Systematic characterization of generative models for *de novo* design of regulatory DNA

Nic Fishman [1] **Avanti Shrikumar** [1] **Georgi K. Marinov** [1] **Anshul Kundaje** [1]

Abstract

Generative machine learning algorithms have been developed for *de novo* generation of realistic images and videos. Recently, these approaches have also been adapted for designing and optimizing biomolecules with desired properties. However, systematic comparison of these methods for modeling regulatory DNA sequences have been lacking. Here, we present a taxonomy of generative learning algorithms and a unified implementation (https://github.com/kundajelab/seq_gen) that enables complementary classes of algorithms to be seamlessly combined in novel ways via a common API. We then systematically characterize the performance of these methods for *de novo* designing gene promoter elements to optimize gene expression in yeast. We identify biologically relevant stopping criteria as critically important for the generation of meaningful novel sequence elements, and propose leveraging a previously-introduced 1-nearest neighbor (1NN) approach for evaluating divergence from realistic elements. We also introduce a new generative model called a "supervised GAN", and evaluate its performance using the aforementioned metrics. Taken together, our results constitute a valuable stepping stone towards robust *de novo* design of regulatory DNA sequences.

1. Introduction

Generative deep learning models have revolutionized the ability to generate, enhance and manipulate realistic images, text and videos. They also allow learning latent representations that could be interpreted to understand the fundamental properties of different data modalities and the systems generating them. Recently, these approaches have shown great promise in various biological applications: single cell genomics, synthetic biology, protein engineering and antibody design. However, generative models have not been extensively studied in comparative frameworks or benchmarked on applications involving design and optimization of DNA sequences, particularly regulatory DNA sequences that modulate gene expression. In this work, we present a taxonomy for generative learning algorithms and models along with their implementations in a unified API that makes it easy for a user to mix-and-match complementary methods in novel combinations, as well as to allow comparing approaches systematically. We further evaluate these methods at the task of synthesizing regulatory DNA sequences with maximum expression activity using a biologically well-characterized yeast promoter massively parallel reporter assay (MPRA) dataset (van Dijk et al., 2017). By comparing the outputs to well-established biological ground truths, we find that continued training well past the point of divergence from the space of realistic sequences is a common failure mode for most methods. To mitigate against this problem, we propose leveraging the 1-nearest neighbor (1NN) approach (originally proposed by Xu et al. (2018) in the context of GANs) for detecting this divergence. We also introduce a generative model which we call a "supervised GAN", and evaluate its performance using the aforementioned metrics.

2. A Taxonomy of Generative Approaches

In this section, we develop a taxonomy to systematically characterize existing generative deep learning approaches. Forming the basis of all current deep generative models is a neural network architecture that learns a transformation $\Phi : Z \to X'$, where Z is a latent space and X' are generated instances of a data modality.In our case, these instances would be DNA sequences of some arbitrary length. We refer to this formulation as the "transducer". This core formulation can be extended by methods that take a transducer Φ and alter the transformation in such a way that the sequences produced are more likely to maximize properties of interest that are evaluated by a separate "oracle"/"analyzer", where the oracle/analyzer is often a predictive model. We refer to methods that follow this formulation as "transducer tuners". Finally, there are methods that hold Φ constant and search for locations in the latent space Z that maximize a property predicted by an oracle/analyzer; we refer to these as "sample

*Equal contribution [1]Stanford University. Correspondence to: Nic Fishman <njwfish@stanford.edu>, Anshul Kundaje <akundaje@stanford.edu>.

optimizers". Although generally treated separately in the literature, these classes of models are all complementary.

Transducers: Transducers include Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) and Variational Autoencoders (VAEs) (Kingma & Welling, 2013). Although both GANs and VAEs learn latent spaces that keep "similar" examples closer, the latent space can be hard to optimize over. To structure the latent space based on a property of interest, an additional network predicting the property can be added to a VAE encoder/decoder. This approach has been successfully used in the context of chemical graphs (Gómez-Bombarelli et al., 2018), and we refer to it as a "supervised VAE" or **supVAE**. Inspired by it, we propose here a novel extension of supVAE to GANs as follows: we predict properties from the latent space, but these predictions cannot be evaluated (precisely because they are for generated samples) so we then feed both the generated sequence and its "predicted" property (e.g. expression associated with a regulatory sequence) through the discriminator, updating both the generator and the analyzer according to the resultant loss. Here, the discriminator is learning the relationships between examples and the predicted property, restricting the latent space to keep similar expression sequences close together in the latent space so as to keep the prediction model accurate. We refer to this architecture for a GAN as a "supervised GAN" or **supGAN**. The four different transducer architectures (GAN, VAE, supGAN and supVAE) are explained in more detail in **Sec. A.3** and summarized in **Figure S1**.

Transducer Tuners: The common underlying idea behind transducer tuning is to sample or upweight "good" sequences generated by the transducer, and to continue transducer training using those sequences. Over the course of training, the transducer is thus encouraged to generate only sequences that have the desirable properties. Several such methods have been presented, mostly in the context of designing coding DNA sequences. In this work, we benchmark FBGAN (Gupta & Zou, 2019) as well as each method in the "CbAS family" (Brookes et al., 2019) (CEMPI, RWR, DbAS and CbAS). These methods, as well as implementations, are described in more detail in **Sec. A.4**. Note that because CbAS requires a probability estimate that is not naturally provided by GANs, we did not include the combination of GANs and CbAS in our summaries even though a potential GAN-compatible implementation is available in our codebase (see **Sec. A.4** for details).

Sample Optimizers: Sample optimizers attempt to find positions in the latent space corresponding to "good" examples. The simplest such technique is sampling randomly from the underlying distribution and calling the top $X\%$ the "optimized" sample. (Killoran et al., 2017) improved on the random sampling by using a gradient descent approach (Nguyen et al., 2017) to directly optimize samples in the latent space. In a similar vein, Gomez *et al.* proposed training a Gaussian Process Regressor to predict a property from the latent space and then using constrained optimization by linear approximation to iteratively optimize the latent space without direct use of a derivative (Gómez-Bombarelli et al., 2018).

3. Results and Methods

We set out to apply generative methods in the context of designing gene promoter elements in yeast. We used a previously published MPRA dataset (van Dijk et al., 2017), which contains ∼5,000 synthetic promoter sequences containing pre-defined transcription factor binding sites (TFBS), in particular for the GCN4 TF, in a variety of configurations. This dataset allows us to compare *de novo* generated sequences against known biological ground truths. This is a key advantage with respect to evaluating generative methods, as most work in the field so far has focused on generating protein coding sequences, a domain in which it is much more difficult to evaluate generated sequences.

We first tested the ability of multiple convolutional neural networks (CNN) architectures to predict MPRA activity from DNA sequence. We observed high concordance between predicted and experimentally measured activities (Test Set Spearman Correlation: 0.93; Figure S2), including the successful capture of the main biologically novel result of the original publication, namely that the presence of too many TFBSs (GCN4 sites in this case) reduces rather than increases expression (due to steric competition between the neighboring sites for TF molecules).

We then used this CNN architecture as a basis for training generative models. In order to evaluate the ability of generative models to maximize expression, i.e. to generate realistic sequences "out of distribution", we trained models to predict expression from sequence using both the bottom 95% of the dataset and also using the full dataset. Similar to (Brookes et al., 2019), for each input set, we trained two ensembles of 20 networks, composed of residual CNNs (Supplemental Methods; the two ensembles provide an estimate of uncertainty). Models trained on the bottom 95% give broadly similar predictions for the top 5% of sequences compared to models trained using the full dataset (the Spearman correlation between the model predictions on the top 5% of sequences is 0.7263; Figure S3). The main difference between the predictions is that those from the model trained on the bottom 95% tend to be slightly lower than the ground-truth for the top 5% of sequences (as one might expect).

We then trained a GAN, VAE, supGAN, and a supVAE to generate new DNA sequences. For the GANs, we used the Wasserstein distance with a gradient penalty as proposed

in (Gulrajani et al., 2017). All methods were trained on the bottom 95% of sequences and used essentially the same underlying architectures, with minor alterations (**Sec. A.3**). We used a similar architecture for the decoder in the VAE as for the generator in the GAN, and a similar architecture for the encoder in the VAE as for the discriminator in the GAN. In both cases, the decoder/generator and encoder/discriminator involved a series of convolutional layers. For the supVAE and supGAN, we use three fully connected layers as the predictor.

Contrary to previous findings in the coding sequence domain (Brookes & Listgarten, 2018), we find significant differences between GAN and VAE results. GANs perform better in terms of capturing the diversity and complexity of the training set space (Figures 1, S5–S9, S10, and S21). We also find that supVAEs do not achieve significantly different results compared to VAEs. The supGAN covers a wider range of the input distribution compared to the GAN, but simultaneously suffers minor mode collapse (where some examples are very close to each other) as shown in **Fig. S28**.

We then generated sequences using each of the four generative strategies combined with each of the transducer tuning methods. Examination of the final output (Figure S10) and the behavior throughout training (Figures S16-S27) revealed some common problems. Methods quickly diverge from the "realistic" distribution as they optimize the property of interest, eventually arriving into the space of "pathological" sequences, where examples generated obviously make little biological sense. Such undesired behaviors included highly skewed nucleotide composition (e.g. sequences consisting almost entirely of polyTs and other homopolymers) and sequences containing either zero or too many TFBSs.

Additionally, for each combination of transducer/tuning method, we applied top percentile and gradient descent sample optimizer methods in the latent space (Figures 2B and S34-S35). The top percentile method appears to perform remarkably well while maintaining a good balance of "realisticness", and it also outperforms the gradient descent approach.

While biological evaluation is highly informative for judging the quality of generated sequences, it does not provide a straightforward way of automatically evaluating convergence or divergence for a given generative method. Nearest neighbor algorithms (usually the 1NN algorithm, see Table S1) offer an alternative solution. The 1NN evaluation (originally proposed for GANs by Xu et al. (2018)) works as follows: a collection of synthetic sequences is generated to be equal in size to a set of available real sequences (for our comparisons, we used 2,024 sequences). Each sequence is then classified as either 'real' or 'fake' according to whether its nearest neighbor (excluding the sequence itself) is from the generated set or from the real set, where the nearest-neighbors are found by embedding the sequences into an informative latent space and calculating euclidean distances. If the 1-nearest-neighbor algorithm achieves high accuracy, this indicates that the generated sequences are very distinct from the real sequences; by contrast, an accuracy near 50% indicates that the generative method is approximating the underlying distribution well. The classification accuracy captures how closely the generated samples resemble the true distribution and also the diversity of the generated sequences: "mode collapse" of the generated sequences would result in high classification accuracy on the real sequences, as all the generated sequences would occupy a narrow region of the space. Key to the performance of the 1NN algorithm is the choice of an informative embedding; in our case, we used the activations of the fifth convolutional layer of the CNN model as described in **Fig. 1**. We also explored k larger than 1 to gain additional insight into the distributions.

The decrease in nearest-neighbor accuracy over the course of training for generative architectures is show in Figures 2A & S28). We observe that tuning methods relatively quickly diverge from realistic sequences (Figures 2A and S29–S33), which is concordant with developing deviations from realistic nucleotide and TFBS composition in the same time (Figures 2C and S11–S22). However, predicted expression increases with training. This suggests that current methods are optimizing the analyzer network rather than true biological activity, essentially leading to adversarial examples in the DNA domain. However, the drastic divergence from the space of realistic sequences does not occur immediately, but after some number of early epochs have elapsed, and the 1NN criterion (Xu et al., 2018), which is generally concordant with biological criteria, can thus be used to identify better stopping points.

4. Discussion

Recent years have seen a flurry of diverse methods for *de novo* generation of biological sequences. In this work, we synthesize existing methods into three broad classes: (1) *transducers*, which transform a latent embedding into a generated sequence, (2) *transducer tuners*, which optimize a transducer to produce desirable sequences, and (3) *sample optimizers*, which do a search in the latent embedding to find desirable sequences. Bolstered by the observation that these three types of methods address complementary pieces of the sequence generation problem, we implement the methods in a single codebase with a unified API that allows the user to seamlessly combine their preferred choice of *transducer*, *transducer tuner* and *sample optimizer*. We also propose a novel type of *transducer* architecture that we call a *supervised* GAN, and experimentally benchmark the performance of different approaches using a well-characterized yeast MPRA dataset.

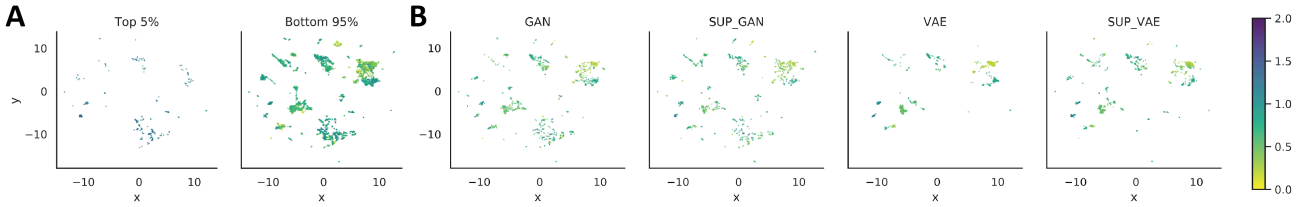

Figure 1. **GANs capture the diversity of real sequences better than VAEs**. Sequences were generated using four methods: GAN, supGAN, VAE and supVAE. A common 2D projection for the sequences was created as follows: all generated and real sequences were scanned with the first five convolutional layers of the DeepSEA network(Zhou & Troyanskaya, 2015), and the output of the last convolutional layer was flattened to derive a sequence embedding. This embedding was projected into 2D space using UMAP. A single UMAP projection was used for all sequences so that different subsets of sequences could be compared. Panel (A) contains the 2D projections for the top 5% and the bottom 95% of the real data, while Panel (B) shows the projections for sequences sampled from the respective generative methods. The color represents the *log* of the predicted expression. GAN-based methods appear to generate more diverse sequences, better approximating the input space.

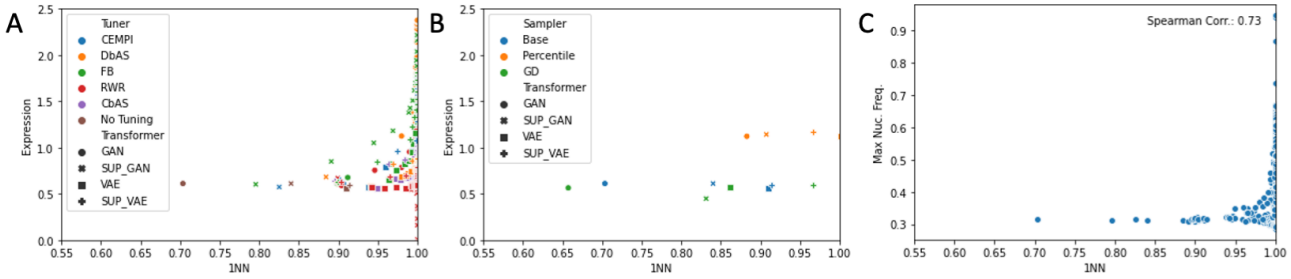

Figure 2. **Performance of Tranformer Tuners and Sample Optimizers based on expression and 1NN metric**. Panels (a) and (b) show the average log-expression (y-axis) achieved by transducer tuners and sample optimizers respectively against the 1NN accuracy (x-axis). High 1NN accuracy indicates that generated sequences diverge substantially from real sequences. Symbol shapes represent the choice of transducer (GAN, supGAN, VAE or supVAE), while colors indicate the choice of transducer tuner (A) or sample optimizer (B). Panel A shows that as transducer tuners achieve high average log-expression over the course of training, the 1NN accuracy also tends to increase¿ The high expression thus comes at the expense of similarity to realistic sequences. There are multiple points for each transducer/transducer tuner combination in (A), corresponding to every five epochs of tuning. Brown points in panel A show the performance of each transducer prior to any tuning or sample optimizing. In (B), the gradient descent (GD) sample optimizer does not noticeably improve the average *log* expression, while the "top percentile" sample optimizer combined with a GAN or supGAN is able to achieve distinct improvements in expression while maintaining <90% 1-NN accuracy. Panel C shows the concordance of the 1-NN metric with the maximum nucleotide frequency over A,C,G,T; each point corresponds to a sequence set generated by transducer tuners in (A). Maximum nucleotide frequency much larger than 0.25 suggests the sequences contain many homopolymer repeats, and is thus biologically unrealistic. Sequence sets that have high maximum nucleotide frequency also achieve high 1NN accuracy, indicating that the 1NN metric agrees with this biologically intuitive measure of unrealistic sequences.

Among the *transducers*, we find that GANs work better at capturing the diversity of sequences compared to VAEs, with our proposed supervised GAN showing advantages for sequence diversity compared to the standard GAN. Among *sequence optimizers*, we find that the naive "top X percentile" method appears to produce superior results compared to the more complex gradient descent method. Among *transducer tuners*, we observe that *all* methods investigated generate essentially adversarial examples when training proceeds for a sufficiently long time - in other words, the sequences diverge so far from the training distribution as to be biologically meaningless, despite scoring favorably according to the analyzer/oracle model. This is because sequence generation sits at the intersection of two well-known challenges in machine learning: adversarial examples and out-of-training-distribution generalization. To address this issue, we propose using the 1NN metric (Xu et al., 2018), which, based on our results, appears to agree with biologically-informed measures of sequence quality and can help strike a proper balance between optimization of the target property and the generation of realistic sequences. This metric can be implemented in a way analogous to *early stopping with a validation set*, a standard practice in supervised learning. Taken together, we believe our contributions are valuable step towards robust design of *de novo* regulatory DNA sequence.

References

Arjovsky, M., Chintala, S., and Bottou, L. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

Brookes, D. H. and Listgarten, J. Design by adaptive sampling. *arXiv preprint arXiv:1810.03714*, 2018.

Brookes, D. H., Park, H., and Listgarten, J. Conditioning by adaptive sampling for robust design. *arXiv preprint arXiv:1901.10060*, 2019.

Gómez-Bombarelli, R., Wei, J. N., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., and Aspuru-Guzik, A. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680, 2014.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. C. Improved training of wasserstein gans. In *Advances in neural information processing systems*, pp. 5767–5777, 2017.

Gupta, A. and Zou, J. Feedback gan for dna optimizes protein functions. *Nature Machine Intelligence*, 1(2):105, 2019.

Killoran, N., Lee, L. J., Delong, A., Duvenaud, D., and Frey, B. J. Generating and designing dna with deep generative models. *arXiv preprint arXiv:1712.06148*, 2017.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Nguyen, A., Clune, J., Bengio, Y., Dosovitskiy, A., and Yosinski, J. Plug & play generative networks: Conditional iterative generation of images in latent space, 2017.

van Dijk, D., Sharon, E., Lotan-Pompan, M., Weinberger, A., Segal, E., and Carey, L. B. Large-scale mapping of gene regulatory logic reveals context-dependent repression by transcriptional activators. *Genome research*, 27 (1):87–94, 2017.

Xu, Q., Huang, G., Yuan, Y., Guo, C., Sun, Y., Wu, F., and Weinberger, K. An empirical study on evaluation metrics of generative adversarial networks. June 2018.

Zhou, J. and Troyanskaya, O. G. Predicting effects of noncoding variants with deep learning–based sequence model. *Nature methods*, 12(10):931, 2015.

Supplementary Materials

A. Supplementary Methods

A.1. MPRA Datasets and data preprocessing

We used the MPRA experiments carried out by van Dijk et al. (2017) in the budding yeast *Saccharomyces cerevisiae*, which included testing the activity of ∼5,000 synthetic sequences containing defined numbers of TFBSs for factors known to be important in regulating gene expression upon changes in nutrient availability. Regulatory activity was measured under a range of six different increasing amino acid concentrations.

DNA sequences were one-hot encoded following established practices. We also $log+$-transformed the regression targets so we could use the ReLU non-linearity as the output of our regression networks.

