Representation learning for biological sequences

Kasper Munk Rasmussen <wdq486@alumni.ku.dk>

November 6, 2020

Projekt udenfor kursusregi, blok 1, 2020. Københavns Universitet, Datalogisk Institut. KU-ID: wdq486. Vejleder: Anders Krogh.

Contents

1 Introduction

In the paper [3] from 2001, the Human Genome Project announced the sequence of the human genome. While almost all human beings have different DNA sequences, they are still very similar, and therefore it is possible to produce a reference for a generic human being.

To gain deep understanding of the genome, it is necessary to understand the properties and functionality of different regions within it. Thus, a large part of the analysis of sequenced genomes is the annotation of subsections of the genome. ENCODE [5] is an example of a large project which aims to form an encyclopedia of DNA elements. Another related aspect of analysis of the sequences is the comparison and alignment of different sequences which can elucidate evolutionary and functional relationships between sequences.

This project uses human genome data in the context of unsupervised machine learning, specifically representation learning. This means that the sequence data is processed without using external labels or information about the different regions. The goal of representation learning is to take data, in this case sequences, and transform them into structures which capture meaningful and essential information about the data, identifying features which are not only relevant to one specific task [11]. More specifically we want to transform short sequences of DNA into vectors in $\mathbb{R}^d$ which we call the representation space. We call the vectors in this space *encodings* or *representations*. Ideally we would like representations where sequences which are similar in terms of biology would also have representations which are close in representation space.

2 Data sets

From the UCSC Genome Browser [4] it is possible to download human genome sequence data by chromosome. In this project we focus mostly on chromosome 22, and we use the newest version which is *hg38* from 2013. The chromosome is stored in the FASTA file format which we process using the Biopython package for Python [9].

The chromosome 22 is a sequence of ≈ 50 million symbols from $\{a, c, g, t, n\}$ which correspond to the 4 nucleotides and n is used to "represent a sequence gap or unannotated regions" where the nucleotide is not known [6]. The number of n's is substantial and we will often process the sequence by removing longer streaks of pure n's. For example, when substituting streaks of length 79 with nothing using a regular expression, the length of the sequence is reduced to ≈ 39 million.

2.1 Window data set

In many machine learning problems the *data set* is viewed as a set of data points or samples. For use in the neural network experiments we define our data set

$D_w(m)$ as the set of all the *windows* of length w which can be attained from the genomic sequence m after removal of streaks of n symbols.

That is, the chromosome can the viewed as sequence $m_1 m_2 ... m_n$ with $m_i \in \{a, c, g, t, n\}$. We then define the window of length w starting at position j be the sequence $m_j, m_{j+1} ... m_{j+w-1}$. This means that $D_w(m)$ consists of ≈ 39 million windows, when m is chromosome 22.

Instead of simply viewing a window as a sequence, we represent the window as a a $4 \times w$ matrix (tensor): For each symbol in the window, we produce a 4-dimensional *one-hot* encoding, where the one-hot encoding for n is the $\mathbf{0}$ vector. We also have a setting where n is represented as $\langle 0.25, ..., 0.25 \rangle$ but have not used it. The one-hot encodings become columns in a tensor/matrix of shape $4 \times w$. We will also use the notation $(4, w)$ for a tensor of shape $4 \times w$.

When using this data set we reserve $4,000$ windows as the validation set and $10,000$ windows as the test set. These windows are chosen uniformly at random. However, we note that the validation and test sets are not fixed across experiments, which would be preferable.

2.2 Sentence data set

Later we will use a method from natural language processing where the data is viewed as a corpus of text consisting of sentences of words. To use this method we define a data set structure $S_{k,d}(m)$ as a set of "sentences" of d words where each word is a k-mer, that is k subsequent nucleotides. m is the sequence. We move a window of length $k \cdot d$ through m, and we use stride $k \cdot d$. For each window we produce a sentence by splitting the window into d small k-mers. We represent the sentence simply as a list of the k-mers. With the values of d and k we use, we end up with ≈ 39 million sentences, and the maximum number of distinct k-mer words will be 5^k, since we have 5 different symbols.

