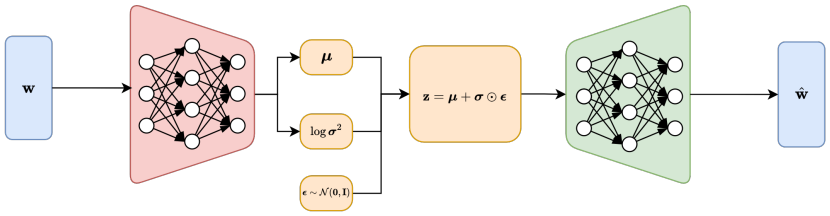

Figure 1: Architecture of the generative model for the single-layer models $\mathcal{M}_1$.

3.1.1 Training Process

Before training, the weight vectors in $\mathcal{R}_1$ were scaled to $[0, 1]$. The VAE was trained using the following loss function:

$$\mathcal{L} = \mathcal{L}_{\mathrm{MSE}}(\hat{\mathbf{w}}, \mathbf{w}) + D_{\mathrm{KL}}(q(\mathbf{z}|\mathbf{w})||p(\mathbf{z}))$$

where $\mathcal{L}_{\mathrm{MSE}}$ measures the reconstruction error, and D_{KL} is the Kullback-Leibler divergence between the approximate posterior $q(\mathbf{z}|\mathbf{w})$, which models the latent

[2]Here, σ refers to the sigmoid function, which is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$.

representation as a Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})$, and the prior $p(\mathbf{z})$, assumed to be a standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$.

The Mean Squared Error (MSE) is defined as:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{w}_i - \hat{\mathbf{w}}_i\|^2$$

The Kullback-Leibler Divergence is given by:

$$D_{\text{KL}}(q(\mathbf{z}|\mathbf{w})||p(\mathbf{z})) = -\frac{1}{2}\sum_{j=1}^{k}\left(1 + \log(\boldsymbol{\sigma}_j^2) - \boldsymbol{\mu}_j^2 - \boldsymbol{\sigma}_j^2\right)$$

The model was trained for 100 epochs using the Adam optimizer with a learning rate of $\eta = 0.003$.

3.2 Enhanced-VAE Architecture for $\mathcal{M}_2$

The second generative model we discuss is applied to the two-layer models, $\mathcal{M}_2$. The input to this model consists of two weight vectors, $(\mathbf{w}_1, \mathbf{w}_2)$, from the set of real models: $(\mathbf{w}_1, \mathbf{w}_2) \in \mathcal{R}_2 \subseteq \mathbb{R}^{25} \times \mathbb{R}^{18}$. Here, $\mathbf{w}_1$ and $\mathbf{w}_2$ correspond to the flattened weights and biases of the first and second layers of the neural network, respectively.

Due to the correlation between the two layers' weights, a simple concatenation of $\mathbf{w}_1 \oplus \mathbf{w}_2$ into a regular VAE would fail to capture the joint dependencies. This is because the VAE aproximate the posterior as a Gaussian distribution with a diagonal covariance matrix, which struggles to capture such correlations. To address this, our architecture first encodes $\mathbf{w}_2$, then processes the combined $\mathbf{w}_1$ and the latent representation of $\mathbf{w}_2$ jointly.

- **Encoder:**[3]
  - First Encoder Block:

$$\begin{aligned}
\mathbf{h}_1 &= \text{ReLU}(\text{BatchNorm}(\mathbf{A}_1\mathbf{w}_2 + \mathbf{b}_1)), && \mathbf{h}_1 \in \mathbb{R}^{16} \\
\mathbf{h}_2 &= \text{ReLU}(\text{BatchNorm}(\mathbf{A}_2\mathbf{h}_1 + \mathbf{b}_2)), && \mathbf{h}_2 \in \mathbb{R}^{8}
\end{aligned}$$

  - Second Encoder Block:

$$\begin{aligned}
\mathbf{h}_3 &= \text{ReLU}(\text{BatchNorm}(\mathbf{A}_3[\mathbf{w}_1 \oplus \mathbf{h}_2] + \mathbf{b}_3)), && \mathbf{h}_3 \in \mathbb{R}^{16} \\
\mathbf{h}_4 &= \text{ReLU}(\text{BatchNorm}(\mathbf{A}_4\mathbf{h}_3 + \mathbf{b}_4)), && \mathbf{h}_4 \in \mathbb{R}^{8}
\end{aligned}$$

[3]Given an input $\mathbf{x}$, Batch Normalization [6] computes the batch mean μ and variance σ^2, and normalizes the input as follows: $\hat{\mathbf{x}} = \frac{\mathbf{x}-\mu}{\sqrt{\sigma^2+\epsilon}}$, where ϵ is a small constant for numerical stability. The normalized output is then transformed using learnable parameters γ and β: $\mathbf{y} = \gamma\hat{\mathbf{x}} + \beta$.