A.2. Training Regression Ensembles for the Analyzer/Oracle

Models are trained directly on one hot encoded base pairs, such that each model takes in an input matrix of size $(L * 4)$, where L is the length of the sequence, and 4 corresponds to the four DNA bases. Each individual model is built of two convolutional residual blocks, each comprised of two convolutional layers of 100 filters each. These layers lead into a series of three fully connected layers, each with dropout (0.1). All layers use the ReLU activation function. Each network produces two outputs for each "task" that the model is being trained on (e.g. for multitask learning, if there are n tasks, then the model has $2n$ outputs). These outputs define a normal distribution on the expected expression level for a given task, with the first output being the mean and the second being the variance. For each ensemble, sub-networks are trained in parallel, using the Adam optimizer (`lr = 0.001`) with the original hyperparameters, to minimize a Gaussian negative log-likelihood loss function. We use a batch size of 64, with a maximum number of epochs of 1000, but use the Keras callbacks Reduce LR on Plateau (with default parameters except `factor = 0.1` and `patience = 3`) and Early Stopping (with `patience = 5`) such that no model actually trains for 1000 epochs (convergence tends to be much more rapid, on the order of 100 epochs at most).

Twenty such networks are trained and then assembled into an ensemble with outputs μ^*, σ^{2*}:

$$\mu^* = \frac{1}{20}\sum_{i}^{20}\mu_i$$

$$\sigma^{2*} = \frac{1}{20}(\sum_{i}^{20}\sigma_i^2 + \sum_{i}^{20}\mu_i^2) - (\frac{1}{20}\sum_{i}^{20}\mu_i)^2$$

We train two such ensembles, using just the bottom 95% of data and one using the full dataset.

A.3. Generative Methods (i.e. "Transducers")

Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) involve a generator that transforms the latent space into the target distribution, and a discriminator that attempts to discriminate real and generated sequences. The generator and discriminator are trained in tandem to allow progressive improvement of both models. A modification to the GAN framework directly optimizes the Wasserstein distance between the latent and target distributions (Arjovsky et al., 2017), greatly improving the stability of the GAN. In this work, we use a Wasserstein GAN with a gradient penalty addition that improves the capability of the GAN to learn more complex distributions (compared to simple weight-clipping to satisfy the Lipschitz constraint) (Gulrajani et al., 2017).

By contrast, Variational Autoencoders (VAEs) (Kingma & Welling, 2013) create encoder and decoder models that translate sequence into and out of a latent space representation. The latent space is regularized by the Kullback-Liebler divergence to ensure that it is normally distributed. This is done via a "reparameterization trick", where the encoder predicts the mean and variance of the latent space, and the actual value of the latent representation is sampled according to this mean and variance. This stochasticity forces the latent space representation to locate "similar" examples close to each other in that space.

In our experiments, all generative methods were implemented to use approximately the same underlying architectures,

and these architectures are roughly symmetrical. All convolutional layers have 100 filters, and window size 5. All latent dimensions are 500 hidden units. All neural network layers use the ReLU activation function unless otherwise stated.

The generator/decoder architecture consists of a dense layer with $(L * 100)$ units, leading into a series of five convolutional residual blocks (each consisting of two convolutional layers). The last layer in the architecture is a convolutional layer with 4 filters, so that the shape of the output is the expected shape of the output sequence. In the GAN setting, the activation function for this layer is the Gumbel-Softmax layer; in the VAE setting, this is a simple softmax activation function instead.

The discriminator/encoder architecture is nearly identical, but inverted. The first layer is a convolutional layer to reshape the input leading into the same five convolutional residual block structure. The final output of the fifth block is flattened and feeds into either a single output in the GAN case (the discriminator output) or two outputs in the VAE case (the mean and log variance).

A.3.1. Wasserstein Generative Adversarial Network with Gradient Penalty

We train the GAN using the Wasserstein loss function with the gradient penalty ($\lambda = 10$) approach, as is now standard in the literature. We train the generator on one batch for every 5 the discriminator is trained on. We use the Adam optimizer with default hyperparameters for optimizing both the discriminator and the generator.

The supervised GAN includes a minor modification. In this setting, an additional network is trained that attempts to predict expression from the location in the latent space passed to the generator. The discriminator then takes in a pair of (sequence, expression value(s)), both for the real and generated sequences. The discriminator architecture is modified by concatenating the expression to the flattened output of the last residual block, and adding three dense layers (each of 100 hidden units) before the output. The only modification to the loss function that needs to be applied is adding the expression values to the gradient penalty. This is achieved by replicating the gradient penalty calculation for the generated and real expression values in a batch and then taking the mean. The expression-prediction network is trained in conjunction with the generator (every 5 batches).

A.3.2. Variational Autoencoder

The variational autoencoder is trained using the standard VAE reconstruction loss function, where each position in the sequence is modeled as a softmax over nucleotides. The Adam optimizer with default hyperparameters is used to minimize the loss. The supervised VAE is trained identically, with the addition of one term to the loss function. The MSE loss of the true expression value of the sequence to be reconstructed and the expression predicted from the latent space is added to the general reconstruction loss.

A.4. Transducer Tuners

All transducer tuning methods employ a pretrained *transducer* (i.e. a generative architecture), on which additional training is layered in order to guide the transducer to produce desirable sequences. We can formalize the notion of a tuner a bit here: we have the real sequences and property data $\{X_{real}, Y_{real}\}$, a trained transducer from which we can sample X_{gen} sequences, and an oracle/analyzer (which need not be differentiable for tuners considered here) to predict $\hat{Y}_{gen}$ from the generated sequences. A tuner is then uniquely defined by a weighting function $w(\{X_{real}, X_{gen}\}, \{Y_{real}, \hat{Y}_{gen}\})$ and a schedule, which dictates when X_{gen} is updated.[1] In this work, we consider five types of transducer tuners: FBGAN (Gupta & Zou, 2019), and the four methods in the "CbAS family" (Brookes et al., 2019), i.e. CEMPI, RWR, DbAS and CbAS.

FBGAN (and the VAE equivalent we implemented) works by iteratively shifting the underlying data distribution that the transducer (i.e. the generator) is trained on in a way that increasingly prefers samples with desirable properties. FBGAN begins by training on the full training set. At each iteration, sequences are sampled from the generator, and the top $p\%$ of sequences are used to replace the oldest sequences in the training set, such that at every iteration a greater fraction of the sequences being trained on are synthetic sequences with desirable properties. The choice of p matters quite a bit; following the original paper we use an 80% threshold, and sample 15/0.2 samples per iteration such that 15 samples are added to the training set at each iteration. We can re-conceptualize FBGAN as a weighting scheme where each a new X_{gen} is sampled each epoch and weights of 1 are assigned to the to a given sampled sequence if it is in the top $p\%$.

The suite of methods developed by (Brookes et al., 2019), which includes CEMPI, RWR, DbAS and CbAS, follow the

[1] The formalization here indicates a superficial similarity to boosting methods, which may be an interesting line of future work.

schema laid out for tuners above, with a schedule where a new X_{gen} is sampled every $t = 10$ epochs of transducer training; the methods differ in the details of the weighting function, although all weighting functions work exclusively on X_{gen} unlike FBGAN which assigns non-zero weights to some real data as well. CEMPI ("Cross Entropy Maximization Pi") draws on cross-entropy methods to condition the generator to focus on rarer samples that have the property of interest. RWR (Reward Weighted Regression) is a reinforcement learning method that reweighs sequences exponentially. Design by Adaptive Sampling (DbAS) and Conditioning by Adaptive Sampling (CbAS) reweigh samples according to the survival function of the normal distribution, and differ in that CbAS uses a prior to avoid diverging too much from the underlying distribution.

All methods were trained on 500 sampled/weighted examples for 10 epochs before resampling. Each tuning method is run for 800 iterations in total, 80 epochs because the 10 iterations over each sample are considered one epoch of tuner training. While most of these methods are "generator agnostic", i.e. the underlying generator does not matter, we found that this is not true for the CbAS method. CbAS is designed to work with probability distributions over nucleotides. However, we find that our generators collapse to producing one-hot encoded sequence very rapidly, which results in NaN errors within the CbAS framework as it corresponds to an infinite penalty. In order to adapt CbAS to circumvent this limitation, we add an epsilon noise term ($\epsilon = 10^{-8}$) to the one-hot encoded sequence. A second issue with porting CbAS to GANs is that the CbAS method involves a step where the prior probability of observing a sequence is calculated by first encoding the sequence to obtain a latent state and then computing the probability of observing the resulting latent state if one were to draw from the VAE's normally distributed prior. This prior is used to prevent excess divergence from the original training distribution. In order to use CbAS in the GAN context, some analogous notion of a prior is needed. As GANs do not train an encoder, one could in principle leverage the discriminator and map the discriminator output to the $[0, 1]$ interval using a sigmoid function to obtain an analogous probability. Note that the output of the discriminator in a Wasserstein GAN is the Wasserstein distance, which is negative if the sequence appears fake and positive if the sequence appears real; when transformed by a sigmoid, this means the probability will be > 0.5 if the discriminator believes a sequence is likely to be real, and < 0.5 if the discriminator believes a sequence is likely to be fake. That said, it is likely that this is not the optimal way of extending CbAS to the GAN setting (e.g. there is no guarantee the sigmoid would be calibrated), and so we did not include comparisons involving CbAS and GANs in our figures even though this functionality exists in our codebase.

A.5. Nearest Neighbor Algorithms and Evaluation

For 1NN evaluation we draw our independent latent space from the DeepSEA network (Zhou & Troyanskaya, 2015) using the first 5 convolutional layers as a feature space. We then implement a simple 1NN algorithm in this latent space to generate the LOO accuracy of a set of generated and real examples.

One key question is whether to use a 1NN or some other value of k in the KNN algorithm. We explore this in Table 2. The value of n is by far the most important factor in the evaluation outcome, with larger n leading to a less sparse "neighbor space", lower variance, and a more accurate evaluation of the generated sequences. The insight that larger n lead to more consistent evaluation is then helpful in evaluating various values of k. Different values of k have at most marginal impact on the evaluation variance. But different k's do have significant impact on the actual evaluation scores. In general, a larger k tends to estimate a better score (closer to 0.5). As n increases though, so does the score, meaning that for low n the 1NN algorithm better approximates the higher n than other values of k. Extrapolating this result, for high n, the 1NN approach is likely still preferable. Hence throughout we use a 1NN with a large sample size, $n = 2024$, to maximize evaluation accuracy and minimize variance.

B. Supplementary Tables

	n	$k=1$	$k=3$	$k=5$	$k=7$	$k=9$
μ	1012	0.68532609	0.67168972	0.67089921	0.66818182	0.6625
σ		0..01165097	0.00964452	0.01088628	0.01195448	0.01182978
μ	256	0.63105469	0.62363281	0.61191406	0.59707031	0.58574219
σ		0.02816092	0.01677983	0.01729481	0.03077086	0.03251176
μ	64	0.584375	0.565625	0.5625	0.54296875	0.540625
σ		0.06553193	0.05289472	0.05390059	0.05952387	0.05203383

Table 1. Results of the nearest neighbor algorithm on the 4000th epoch of GAN training for various k. Each n represents the size of the sample drawn from the real and generated distributions. 10 independent samples were drawn from each distribution and the knn evaluation for various k performed for each sampling. The μ is then the average of these evaluations, the σ the standard deviation.

	n	$k=1$	$k=3$	$k=5$	$k=7$	$k=9$
μ	1012	0.68532609	0.67168972	0.67089921	0.66818182	0.6625
σ		0..01165097	0.00964452	0.01088628	0.01195448	0.01182978
μ	256	0.63105469	0.62363281	0.61191406	0.59707031	0.58574219
σ		0.02816092	0.01677983	0.01729481	0.03077086	0.03251176
μ	64	0.584375	0.565625	0.5625	0.54296875	0.540625
σ		0.06553193	0.05289472	0.05390059	0.05952387	0.05203383

Table 2. Results of the nearest neighbor algorithm on the 4000th epoch of GAN training for various k. Each n represents the size of the sample drawn from the real and generated distributions. 10 independent samples were drawn from each distribution and the KNN evaluation for various k performed for each sampling. The μ is then the average of these evaluations, the σ the standard deviation.

C. Supplementary Figures

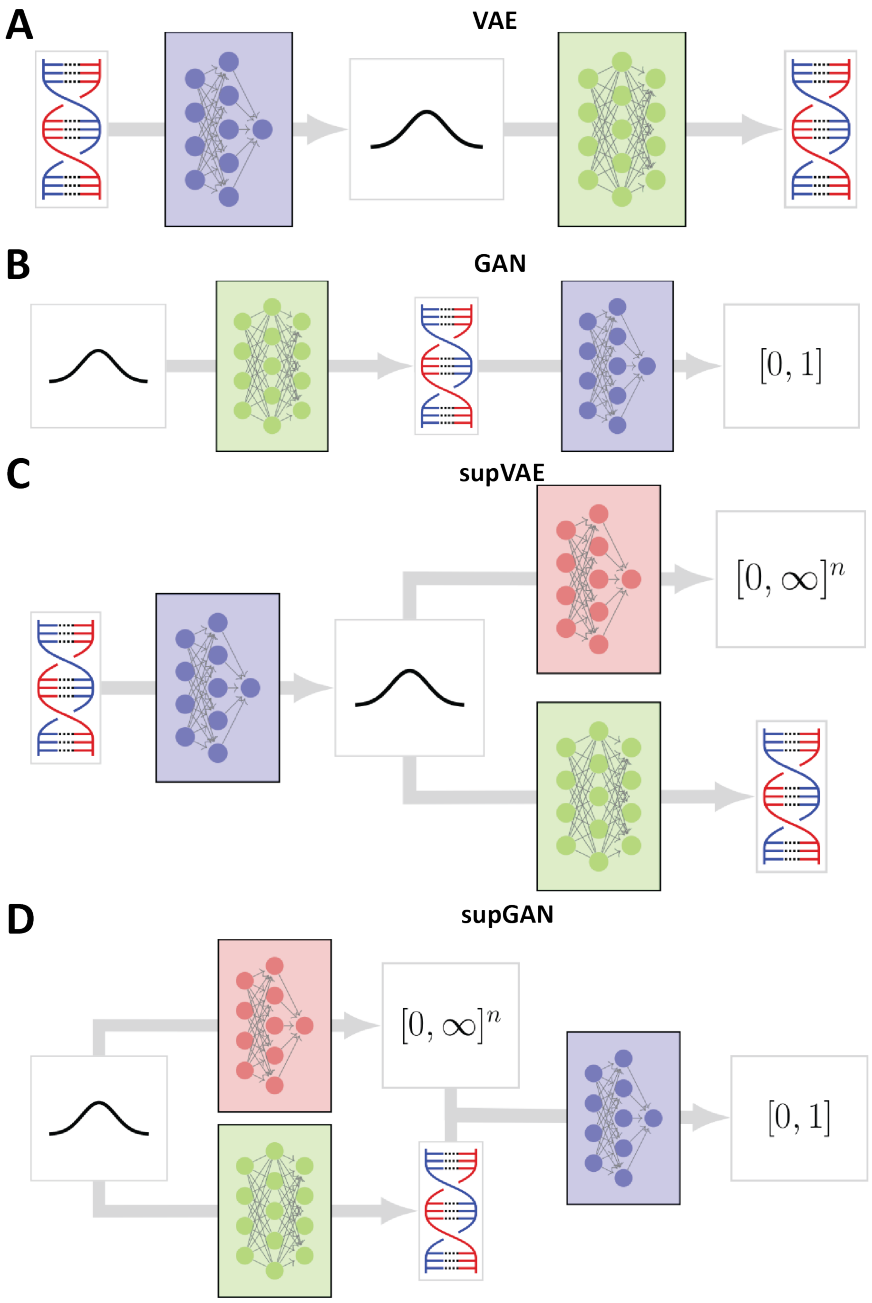

Figure 1. **Overview of sequence generation approaches**. (A) Variation AutoEncoders (VAE); (B) Generative Adversarial Networks (GAN(); (C) Supervised Variation AutoEncoders (supVAE); (D) Supervised Generative Adversarial Networks (supGAN);

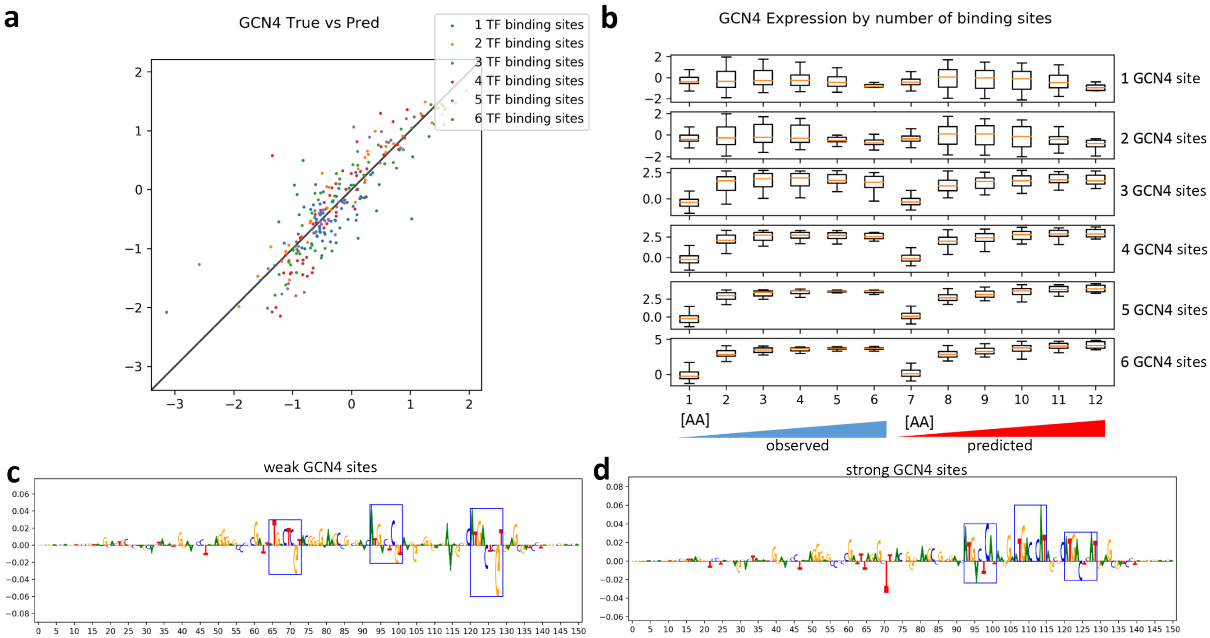

Figure 2. **Performance of predictive models on yeast MPRA dataset**. (A) True (x-axis) and predicted (y-axis) expression levels for constructs with different numbers of GNC4 motifs. (B) Deep learning models reproduce experimentally the observed behaviors of constructs with different numbers of GNC4 motifs in response to increasing [AA]. (C-D) DeepLIFT importance score for a construct containing three weak GCN4 TFBSs (c) and another one containing three strong GCN4 TFBSs correctly identify motifs driving regulatory activity

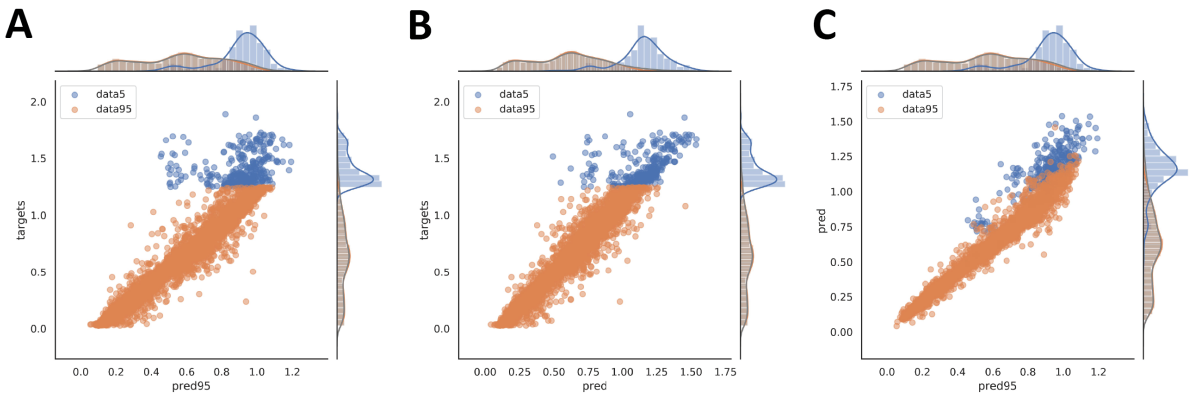

Figure 3. **Performance of predictive models trained on the bottom 95% and on the full yeast MPRA dataset**. (A) True (y-axis) vs predicted from the bottom 95% sequences (x-axis) MPRA activity levels. (B) True (y-axis) vs predicted from the full set of sequences (x-axis) MPRA activity levels. (C) Predicted from the bottom 95% sequences (x-axis) vs predicted from the full set of sequences (y-axis) MPRA activity levels.