3 Autoencoders

An autoencoder can be described as a function $h_\theta : \mathbb{R}^w \to \mathbb{R}^w$. The autoencoder consists of two parts, an encoder $f_\theta : \mathbb{R}^w \to \mathbb{R}^d$ and a decoder $g_\theta : \mathbb{R}^d \to \mathbb{R}^w$, so that $h(\mathbf{x}) = g(f(\mathbf{x}))$ [13]. We use θ to denote all the parameters for a neural network implementing h. With $d < w$ the output of the encoder $\mathbf{z} = f(\mathbf{x})$ is then an *encoding* of $\mathbf{x}$, a lower-dimensional representation.

The aim is to find the parameters θ which maximises the ability for the autoencoder to *reconstruct* the information of the window from its representation, so that the input $\mathbf{x}$ and the output $h(\mathbf{x})$ are similar. The idea is then that the representation layer will be a "bottleneck" where the information is compressed, condensing the most relevant information about the input.

In our case the input $\mathbf{x}$ and output $h_\theta(\mathbf{x})$ will both be tensors of shape $(4, w)$. The output of the autoencoder is a reconstruction $(4, w)$ tensor where the four values in column i can be thought of as parameterising a distribution for a discrete random variable $Y_i \in \{a, c, g, t\}$. This means that the autoencoder

outputs can be viewed as parameters for a probability distribution over the set of possible windows, at least when ignoring n symbols. We then let $\mathbf{Y}$ be the random vector derived from the autencoding of the window $\mathbf{x}$. If we make an assumption of independence we can then score how well the autoencoder reconstructs the window as

$$P(\mathbf{Y} = \mathbf{x}) = \prod_{i}^{w} P(\mathbf{Y}_i = \mathbf{x}_i)$$

When doing this for the whole data set, the autoencoding becomes similar to a form of maximum-likelihood

$$\arg\max_{\theta} \prod_{\mathbf{x} \in D_w(m)} \prod_{i}^{w} P(\mathbf{Y}_i = \mathbf{x}_i)$$

Because our data set is so large we do not train on the whole data set. We perform stochastic gradient descent using *mini-batches* of 32 windows and use Adam to to minimize the negative log-likelihood using softmax cross-entropy which takes care of logarithms and softmax [14]. For every 500 mini-batches ($16,000$ windows) we evaluate on the validation set. We call such an interval of 500 mini-batches a "pseudo-epoch".

For purposes of interpretation of performance we also use what we call *mean nucleotide probability* MNP, the average of the probabilities given to what is in fact the true nucleotide. Since in the best case we would perhaps give full probability to the correct nucleotide, the best MNP is 1. Empirically, MNP is almost perfectly negatively correlated with the loss (less than -0.99).

Looking at $500,000$ windows sampled from the chromosome, the following is a simple distribution over each symbol.

$$(a, c, g, t) = (0.265, 0.234, 0.236, 0.265)$$

Also, for each of the each of the w positions the entropy is 1.99 bits, which arguably means that we can compare our results to a baseline MNP score of 0.25.

3.1 Convolutional autoencoders

Convolution layers are neural network layers which have proved useful for image data, because they can capture local features independetly of its location in the image. 1-dimensional versions of convolutional layers also exist, and are attractive to our use case because of the sequential local structure of the windows, meaning that the order or the nucleotides in a window does matter.

A 1-dimensional convolution layer consists of a number of kernels k, all with the same length of d, and stride of s. We move along the column dimension of the tensor with a window with the same number of rows and d columns. At each step we compute the dot product. We do this with each of the k kernels, and the new row dimension corresponds to the number of kernels while the length

of the column dimension depends on the stride and length. The entries in the kernels are parameters that are learned using optimization [15].

We use ReLU as activation function. While max-pooling has been effective for image data, we shortly experimented with max-pooling and did not find it to be useful, although it could be worth looking further into.