- **Latent Variables:**

$$\begin{aligned} \boldsymbol{\mu} &= \mathbf{A}_5\mathbf{h}_4 + \mathbf{b}_5 & \boldsymbol{\mu} &\in \mathbb{R}^8 \\ \log \boldsymbol{\sigma}^2 &= \mathbf{A}_6\mathbf{h}_4 + \mathbf{b}_6 & \boldsymbol{\sigma} &\in \mathbb{R}^8 \\ \mathbf{z} &= \boldsymbol{\mu} + \boldsymbol{\epsilon} \odot \boldsymbol{\sigma}, & \mathbf{z} &\in \mathbb{R}^8,\ \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \end{aligned}$$

- **Decoder:**
  - First Decoder Block:

$$\begin{aligned} \mathbf{h}_5 &= \text{ReLU}(\text{BatchNorm}(\mathbf{A}_7\mathbf{z} + \mathbf{b}_7)), & \mathbf{h}_5 &\in \mathbb{R}^{16} \\ \hat{\mathbf{w}}_1 &= (\mathbf{A}_8\mathbf{h}_5 + \mathbf{b}_8), & \hat{\mathbf{w}}_1 &\in \mathbb{R}^{25} \end{aligned}$$

  - Second Decoder Block:

$$\begin{aligned} \mathbf{h}_6 &= \text{ReLU}(\text{BatchNorm}(\mathbf{A}_9[\mathbf{z} \oplus \hat{\mathbf{w}}_1] + \mathbf{b}_9)), & \mathbf{h}_6 &\in \mathbb{R}^{16} \\ \hat{\mathbf{w}}_2 &= (\mathbf{A}_{10}\mathbf{h}_5 + \mathbf{b}_{10}), & \hat{\mathbf{w}}_2 &\in \mathbb{R}^{18} \end{aligned}$$

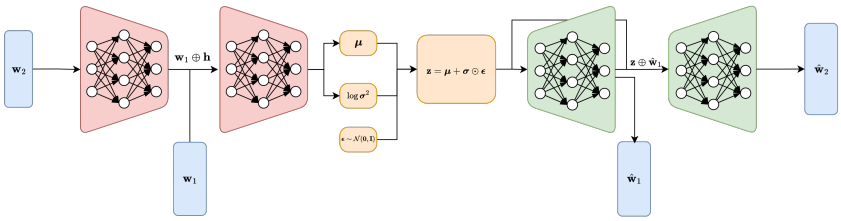

Figure 2: Architecture of the generative model for the two-layer models $\mathcal{M}_2$.

3.2.1 Training Process

Before training, the weight vectors in $\mathcal{R}_1$ were scaled to $[-1, 1]$ and then standardized. The enhanced-VAE was trained using the following loss function:

$$\mathcal{L} = \mathcal{L}_{\text{MSE}}(\hat{\mathbf{w}}_1, \mathbf{w}_1) + \mathcal{L}_{\text{MSE}}(\hat{\mathbf{w}}_2, \mathbf{w}_2) + D_{\text{KL}}(q(\mathbf{z}|\mathbf{w}_1, \mathbf{w}_2)||p(\mathbf{z}))$$

where $\mathcal{L}_{\text{MSE}}$ measures the reconstruction error for both $\mathbf{w}_1$ and $\mathbf{w}_2$. Again, D_{KL} is the Kullback-Leibler divergence between the approximate posterior $q(\mathbf{z}|\mathbf{w}_1, \mathbf{w}_2) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})$ and the prior $p(\mathbf{z}) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

The model was trained for 100 epochs using the Adam optimizer with a learning rate of $\eta = 0.1$ and L_2 regularization with $\lambda = 0.003$.

4 Results

In this section, we present the findings from our experiments aimed at evaluating the performance of the weight matrices obtained from our generative models. We define the sets of generated weights for the models $\mathcal{M}_1$ and $\mathcal{M}_2$ as $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, both of size 500.

4.1 Performance

To evaluate the models, we use the average classification accuracy across the Iris test dataset for all sets of models: $\mathcal{R}_1$, $\mathcal{S}_1$, $\mathcal{G}_1$, $\mathcal{R}_2$, and $\mathcal{G}_2$. The results are summarized in Table 1.

Table 1: Average classification accuracy (in percentage)

Model Set	Average Accuracy	Standard Deviation
$\mathcal{R}_1$	97.68	01.10
$\mathcal{S}_1$	34.26	19.10
$\mathcal{G}_1$	97.97	00.23
$\mathcal{R}_2$	95.64	09.96
$\mathcal{G}_2$	83.44	11.23

The results indicate that the generated weights $\mathcal{G}_1$ show a slightly higher average accuracy than the real weights $\mathcal{R}_1$, with a lower standard deviation. This suggests that the generative model effectively captures the key characteristics of the real weight matrices, producing consistent models. In contrast, $\mathcal{G}_2$ demonstrates a significant drop in average accuracy and higher standrad deviation compared to its counterpart $\mathcal{R}_2$ indicating a need for further investigation into the generative process for more complex architectures. As for $\mathcal{S}_1$, no surprise here, the models trained on scrambled labels perform, on average, like random guessing, but with notably higher variability.