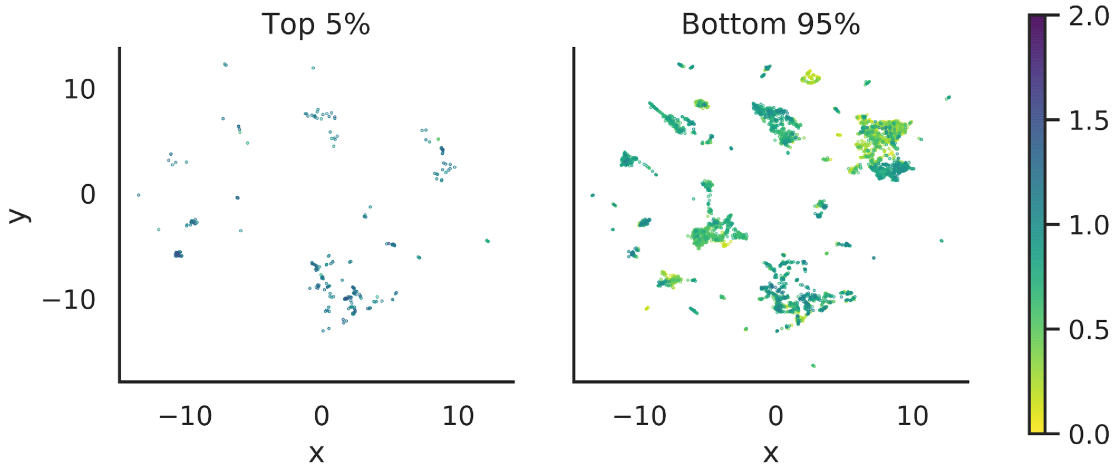

Figure 4. **Distribution of the top 5% and the bottom 95% sequence in the latent space projection**.

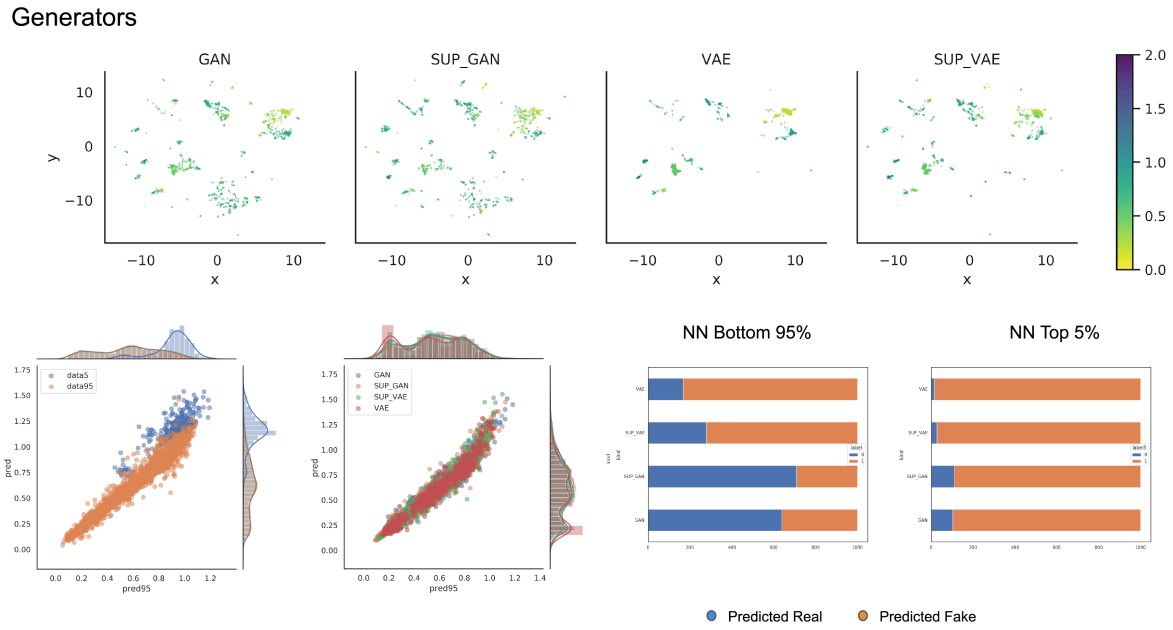

Figure 5. **Overview of performance of general generative methods on the task of generating active yeast promoter sequences**. The top panel shows a UMAP projection to two dimensions of the latent space projection of the sampled sequences of each method. The UMAP projection is held constant so that methods can be meaningfully compared. The bottom left two panels show the predicted expression distribution of the analyzer trained using the bottom 95% of sequence (which was optimized for tuning methods) and the analyzer trained using the full dataset. The left most plot shows the predicted values for the real data, while the plot to the right shows the predicted values for each method. The two bottom right panels shows the fraction of sequences that are predicted to be "real" or not by the 1NN algorithm for each method. The left one displays the portion of predicted "real" when using the bottom 95% as the "real" data, while the right one displays the same predictions but using only the top 5% as the "real" sequences.

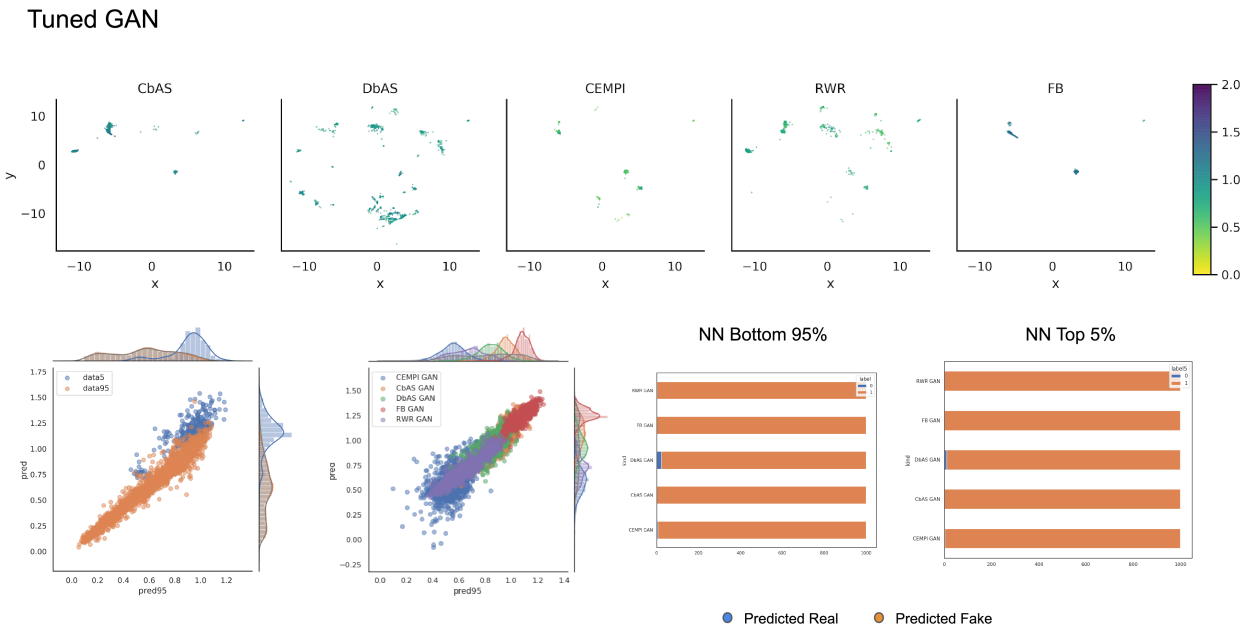

Figure 6. **Overview of performance of tuning methods applied to GANs on the task of generating active yeast promoter sequences**. The top panel shows a UMAP projection to two dimensions of the latent space projection of the sampled sequences of each method. The UMAP projection is held constant so that methods can be meaningfully compared. The bottom left two panels show the predicted expression distribution of the analyzer trained using the bottom 95% of sequence (which was optimized for tuning methods) and the analyzer trained using the full dataset. The left most plot shows the predicted values for the real data, while the plot to the right shows the predicted values for each method. The two bottom right panels shows the fraction of sequences that are predicted to be "real" or not by the 1NN algorithm for each method. The left one displays the portion of predicted "real" when using the bottom 95% as the "real" data, while the right one displays the same predictions but using only the top 5% as the "real" sequences.

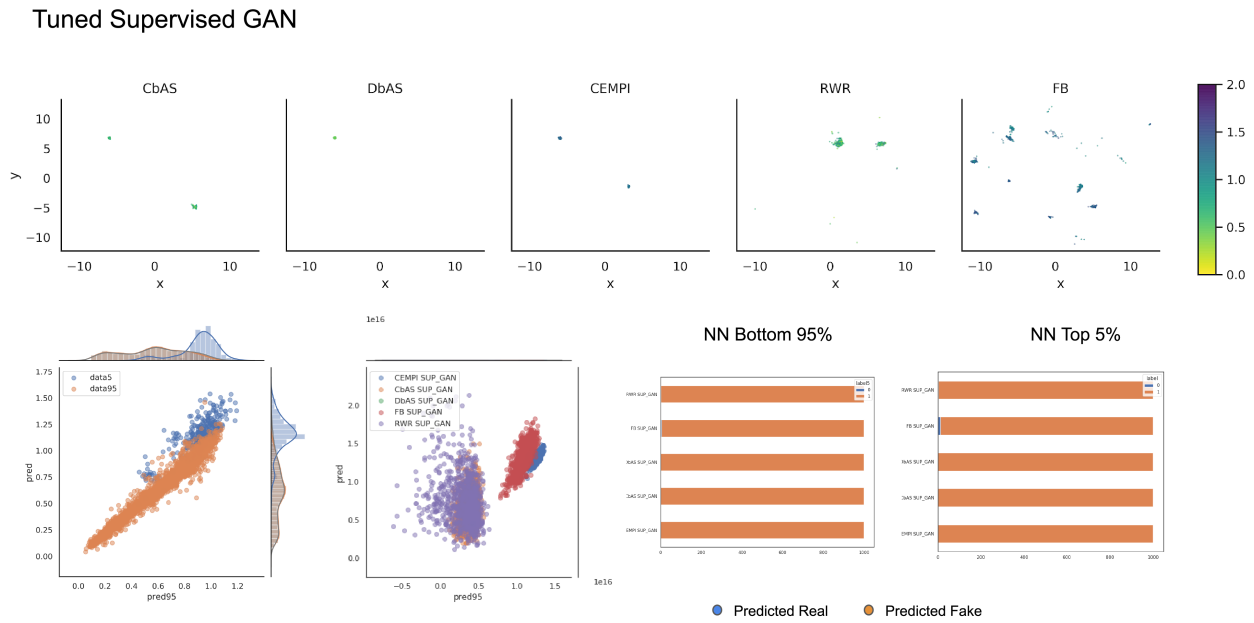

Figure 7. **Overview of performance of tuning methods applied to supervised GANs on the task of generating active yeast promoter sequences**. The top panel shows a UMAP projection to two dimensions of the latent space projection of the sampled sequences of each method. The UMAP projection is held constant so that methods can be meaningfully compared. The bottom left two panels show the predicted expression distribution of the analyzer trained using the bottom 95% of sequence (which was optimized for tuning methods) and the analyzer trained using the full dataset. The left most plot shows the predicted values for the real data, while the plot to the right shows the predicted values for each method. The two bottom right panels shows the fraction of sequences that are predicted to be "real" or not by the 1NN algorithm for each method. The left one displays the portion of predicted "real" when using the bottom 95% as the "real" data, while the right one displays the same predictions but using only the top 5% as the "real" sequences.

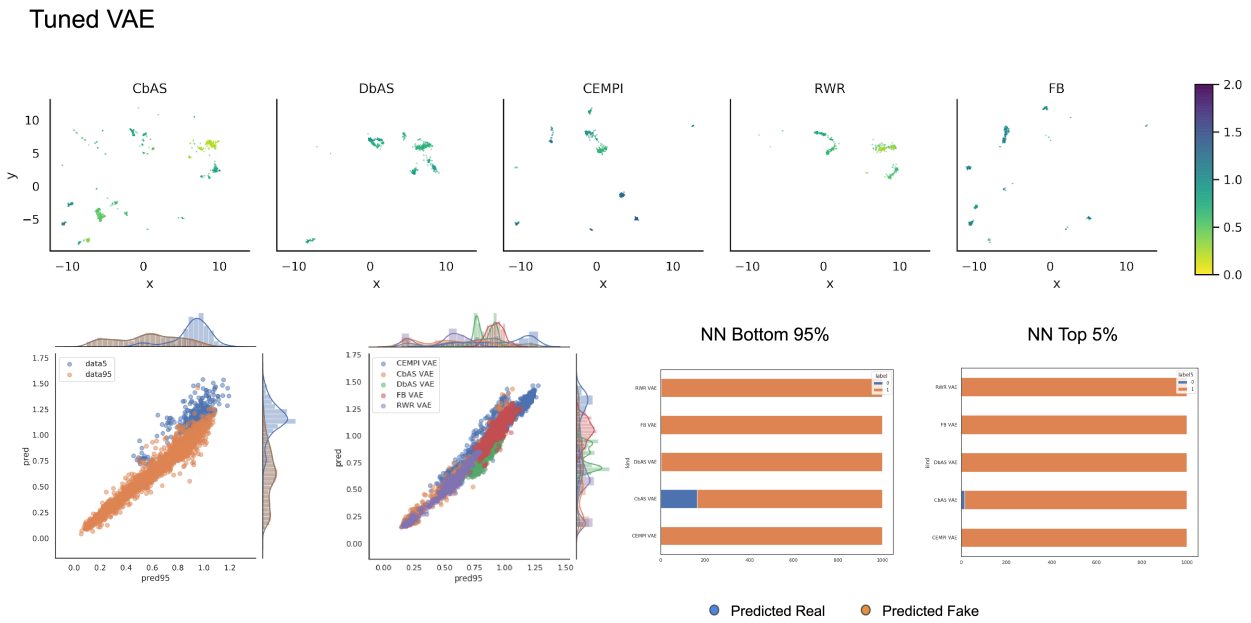

Figure 8. **Overview of performance of tuning methods applied to VAEs on the task of generating active yeast promoter sequences**. The top panel shows a UMAP projection to two dimensions of the latent space projection of the sampled sequences of each method. The UMAP projection is held constant so that methods can be meaningfully compared. The bottom left two panels show the predicted expression distribution of the analyzer trained using the bottom 95% of sequence (which was optimized for tuning methods) and the analyzer trained using the full dataset. The left most plot shows the predicted values for the real data, while the plot to the right shows the predicted values for each method. The two bottom right panels shows the fraction of sequences that are predicted to be "real" or not by the 1NN algorithm for each method. The left one displays the portion of predicted "real" when using the bottom 95% as the "real" data, while the right one displays the same predictions but using only the top 5% as the "real" sequences.

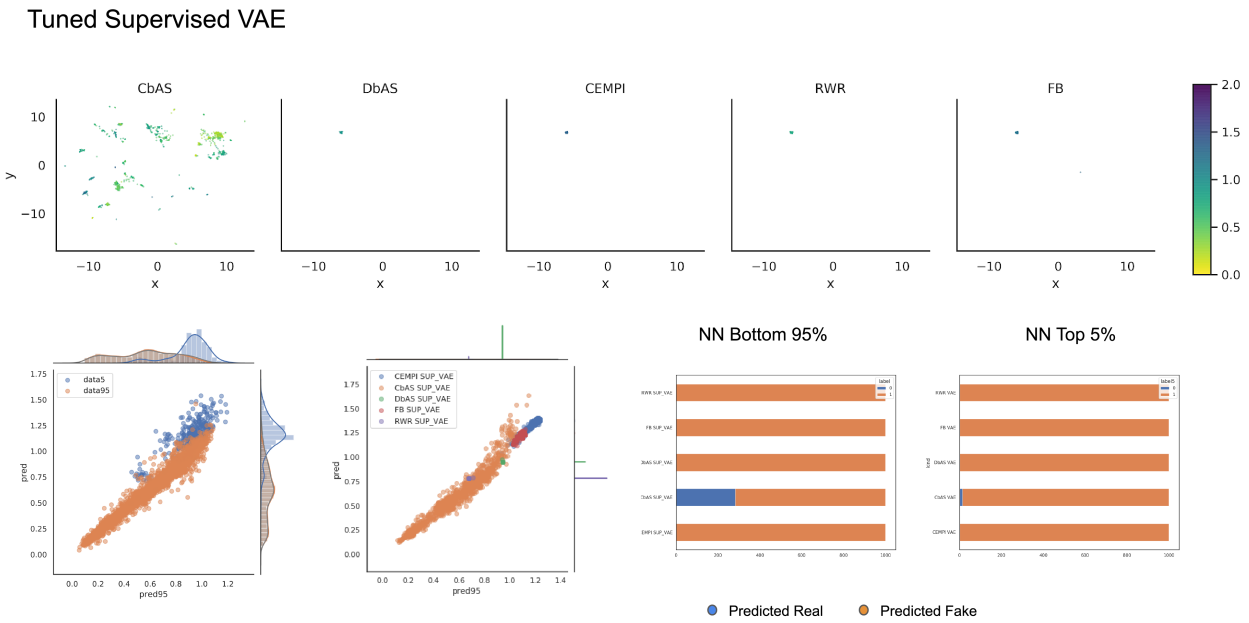

Figure 9. **Overview of performance of tuning methods applied to supervised VAEs on the task of generating active yeast promoter sequences**. The top panel shows a UMAP projection to two dimensions of the latent space projection of the sampled sequences of each method. The UMAP projection is held constant so that methods can be meaningfully compared. The bottom left two panels show the predicted expression distribution of the analyzer trained using the bottom 95% of sequence (which was optimized for tuning methods) and the analyzer trained using the full dataset. The left most plot shows the predicted values for the real data, while the plot to the right shows the predicted values for each method. The two bottom right panels shows the fraction of sequences that are predicted to be "real" or not by the 1NN algorithm for each method. The left one displays the portion of predicted "real" when using the bottom 95% as the "real" data, while the right one displays the same predictions but using only the top 5% as the "real" sequences.