In the decoder we want a reverse process. As a reverse of the convolutional layers we have *transpose convolutions*, also called "fractionally-strided convolutions". Here the kernel with its length and stride moves across the resulting output tensor and adds values gained by moving with stride 1 along the input tensor and scaling the kernel entries by the current input [16].

3.2 Experiment 1: Convolution and dense layers

In this experiment we have an architecture "WithDense" where the encoder has two convolution layers followed by a dense layer. Convolution 1 has kernel size 3 and stride 1, the next has kernel size 5 and stride 1. We use ReLU activation. The number of filters are 8 and 4, so a single sample has the following shapes through the encoder.

$$(4 \times w) \to (8 \times w - 3 + 1) \to (4 \times w - 8 + 2) \to \text{flatten} \to (1 \times d)$$

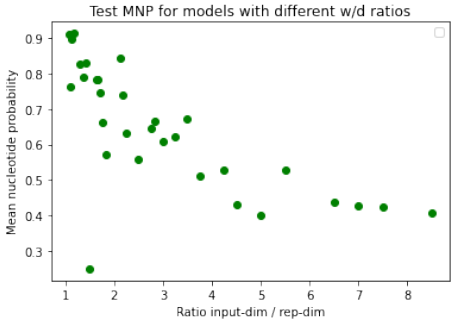
After flattening we use a fully-connected linear layer to the representation space. Here ReLU is not used, since we want to make it possible for representations to have negative entries, and we find that it does not significantly impact the performance. The decoder has the reverse series of shapes. We also experimented with multiple dense layers but this did not improve performance, or at least training was very slow.

We trained a large set of models, varying window size and representation dimension. Learning learning rate was 0.001 and we stopped after 10 pseudo-epochs without improvement in validation loss.

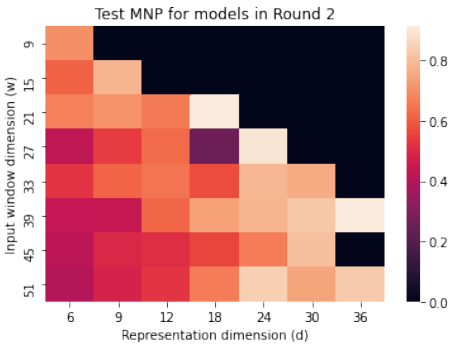
In Figure 1b we show a heat map where the axes are window length w and representation dimension d and the color describes the MNP. As expected, increasing representation dimensionality generally increases the MNP. In Figure 1a we look at the ratio $\frac{w}{d}$ between window length w and representation dimension d. This is a rough measure of how much the dimensionality has been reduced. Since the input window consists of one-hot encodings $4 \times w$ and the encoding is in $\mathbb{R}^d$, it can be hard to compare. However, we have found that there exist autoencoders with $d = w$ which attain MNPs very close to 1, indicating that it is reasonable to compare w and d, since apparently, w floats are sufficient for almost lossless encoding of the window. The plot shows that when the ratio increases the MNP is falling. We also see some level of variability in the MNP supposedly resulting from differences in network initialization and training data.

In Figure 2 we show training curves showing a high rate of improvement early on and then only improving slowly. We have found that there training curves for different models with the same values of w and d can vary lot in terms of time before reaching the stage of slow improvement.

In other machine learning prolems where we have smaller data sets and train for multiple epochs, the network will often learn so much about the training data that the training loss will become lower than the validation set. In our case, while distinct places in the chromosome could contain windows which are identical, the network will not generally see the same training examples again.