4.2 t-SNE Analysis

To further investigate the relationships between the real, scrambled, and generated weight matrices for $\mathcal{M}_1$, we employed t-SNE [7] embedding into $\mathbb{R}^2$ based on the spectral norm of the difference between each pair of weight matrices (including the bias term as the last column). t-SNE, or t-distributed Stochastic Neighbor Embedding, is a dimensionality reduction technique that helps visualize high-dimensional data in lower-dimensional spaces while preserving local structures. The spectral norm provides a measure of how close or distant two matrices are in terms of their induced metric, defined as:

$$\|\mathbf{W}_i - \mathbf{W}_j\|_2 \coloneqq \max_{\|\mathbf{x}\|_2=1} \|(\mathbf{W}_i - \mathbf{W}_j)\mathbf{x}\|_2 = \sigma_{\max}(\mathbf{W}_i - \mathbf{W}_j)$$

where $\sigma_{\max}$ denotes the largest singular value of the matrix difference. Our expectation was that the real weight matrices in $\mathcal{R}_1$ would cluster closely together, while the scrambled matrices would be dispersed. However, our findings revealed that both the real and scrambled matrices formed tight clusters, with the generated weights occupying a distinct cluster. This suggests that while the generated weights perform well on the classification task, they exhibit different underlying characteristics compared to the real weights. As for the scrambled matrices, despite their training on randomly assigned labels, their clustering

may be attributed to the fact that the underlying architecture and data distribution remain unchanged. This may indicates that the model can still learn certain features of the dataset, even when the labels are scrambled.

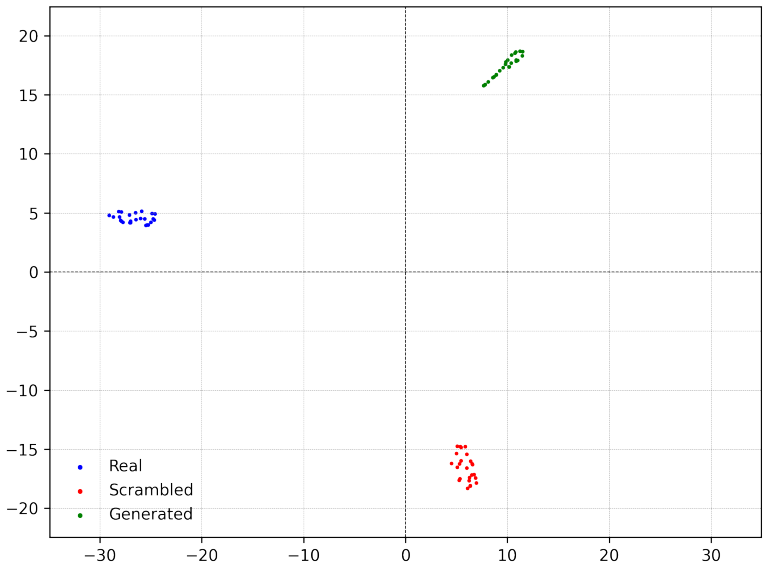

Figure 3: t-SNE embedding of $\mathcal{R}_1$, $\mathcal{S}_1$ and $\mathcal{G}_1$ into $\mathbb{R}^2$.

5 Conclusion

In this research, we have explored the feasibility of using a generative model to produce the weights for machine learning models without reliance on the original training data. By investigating Variational Autoencoder architectures tailored for single-layer and two-layer neural networks, we aimed to simplify the training process while maintaining or improving model performance.

Our findings indicate that while the generated weights differ from those of the real models based on the t-SNE analysis, they still achieve comparable performance when evaluated on the Iris dataset. The results highlight the potential of generative models in reducing the computational burden associated with hyperparameter tuning and architecture optimization. Moreover, the approach of utilizing latent representations demonstrates a promising avenue for future work in this domain.

In future work, we plan to investigate the robustness of the generated weights, particularly how they perform under varying conditions and in the presence of

missing data. This exploration will help us understand the reliability of our generative model in practical applications. Additionally, we aim to address scalability challenges associated with increasing model complexity. As we progress, we will focus on developing methods to ensure that the generative model maintains its performance across diverse architectures, datasets and tasks, ultimately enhancing its applicability in real-world scenarios.

To facilitate reproducibility and further exploration, the codebase for all experiments, including model implementations and data preprocessing, is publicly available at `https://github.com/shilo-a7x/Model-Generation`.

References

[1] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[2] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

[3] Diederik P Kingma. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[4] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[5] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[7] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.