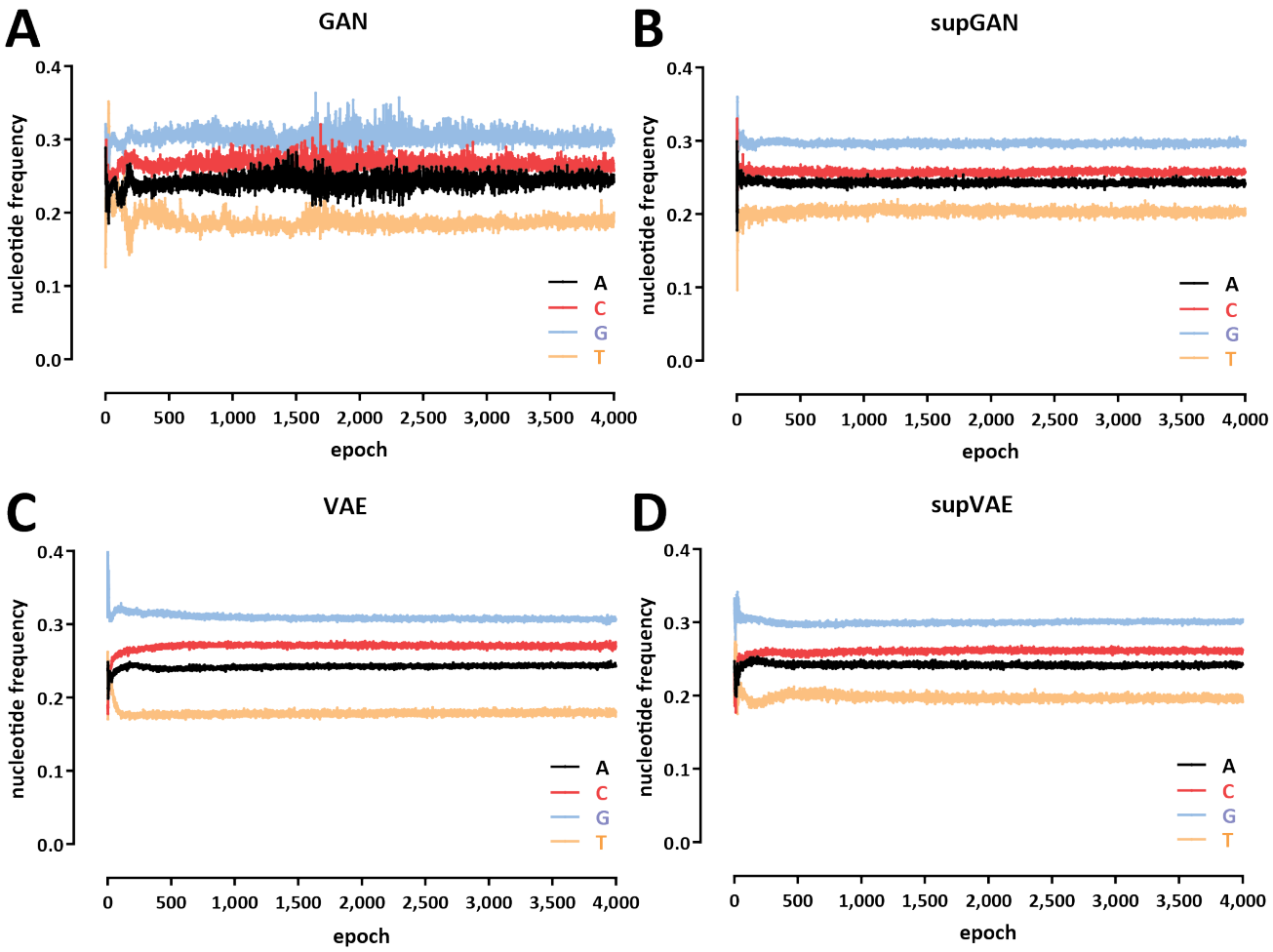

Figure 10. **Base pair composition of final generated sequences for base generative models, tuned models, and sample optimized outputs**.

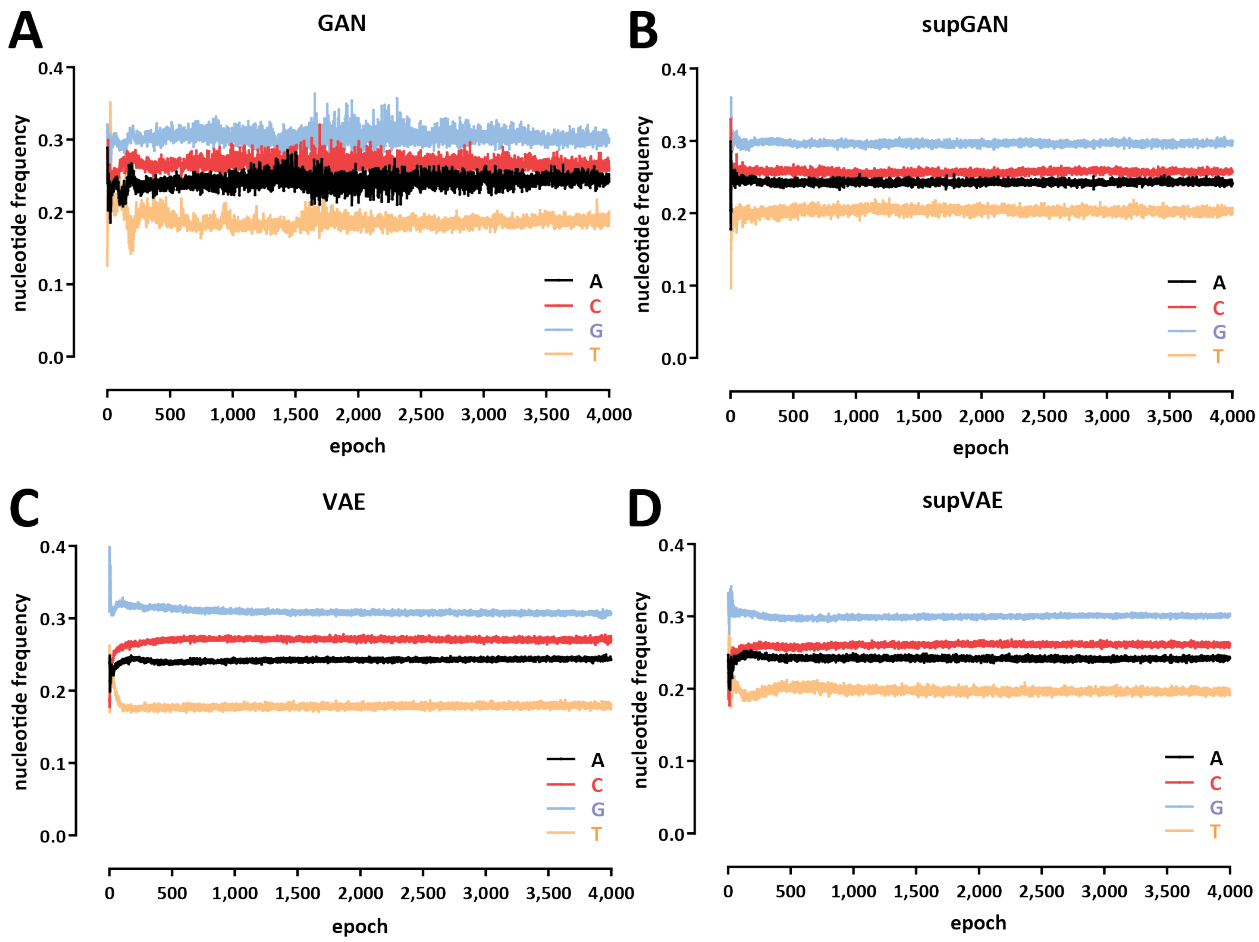

Figure 11. **Base pair composition throughout training for base models**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE).

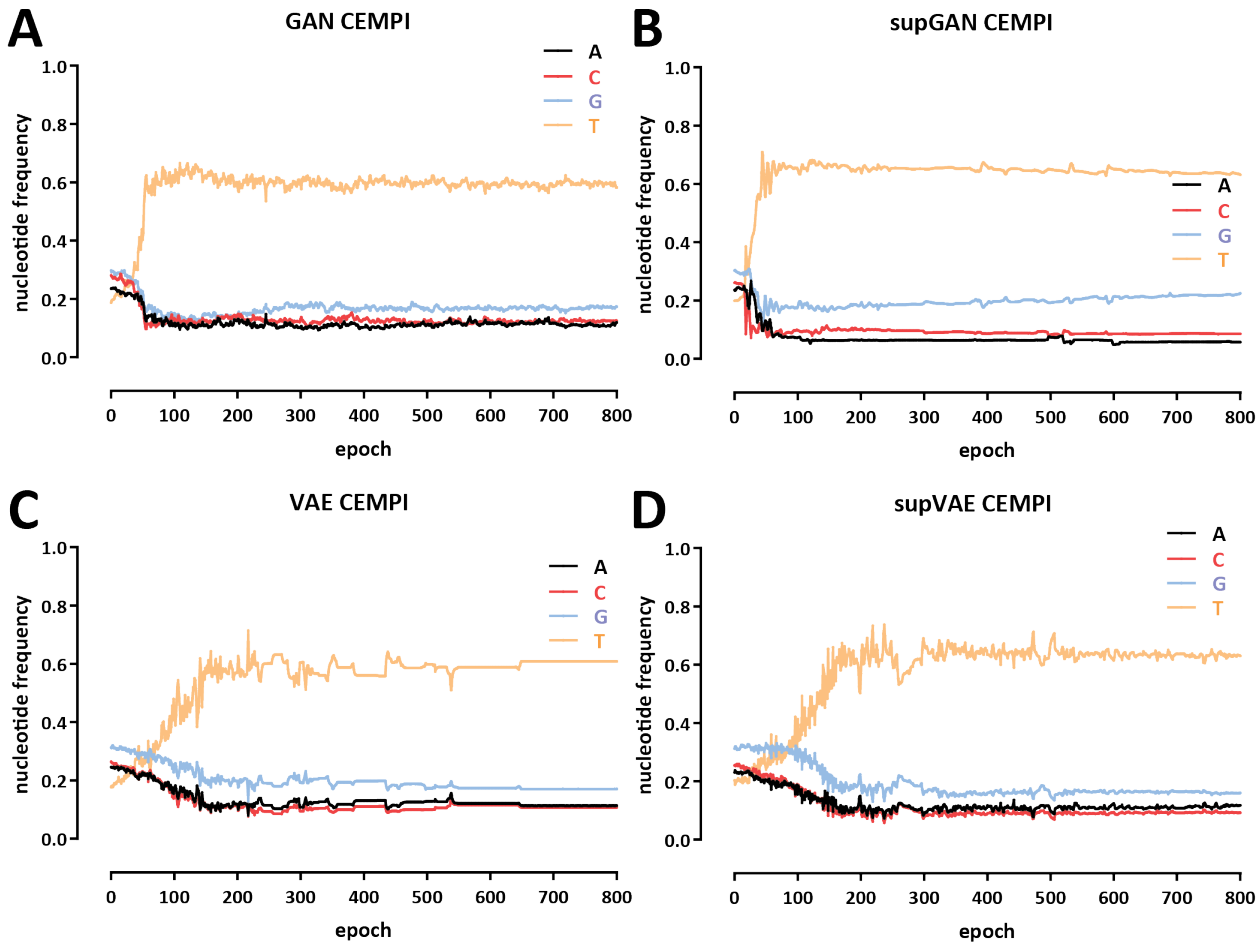

Figure 12. **Base pair composition throughout training for the CEMPI tuning method**. (A) Generative Adversarial Networks (GAN) + CEMPI; (B) Supervised Generative Adversarial Networks (supGAN) + CEMPI; (C) Variation AutoEncoders (VAE) + CEMPI; (D) Supervised Variation AutoEncoders (supVAE) + CEMPI.

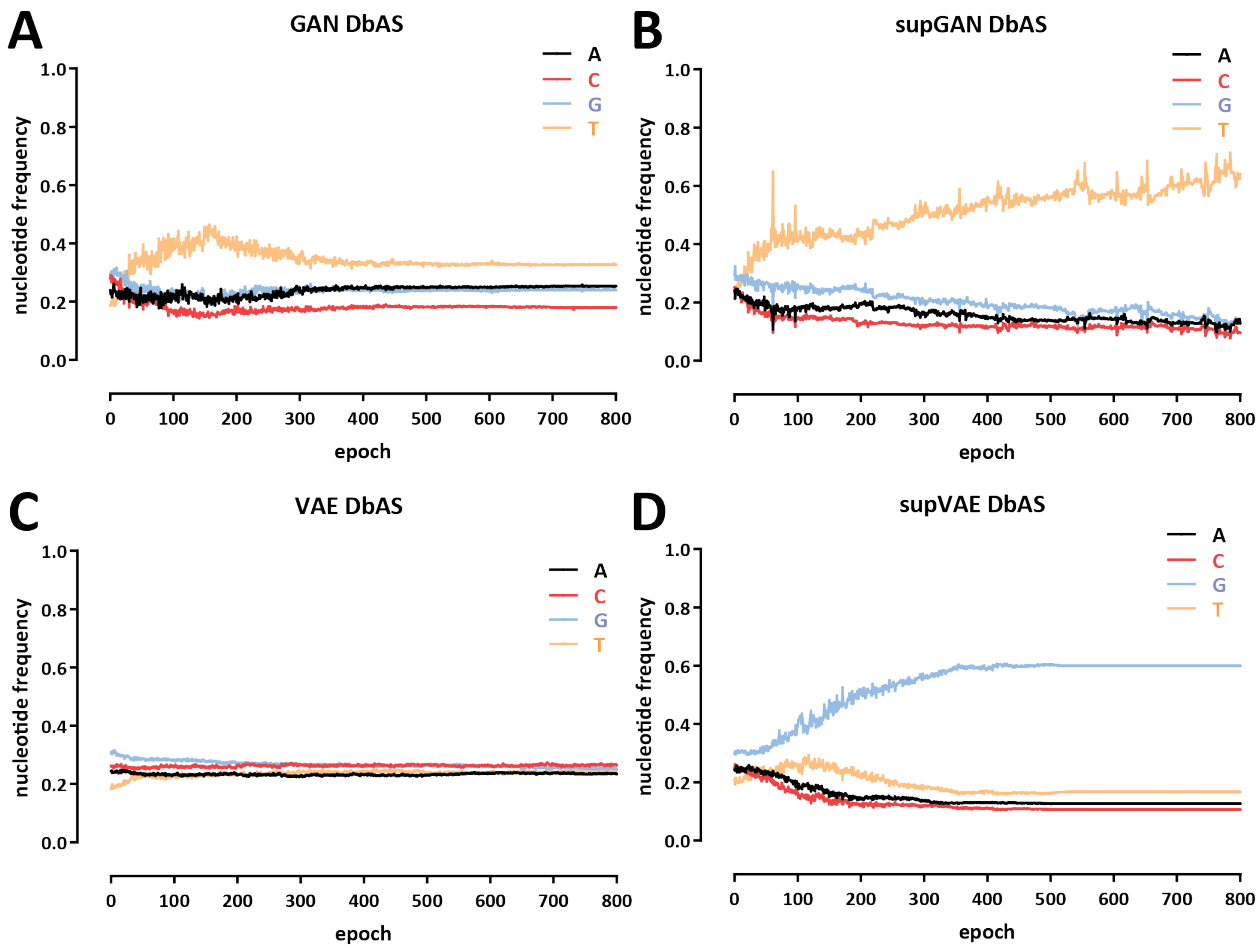

Figure 13. **Base pair composition throughout training for the DbAS tuning method**. (A) Generative Adversarial Networks (GAN) + DbAS; (B) Supervised Generative Adversarial Networks (supGAN) + DbAS; (C) Variation AutoEncoders (VAE) + DbAS; (D) Supervised Variation AutoEncoders (supVAE) + DbAS.

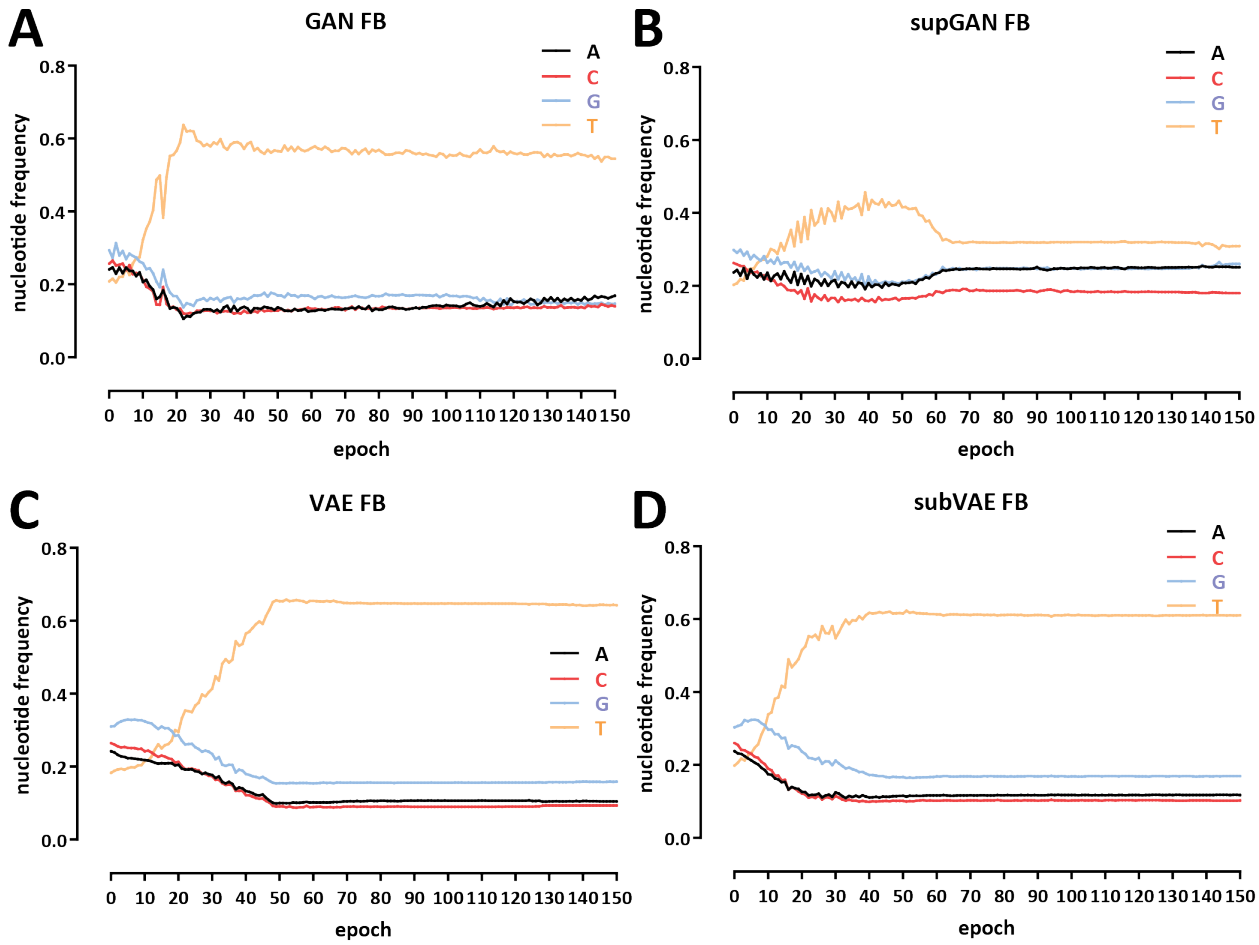

Figure 14. **Base pair composition throughout training for the FBGAN tuning approach**. (A) Generative Adversarial Networks (GAN) + FB; (B) Supervised Generative Adversarial Networks (supGAN) + FB; (C) Variation AutoEncoders (VAE) + FB; (D) Supervised Variation AutoEncoders (supVAE) + FB.

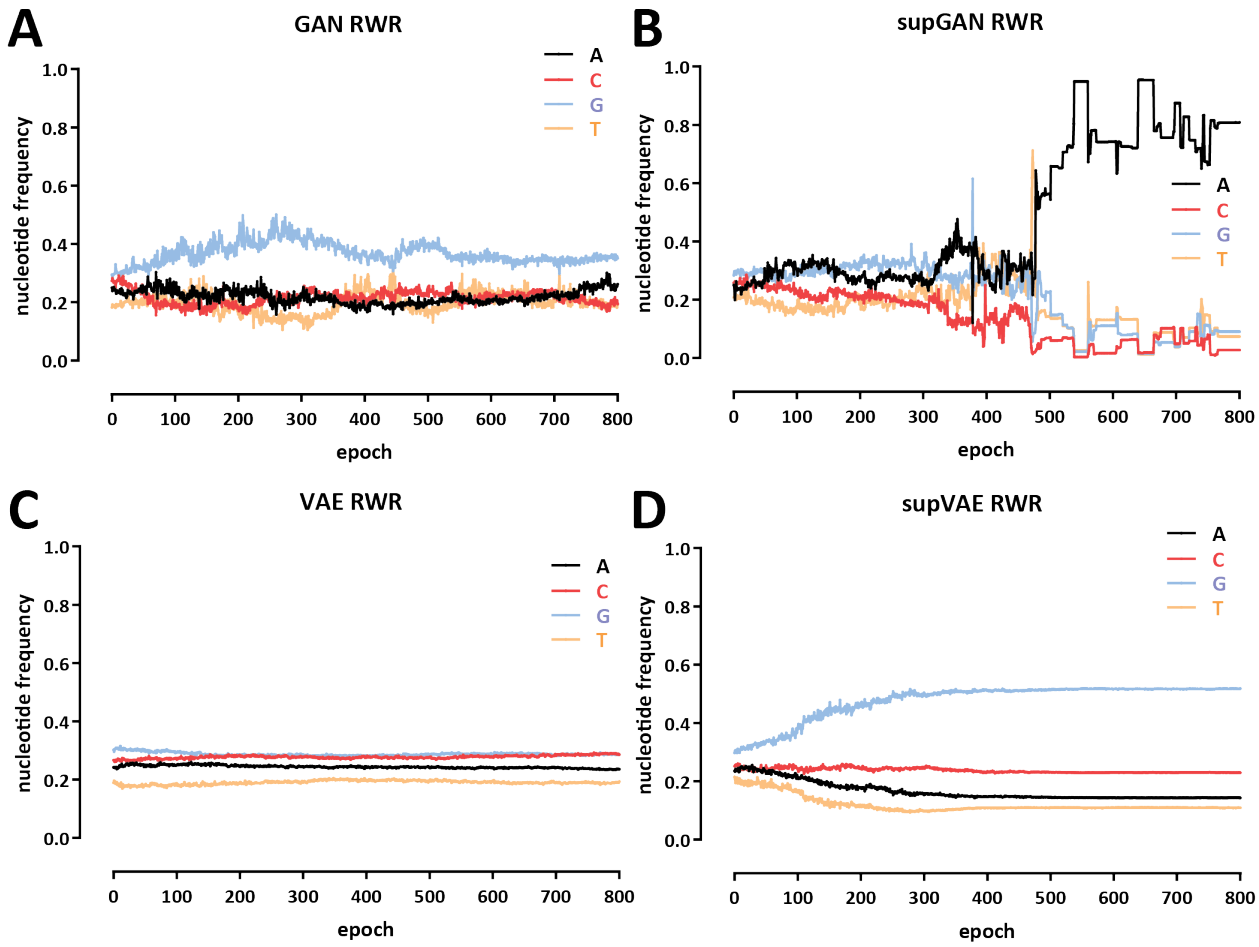

Figure 15. **Base pair composition throughout training for the RWR tuning method**. (A) Generative Adversarial Networks (GAN) + RWR; (B) Supervised Generative Adversarial Networks (supGAN) + RWR; (C) Variation AutoEncoders (VAE) + RWR; (D) Supervised Variation AutoEncoders (supVAE) + RWR.

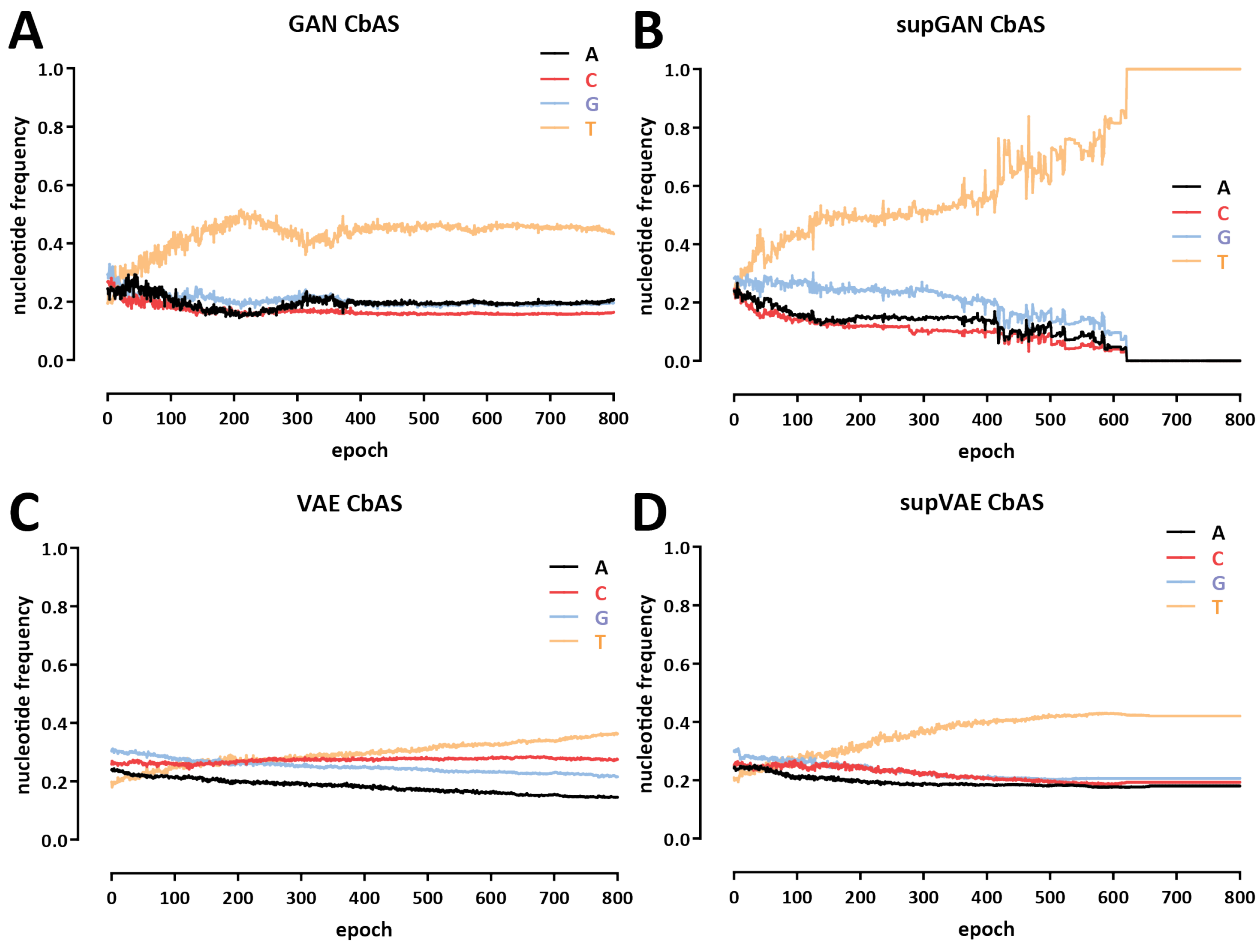

Figure 16. **Base pair composition throughout training for the CbAS tuning method**. (A) Generative Adversarial Networks (GAN) + CbAS; (B) Supervised Generative Adversarial Networks (supGAN) + CbAS; (C) Variation AutoEncoders (VAE) + CbAS; (D) Supervised Variation AutoEncoders (supVAE) + CbAS.

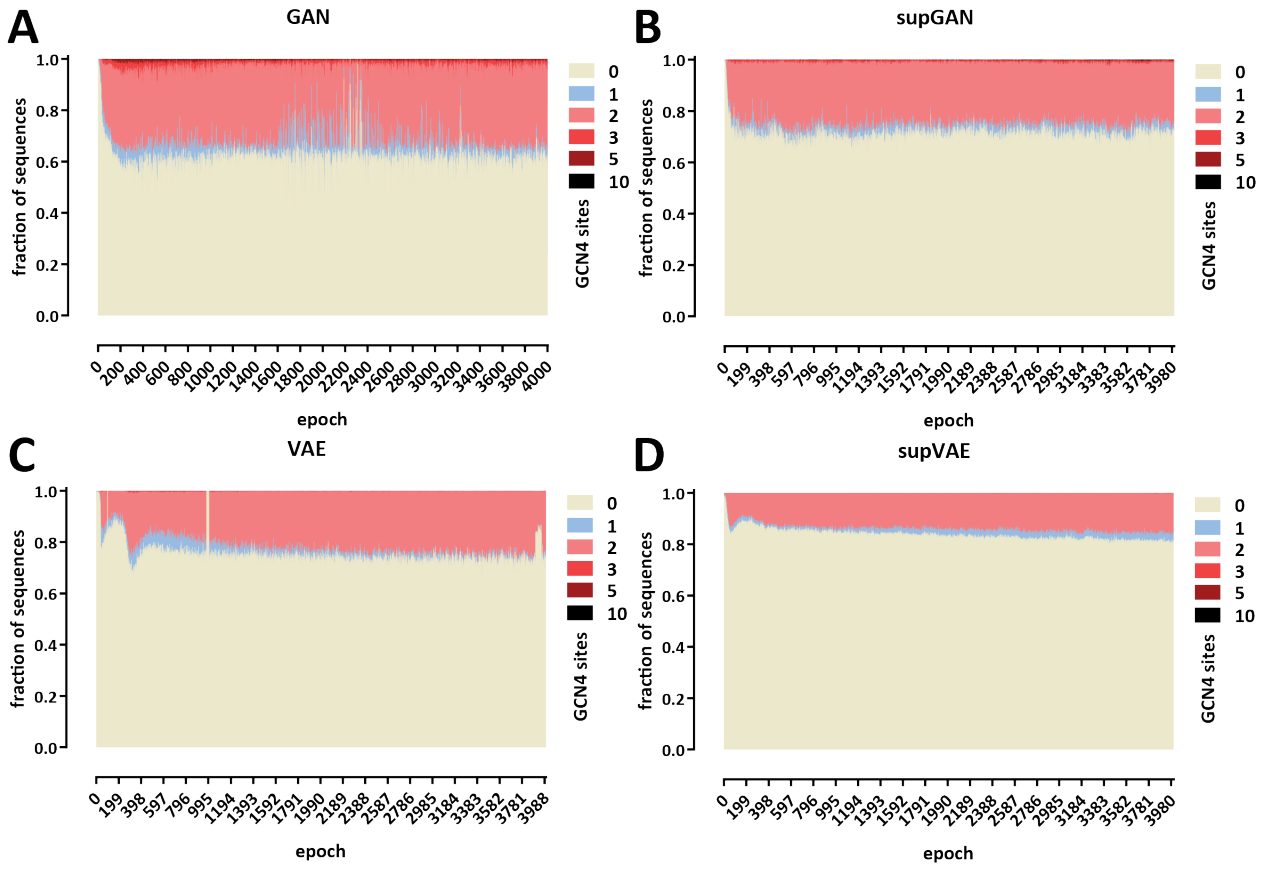

Figure 17. **GCN4 motif content throughout training for base models**. (A) Generative Adversarial Networks (GAN) (B) Supervised Generative Adversarial Networks (supGAN) (C) Variation AutoEncoders (VAE) (D) Supervised Variation AutoEncoders (supVAE)

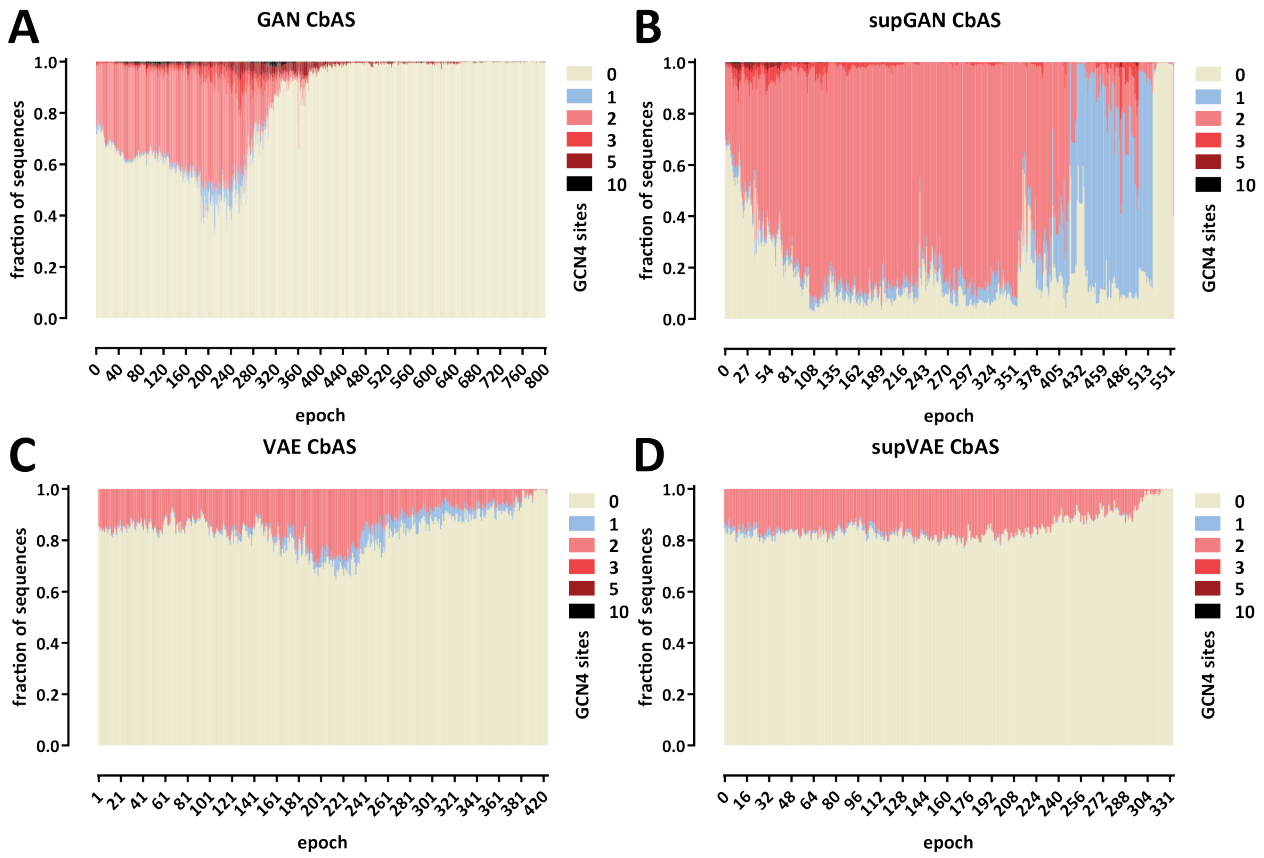

Figure 18. **GCN4 motif content throughout training for the CbAS tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

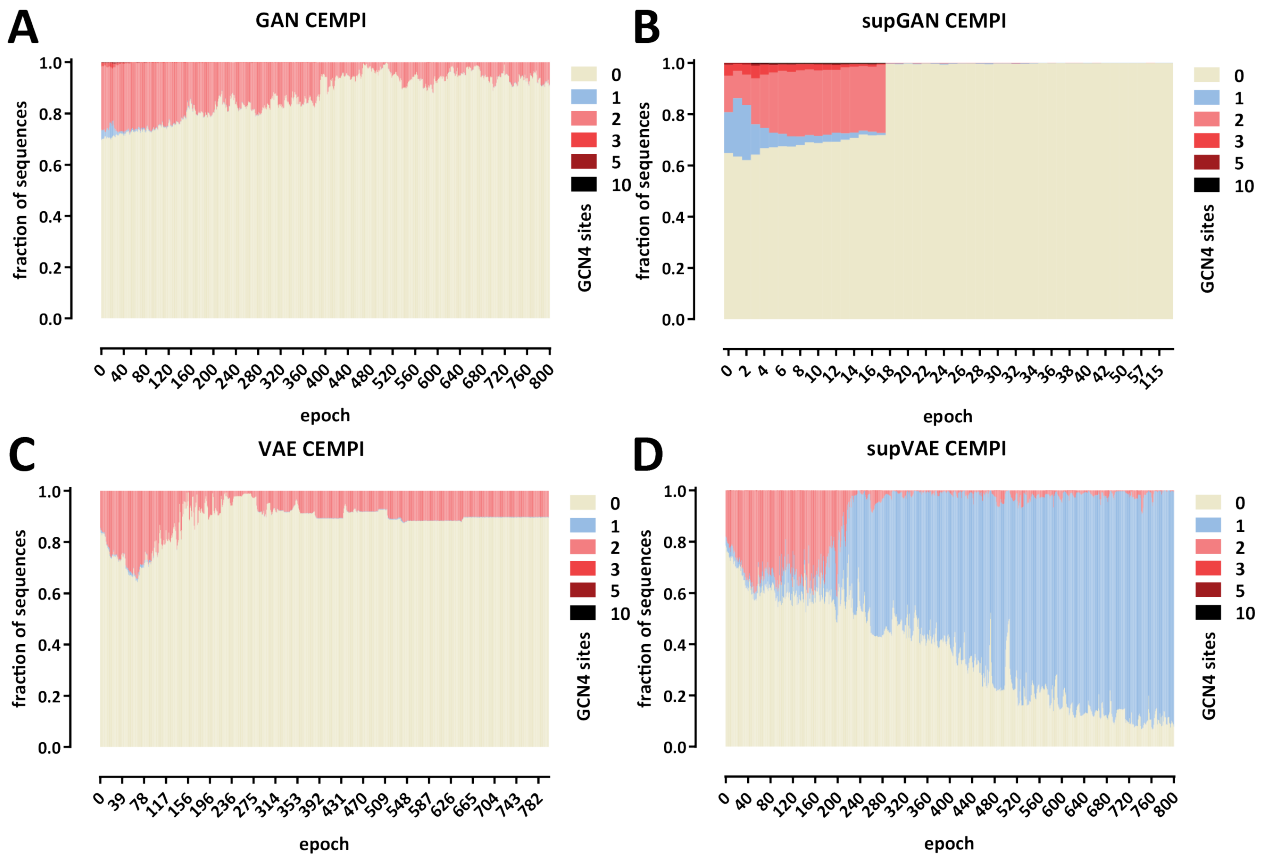

Figure 19. **GCN4 motif content throughout training for the CEMPI tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

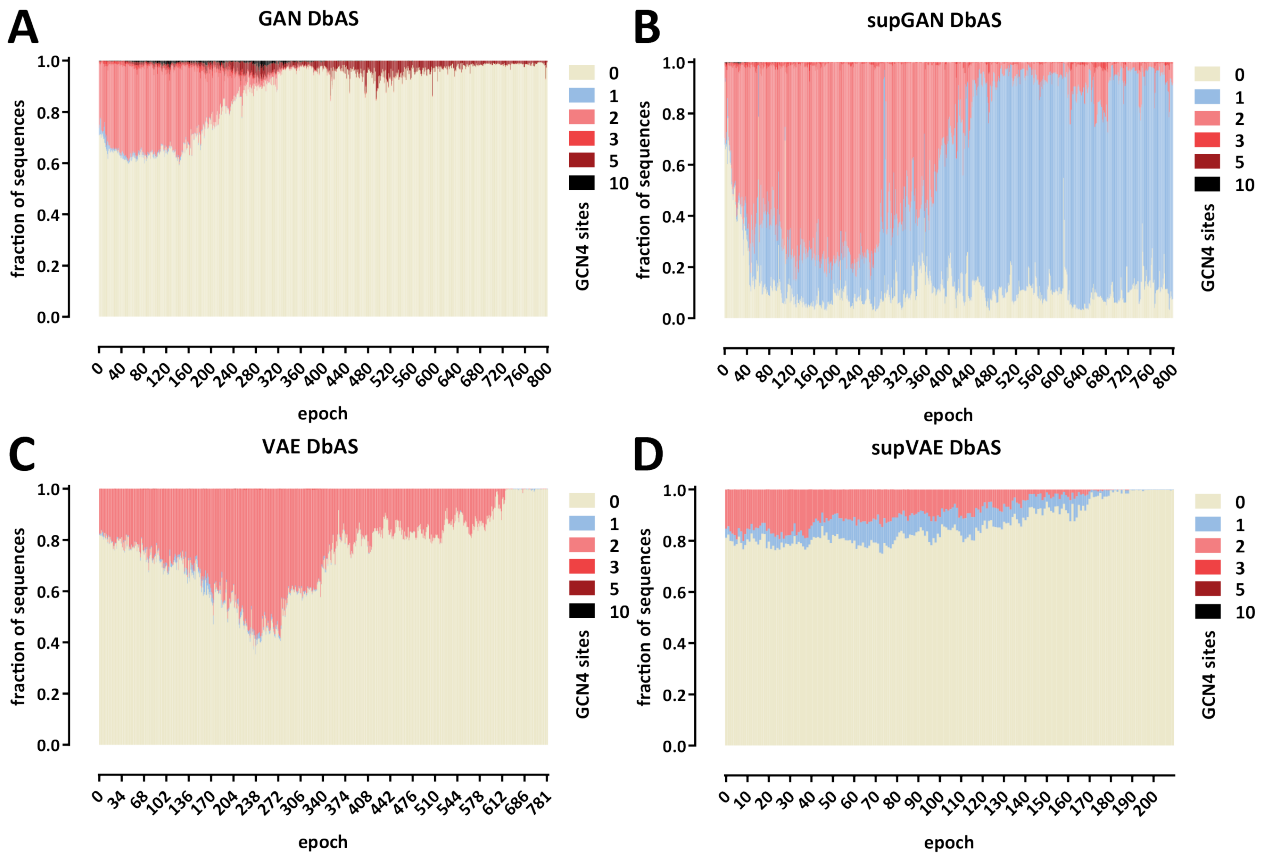

Figure 20. **GCN4 motif content throughout training for the DbAS tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

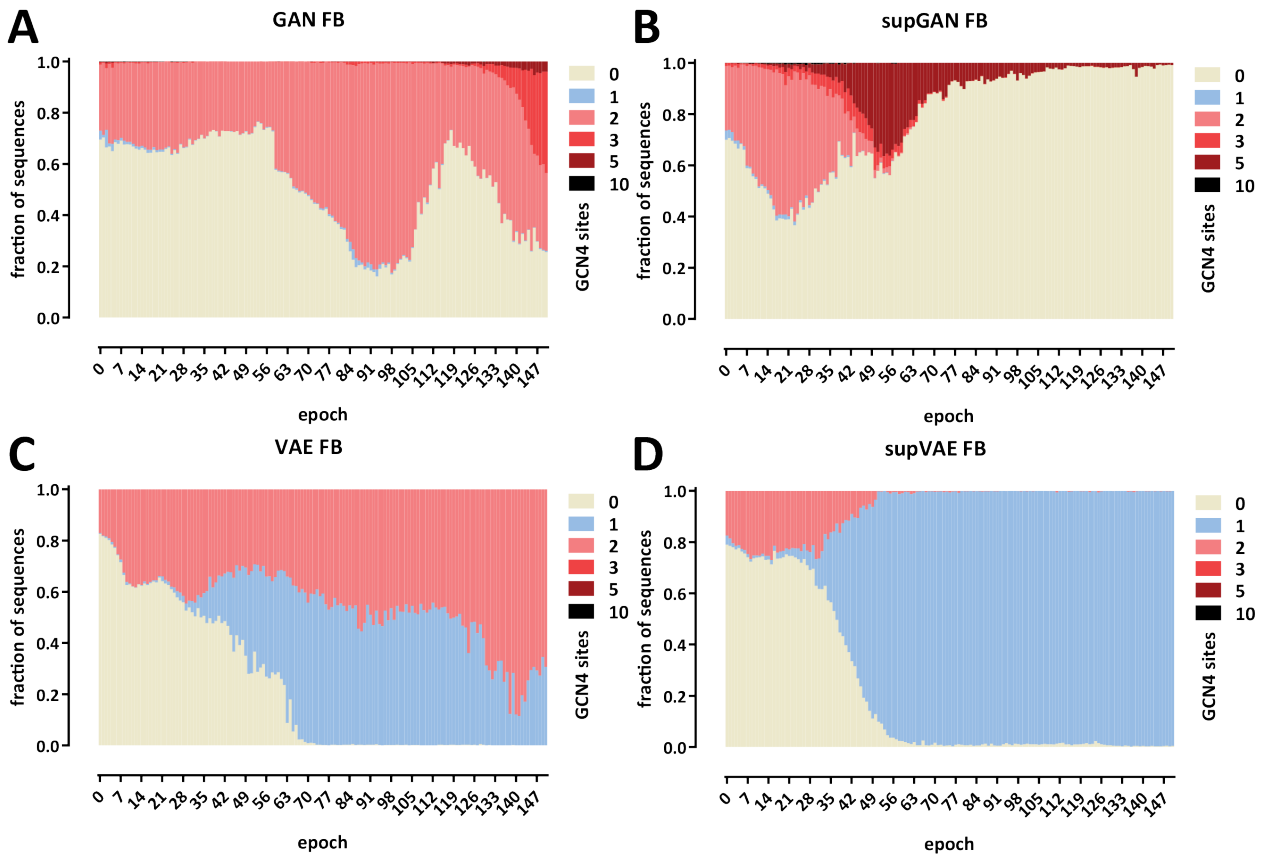

Figure 21. **GCN4 motif content throughout training for the FBGAN tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE).

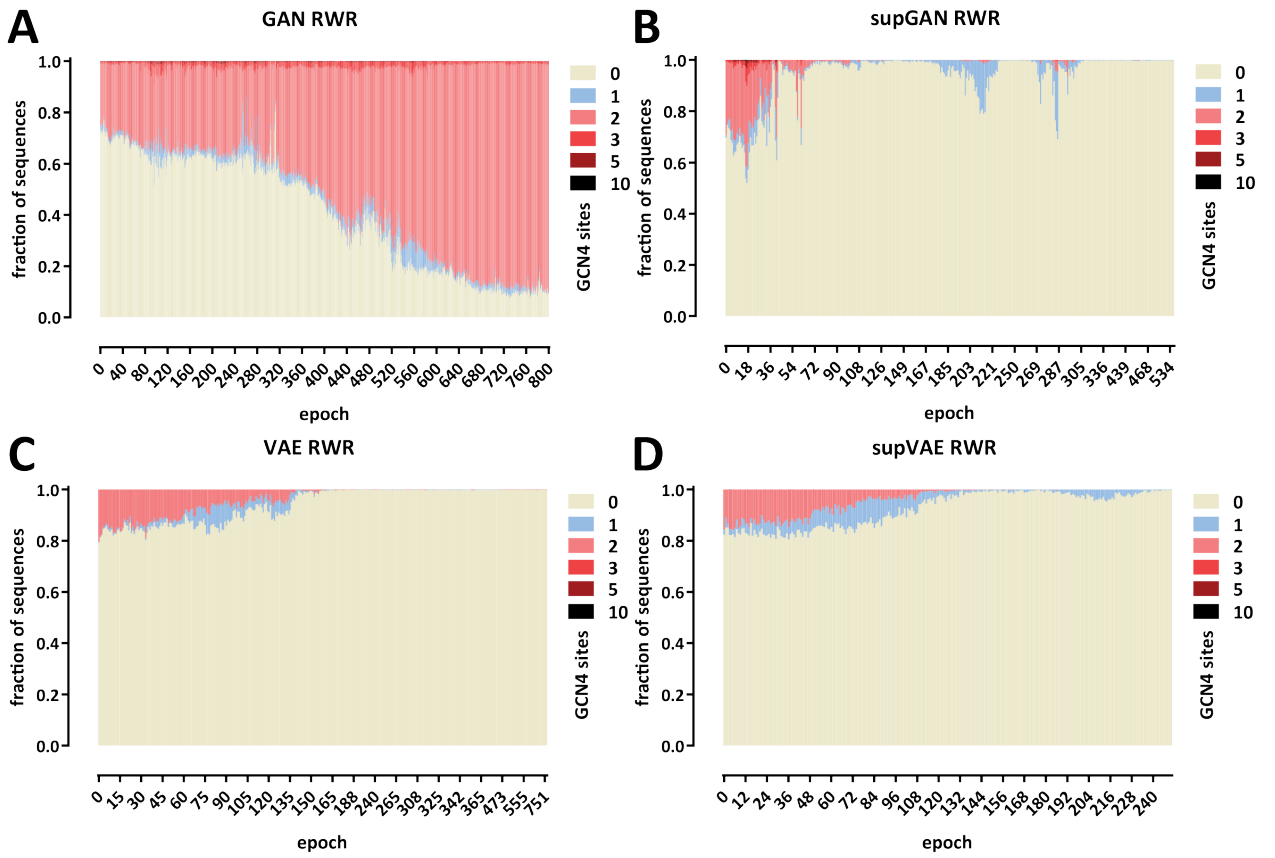

Figure 22. **GCN4 motif content throughout training for the RWR tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

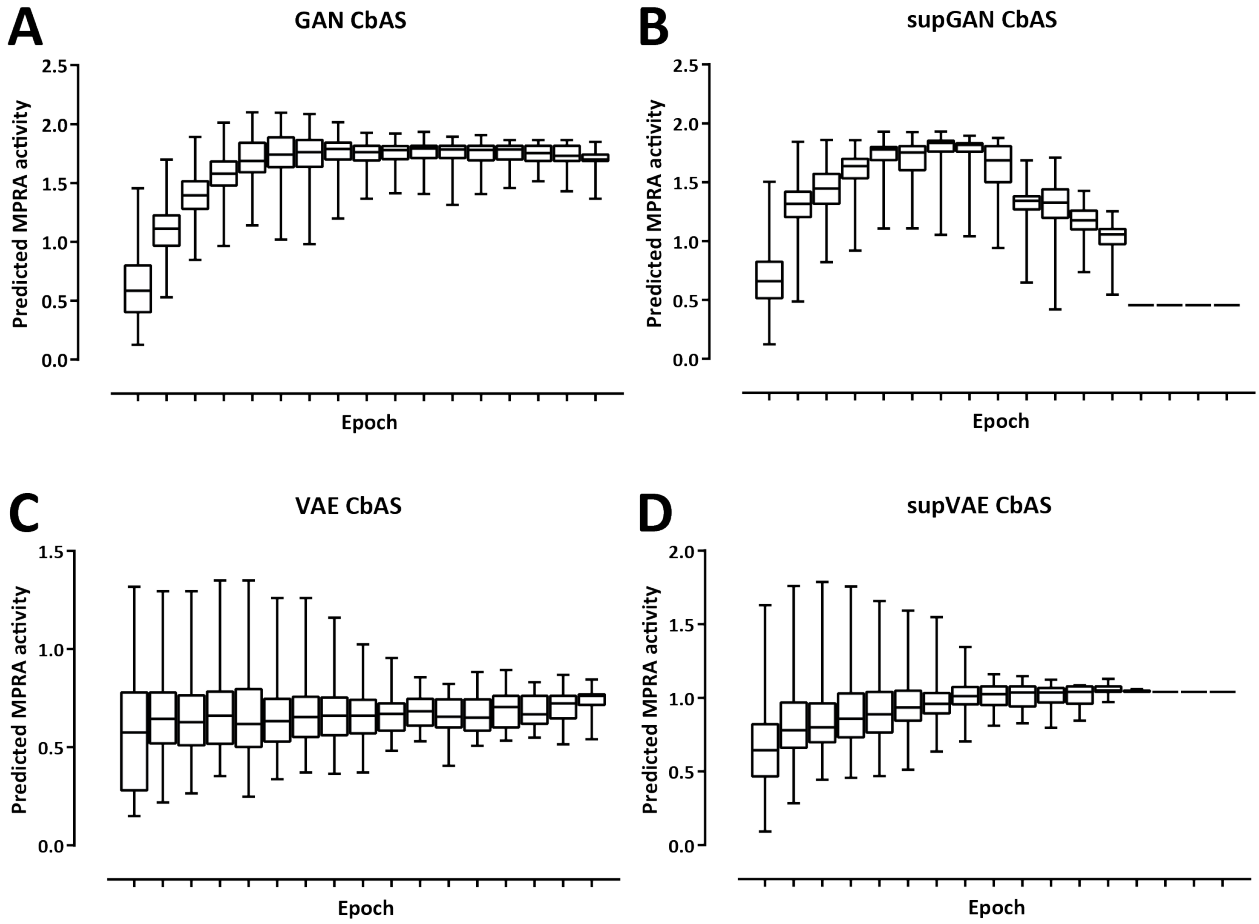

Figure 23. **Predicted expression levels throughout training for the CbAS tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

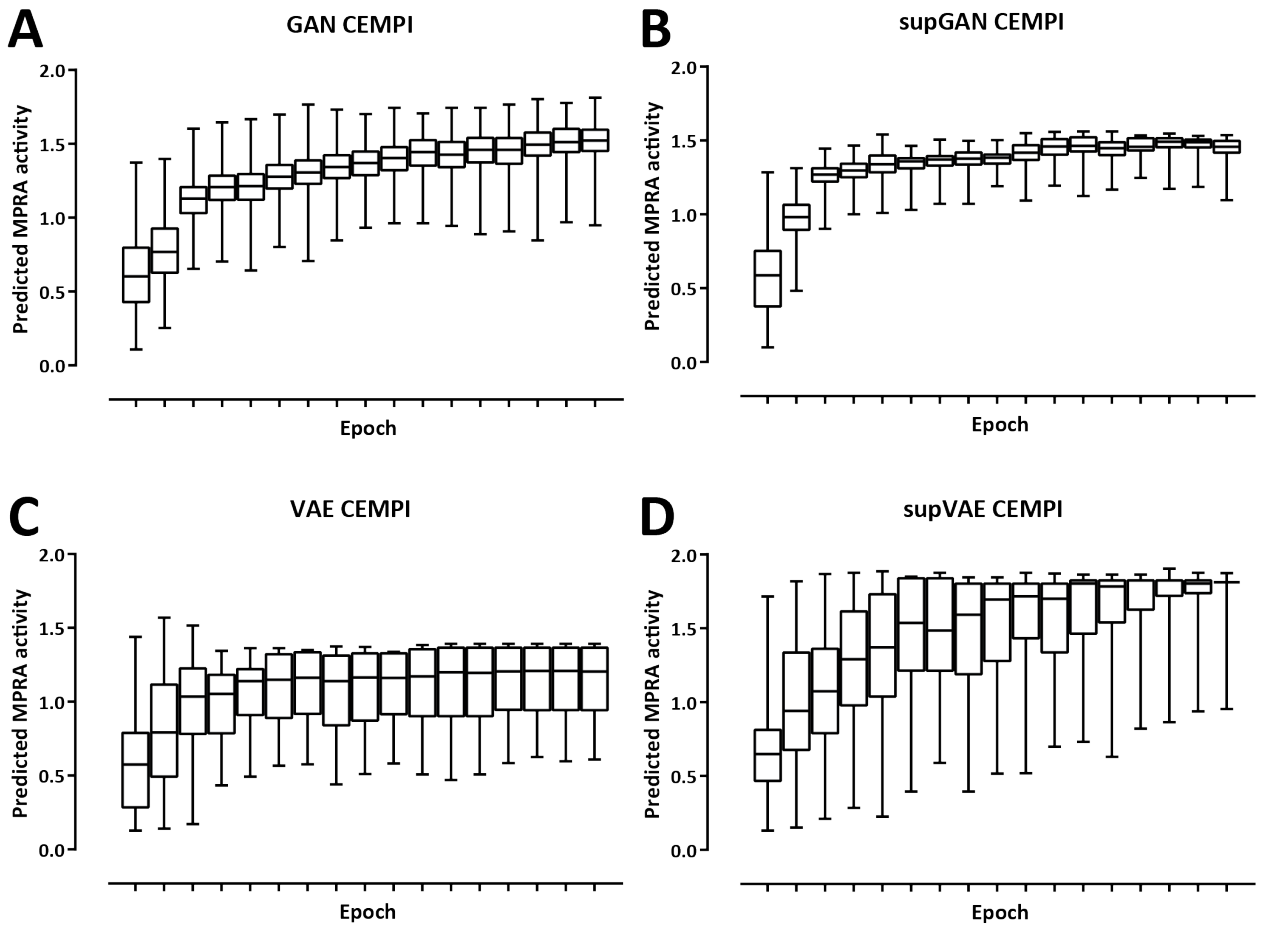

Figure 24. **Predicted expression levels throughout training for the CEMPI tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

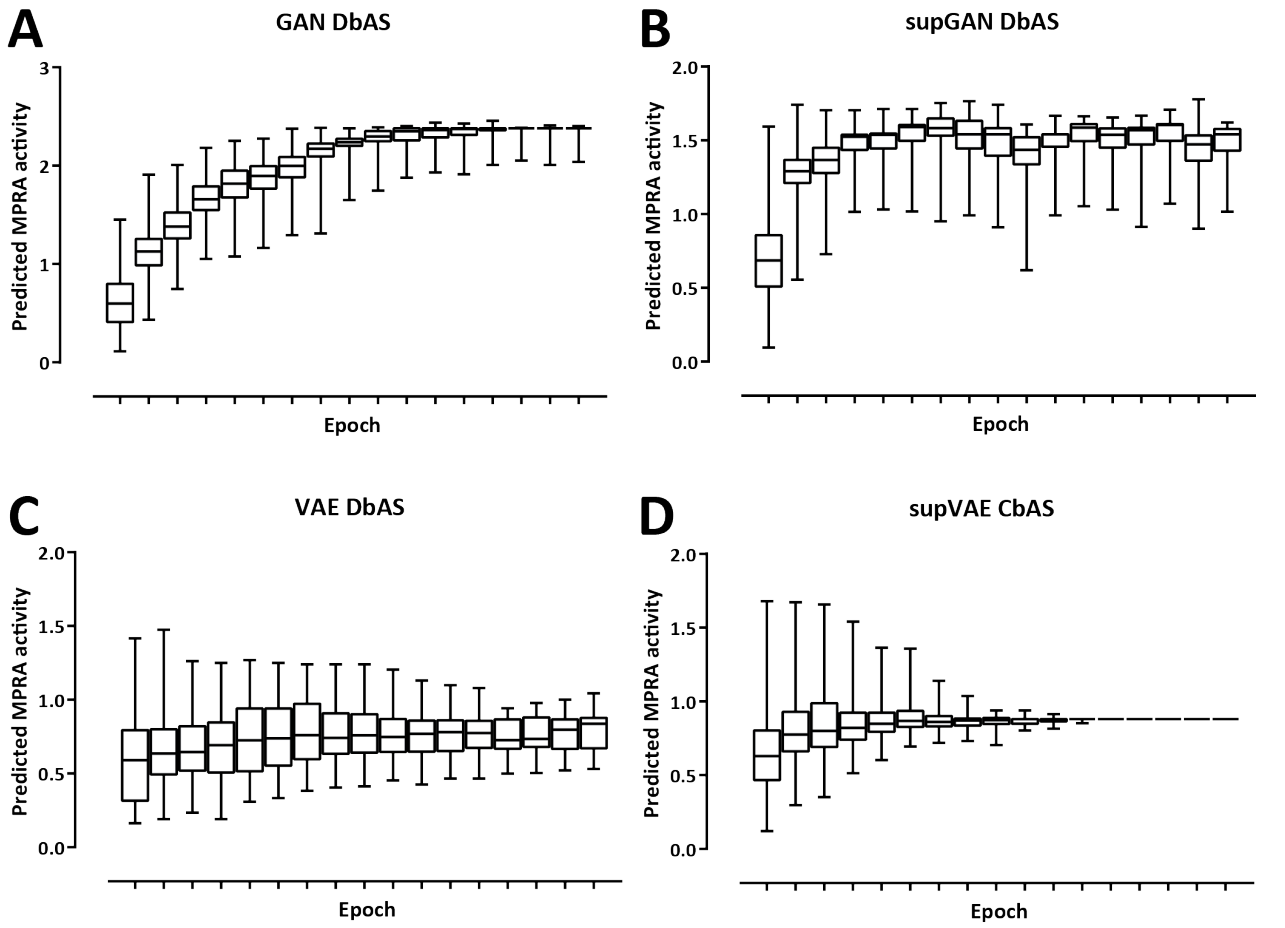

Figure 25. **Predicted expression levels throughout training for the DbAS tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

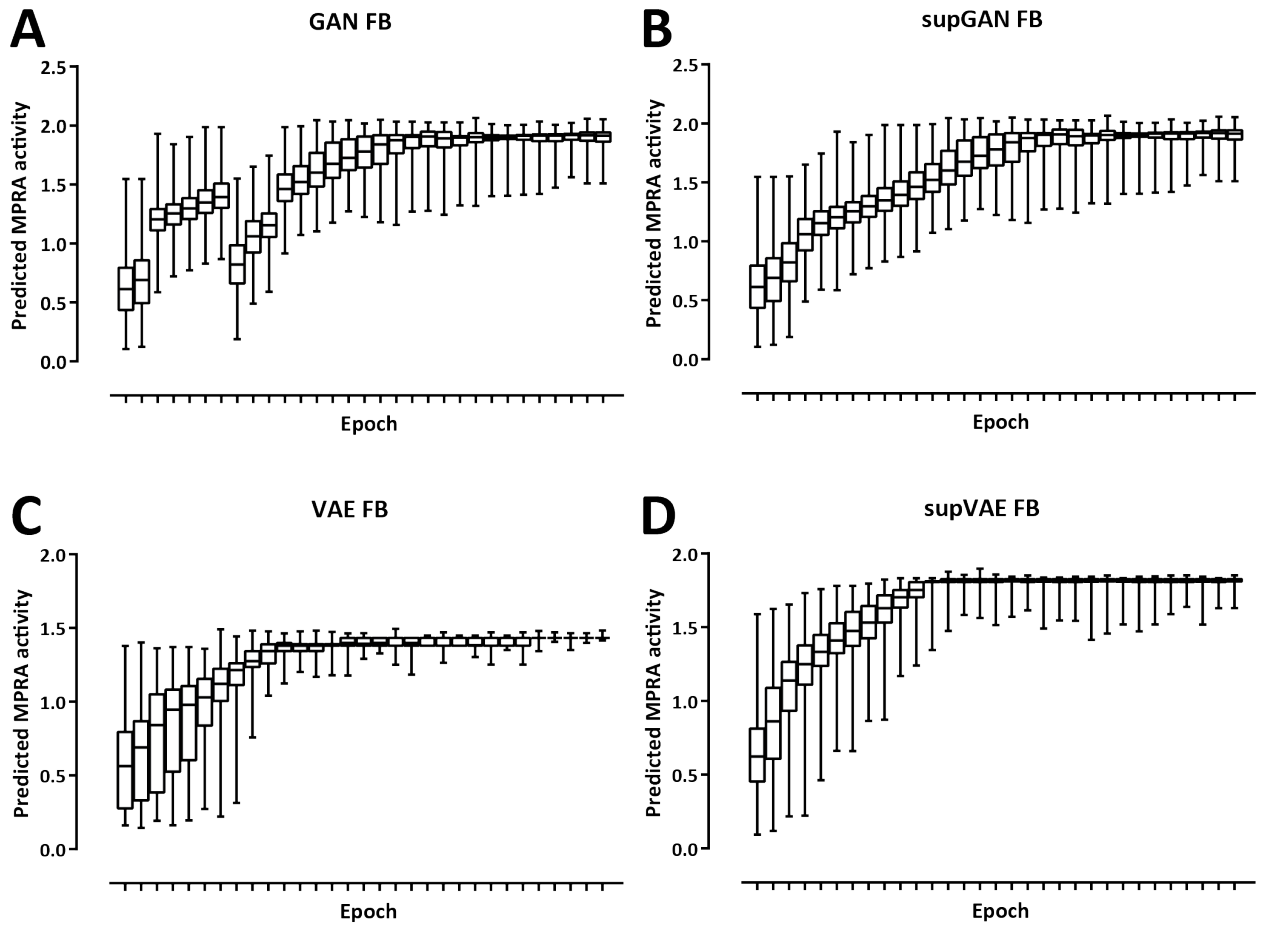

Figure 26. **Predicted expression levels throughout training for the FBGANtuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

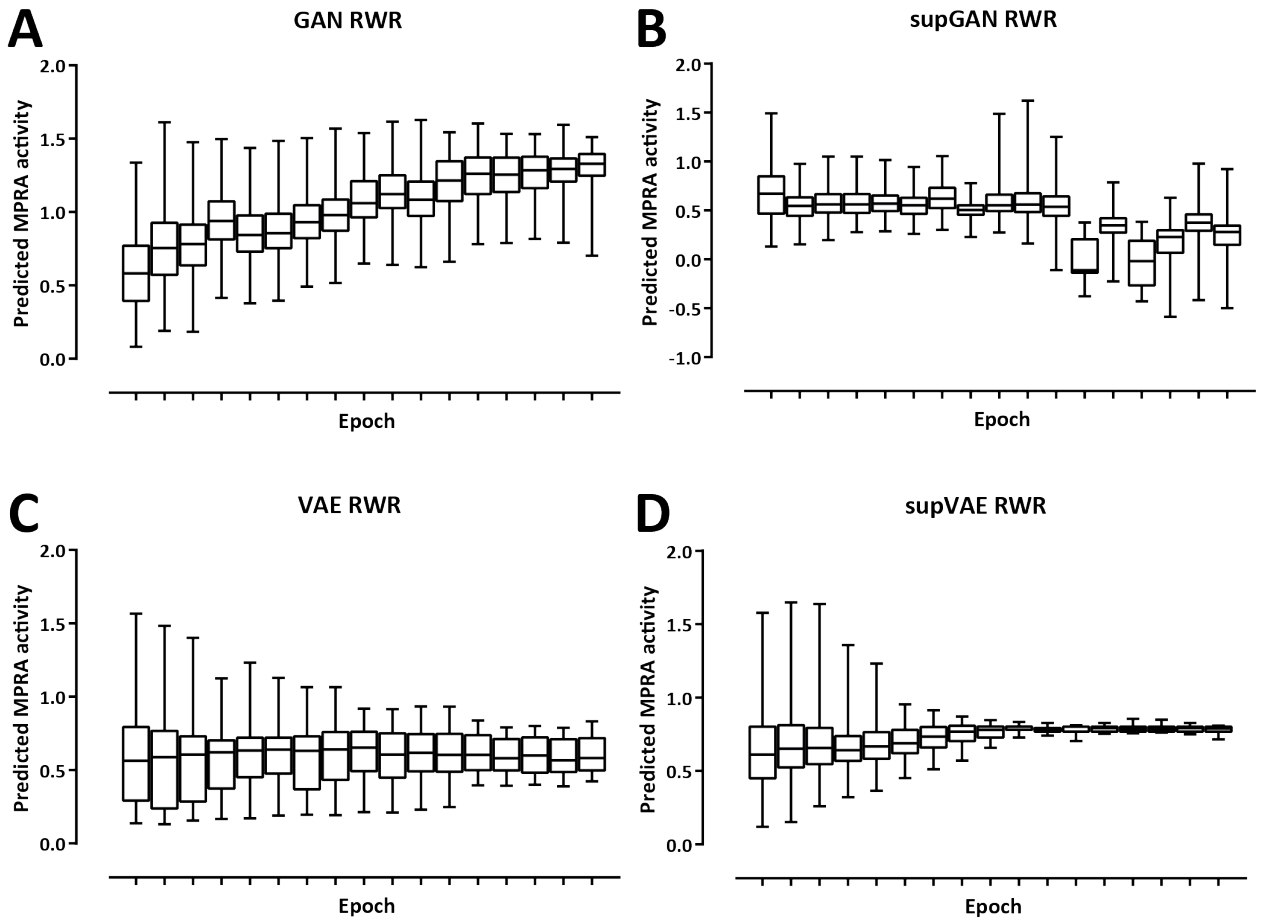

Figure 27. **Predicted expression levels throughout training for the RWR tuning approach**. (A) Generative Adversarial Networks (GAN); (B) Supervised Generative Adversarial Networks (supGAN); (C) Variation AutoEncoders (VAE); (D) Supervised Variation AutoEncoders (supVAE). Epochs beyond the ones shown do not contain any GCN4 motifs.

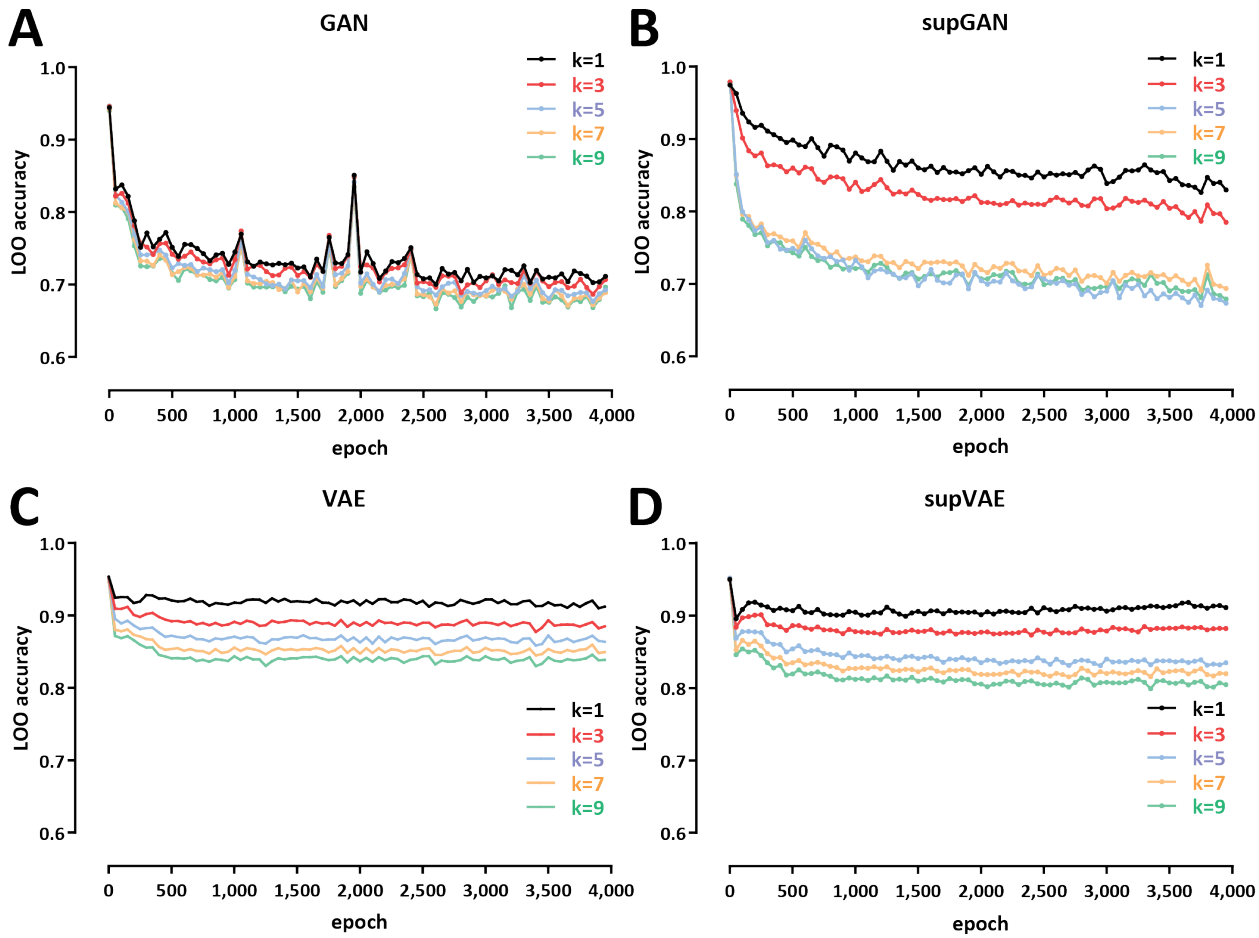

Figure 28. **Nearest neighbor evaluation of generated sequences for base models** . (A) Generative Adversarial Networks (GAN) (B) Supervised Generative Adversarial Networks (supGAN) (C) Variation AutoEncoders (VAE) (D) Supervised Variation AutoEncoders (supVAE)

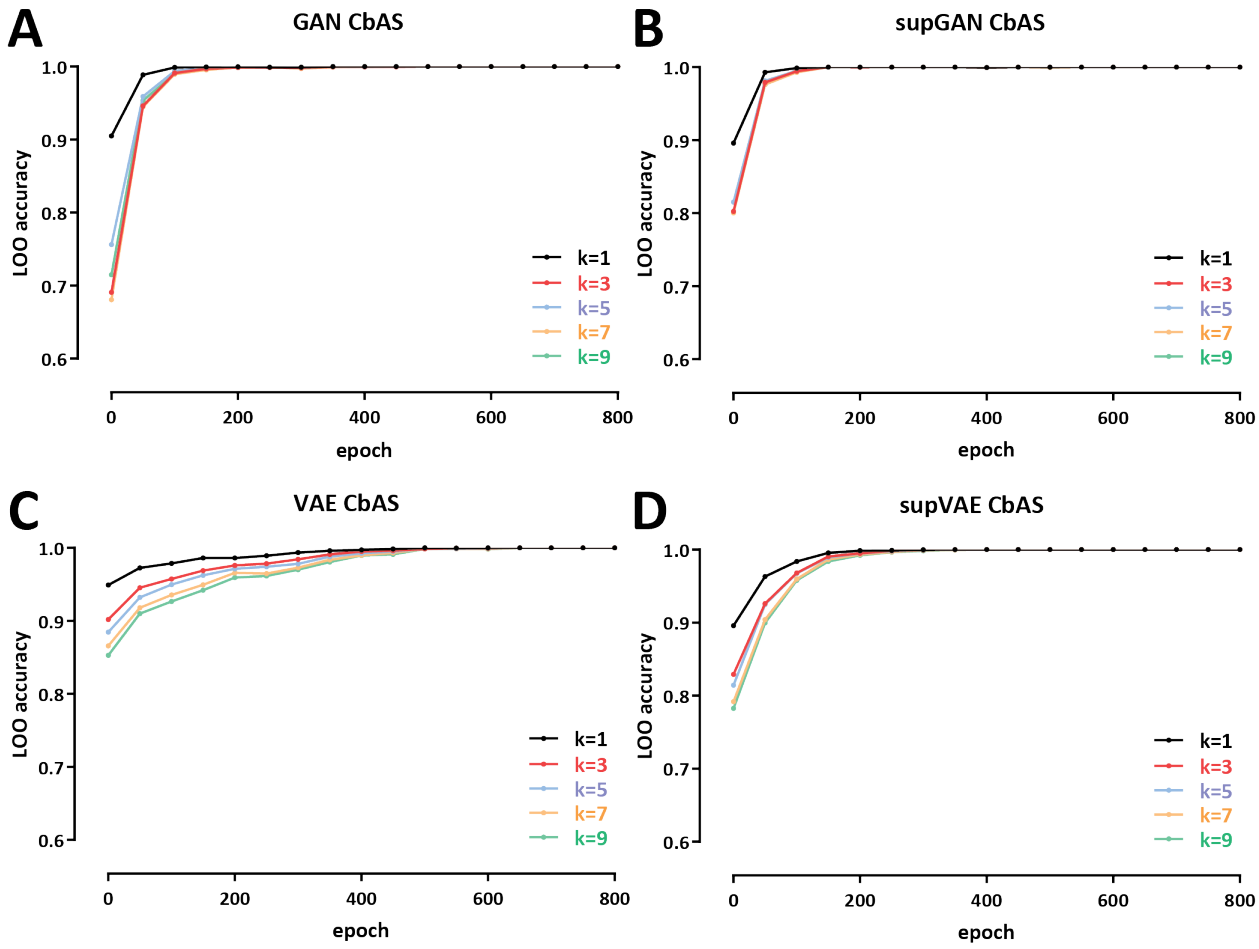

Figure 29. **Nearest neighbor evaluation of sequences generated using the CbAS tunning method** . (A) Generative Adversarial Networks (GAN) (B) Supervised Generative Adversarial Networks (supGAN) (C) Variation AutoEncoders (VAE) (D) Supervised Variation AutoEncoders (supVAE)

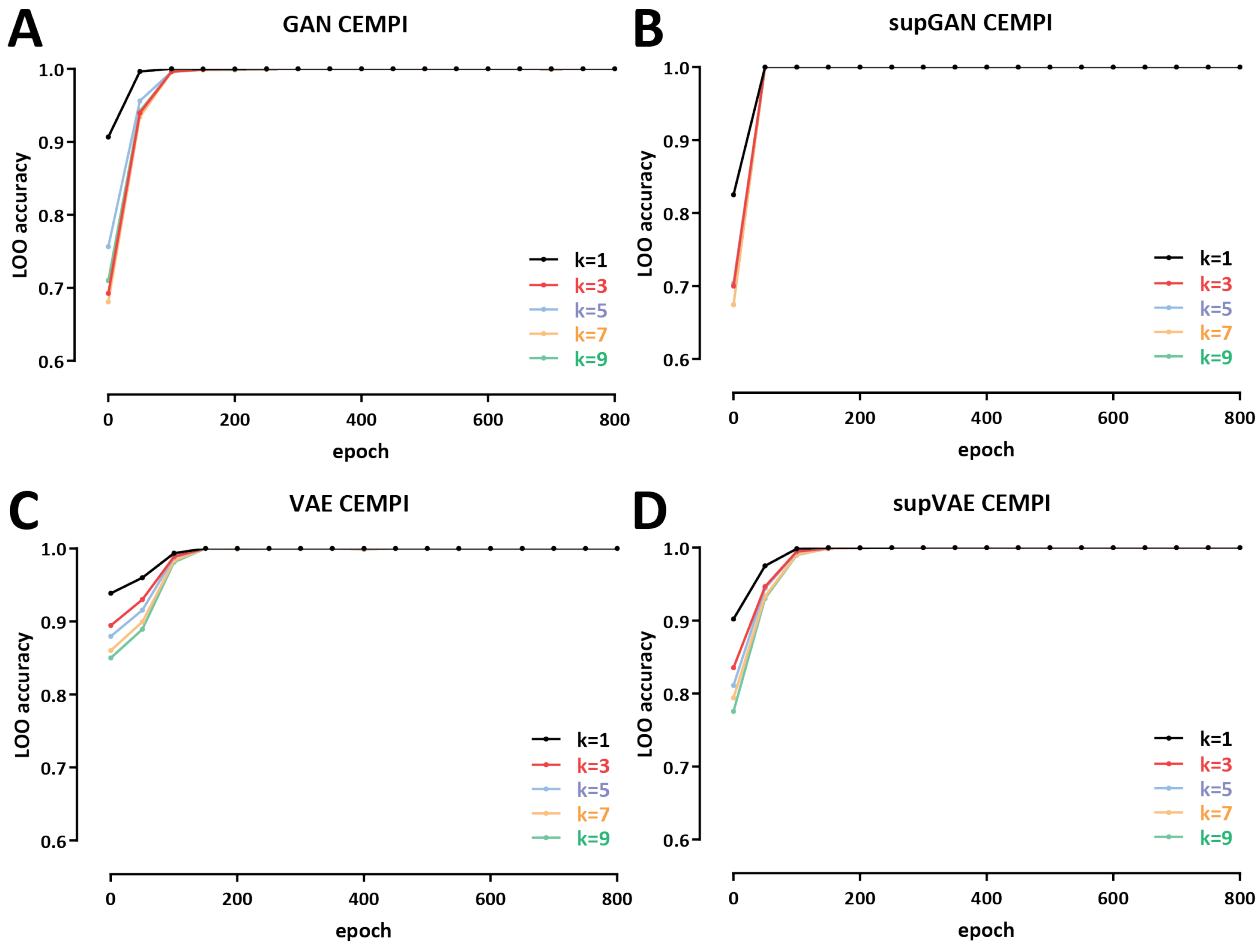

Figure 30. **Nearest neighbor evaluation of sequences generated using the CEMPI tunning method** . (A) Generative Adversarial Networks (GAN) (B) Supervised Generative Adversarial Networks (supGAN) (C) Variation AutoEncoders (VAE) (D) Supervised Variation AutoEncoders (supVAE)

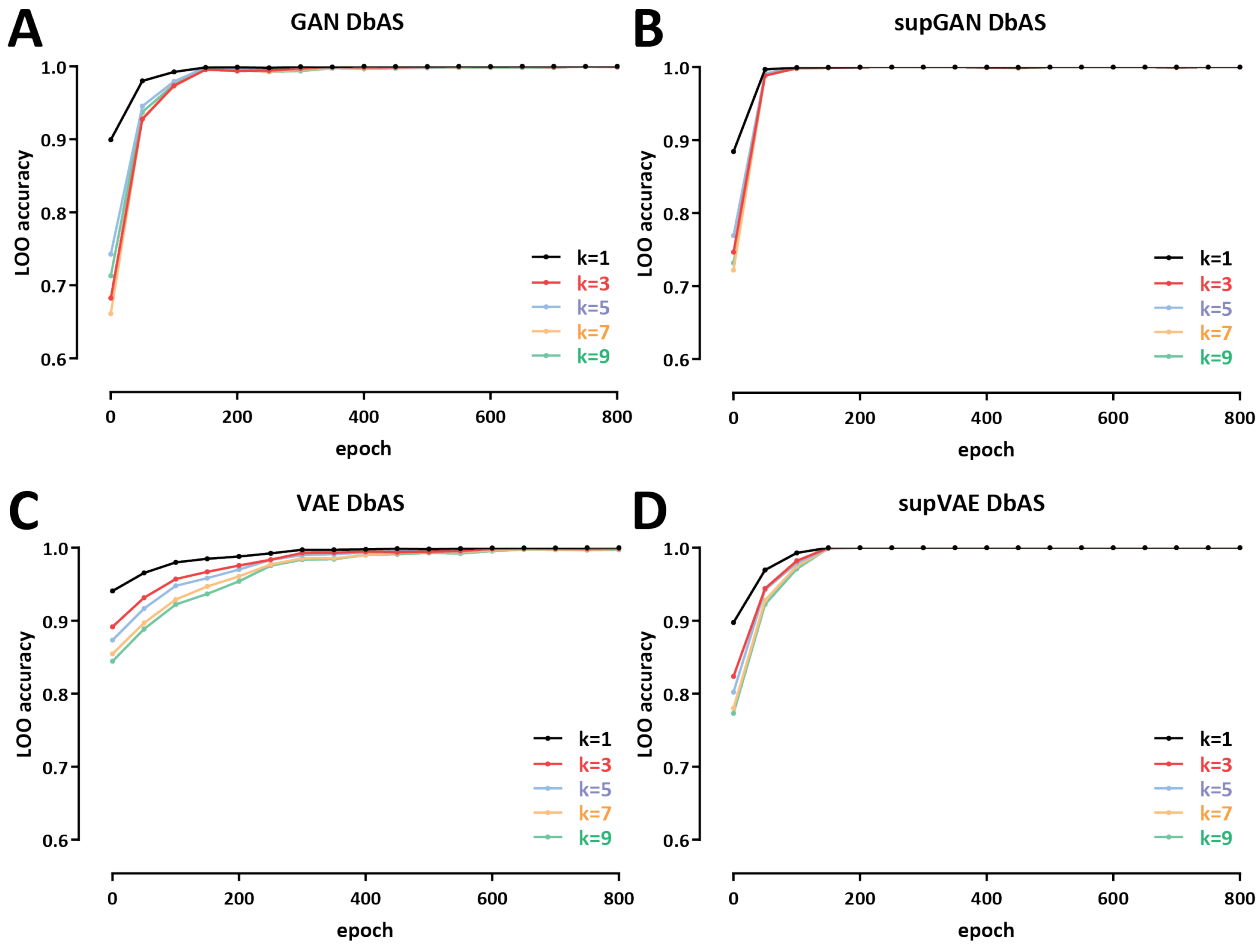

Figure 31. **Nearest neighbor evaluation of sequences generated using the DbAS tunning method** . (A) Generative Adversarial Networks (GAN) (B) Supervised Generative Adversarial Networks (supGAN) (C) Variation AutoEncoders (VAE) (D) Supervised Variation AutoEncoders (supVAE)

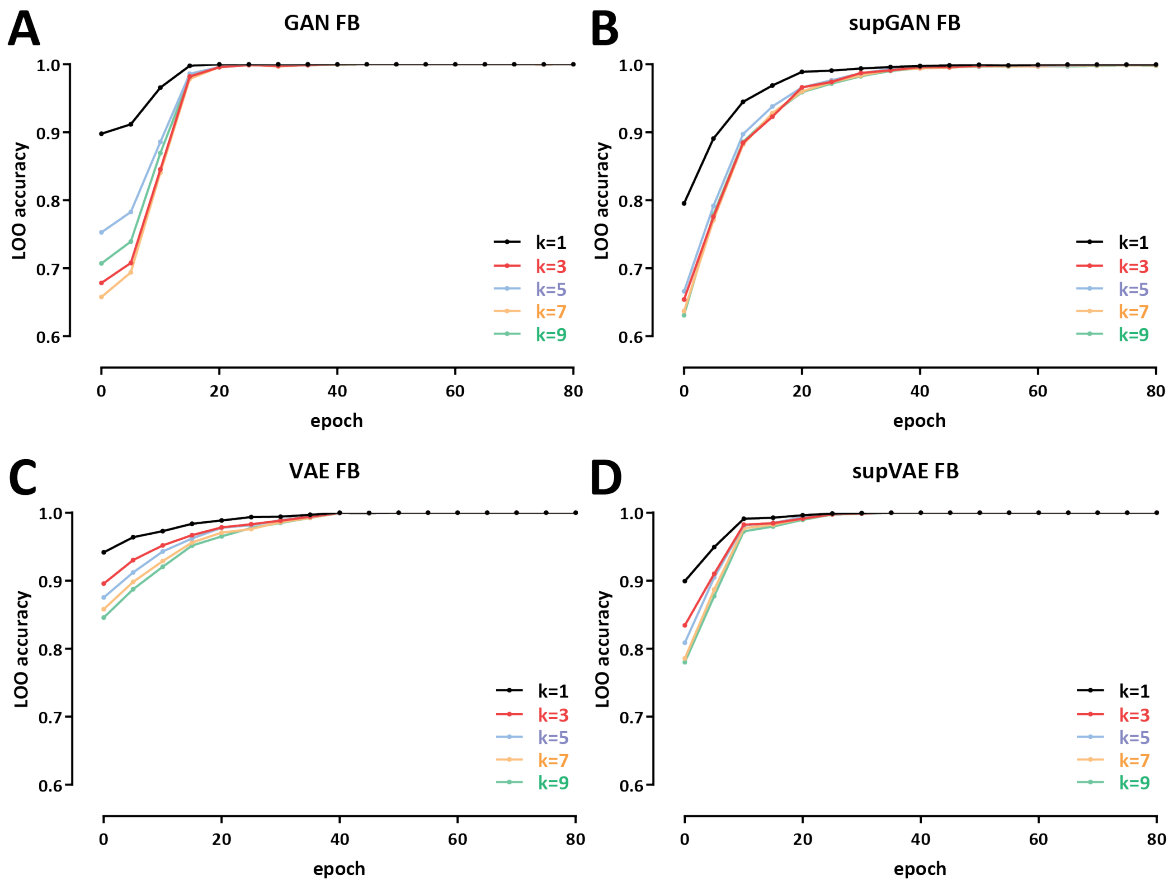

Figure 32. **Nearest neighbor evaluation of sequences generated using the FBGAN tunning method** . (A) Generative Adversarial Networks (GAN) (B) Supervised Generative Adversarial Networks (supGAN) (C) Variation AutoEncoders (VAE) (D) Supervised Variation AutoEncoders (supVAE)

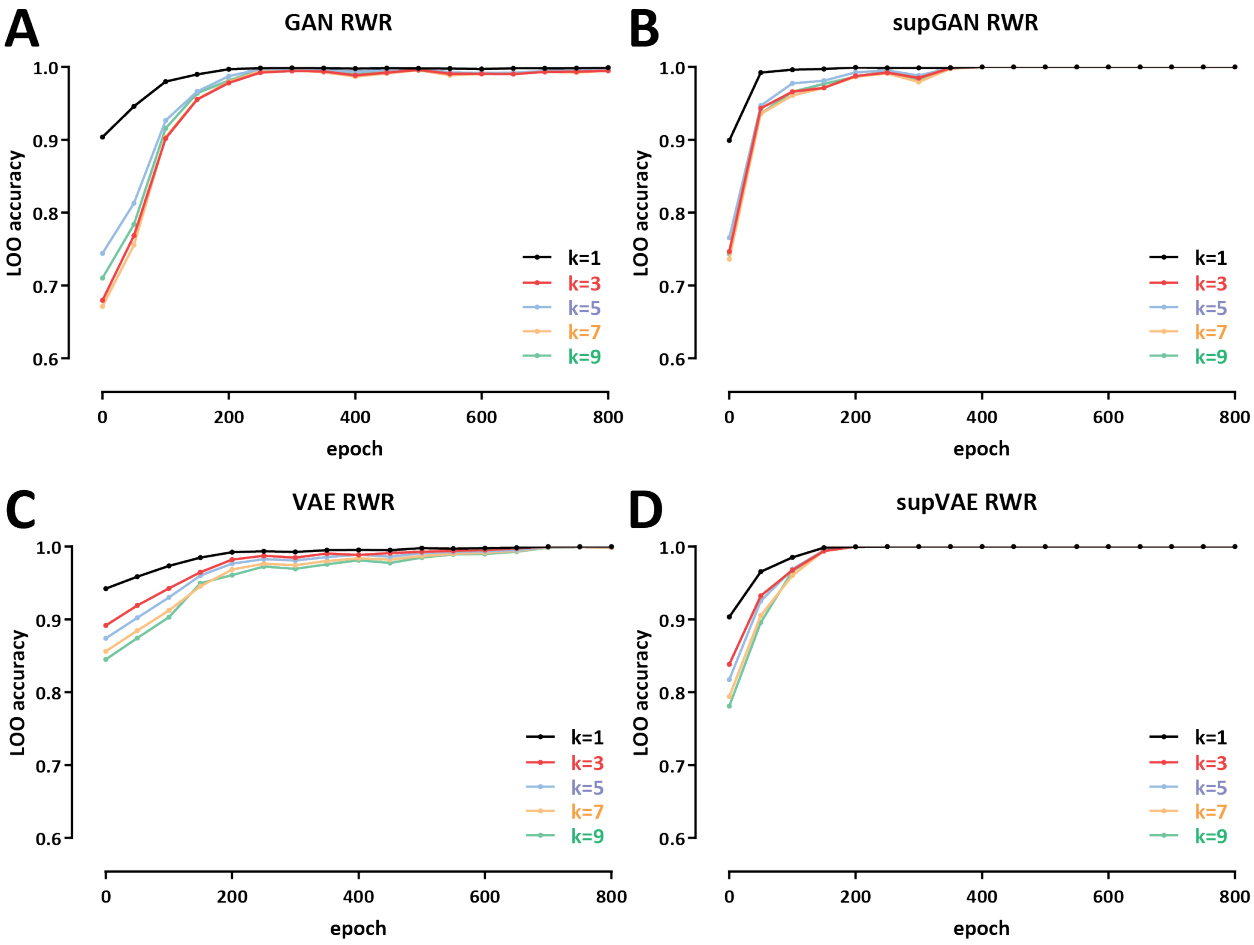

Figure 33. **Nearest neighbor evaluation of sequences generated using the RWR tunning method**. (A) Generative Adversarial Networks (GAN) (B) Supervised Generative Adversarial Networks (supGAN) (C) Variation AutoEncoders (VAE) (D) Supervised Variation AutoEncoders (supVAE)

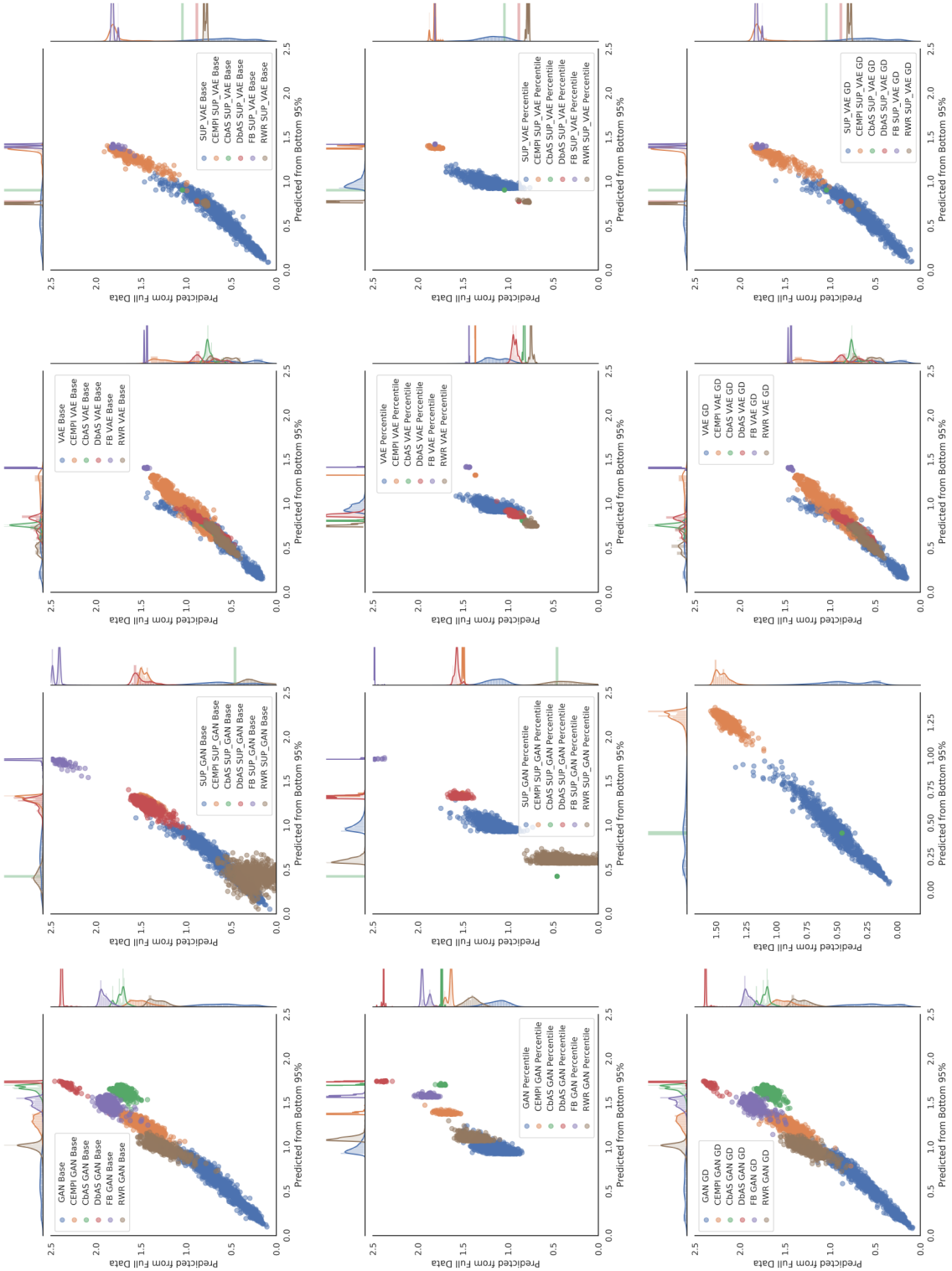

Figure 34. **Predicted expression values and distributions for sampling methods.**

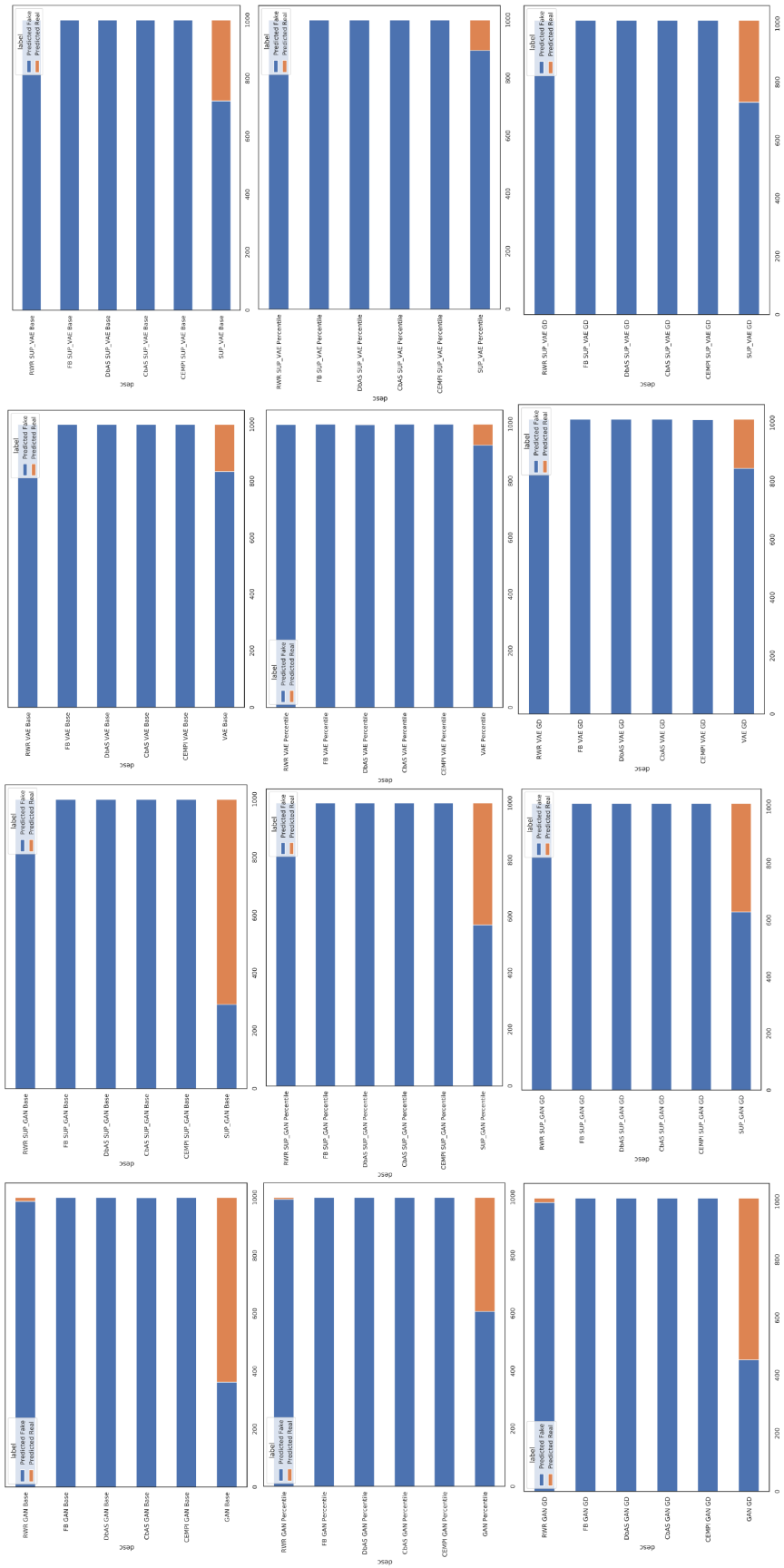

Figure 35. **1NN predictions for sampling methods.**