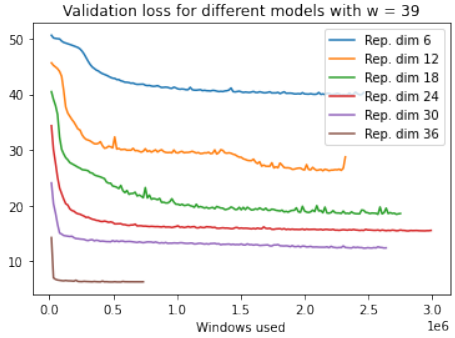


(a)

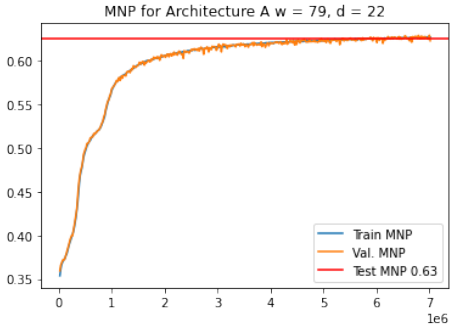


(b) The number of models trained is limited by the compute available from Google Colab.

Figure 1: Experiment 1



(a)



(b)

Figure 2: Experiment 1

3.3 Purely convolutional Autoencoders

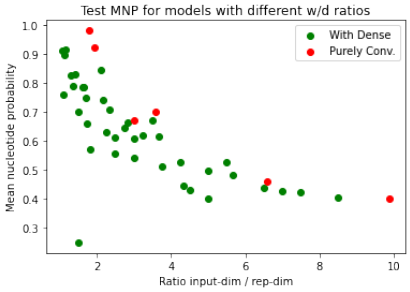
We try another architecture "PureConv" where only convolutions and transpose convolutions are involved. Dimensionality reduction occurs by using stride-2, thus roughly reducing the dimensionality by a factor of 2 for each layer. This system provides no easy way of allowing arbitrary values of w and d, and so

only a few combinations have been trained. One examples of the shapes of a sample going through the encoder with $w = 79$ and $d = 22$ is the following.

$$(4 \times 79) \to (16 \times 79) \to (16 \times 41) \to (1 \times 22)$$

Padding is used in order to make the reverse set of shapes through the decoder equivalent to the above shapes.

In this example, we say that the ratio $\frac{w}{d} = 79/22 \approx 3.6$. We train 6 models, where we also use different number of layers, giving different ratios w/d. The MNPs for the different architectures can be seen in the table of Figure 3b. These models were trained with learning rate 0.01 and with this learning rate the success of the network was very dependent on the intitialisation of the parameters: At least half of the times, the MNP would not improve beyond 0.25. Later, when getting more experience with neural networks, we found that decreasing the learning rate to 0.001 yielded more consistent results. A smaller learning rate also slightly increased the best MNPs attained, but not by much. Another change which is discussed below is that at first we had ReLU activation on the last layer mapping to the representation which we later removed without decrease in MNP scores. Figure 3a shows that these autoencoders perform at least as well as the previous architecture.



(a)

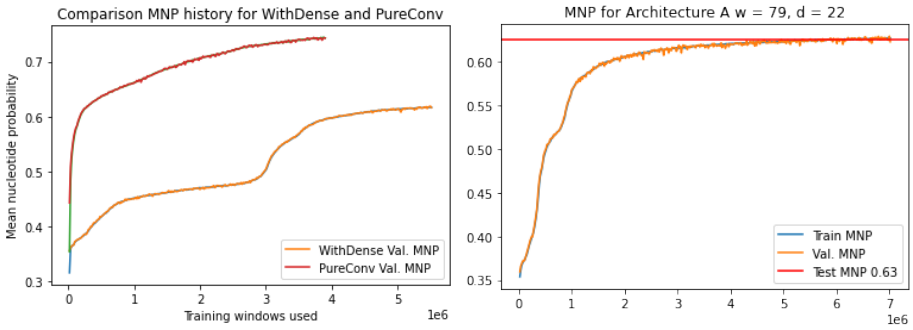
w	d	M.N.P.
27	9	0.67
27	15	0.98
81	42	0.92
79	22	0.70
79	12	0.46
79	8	0.40

(b) MNPs for Purely Convolutional autoencoders

Figure 3

These purely convolutional autoencoders contain fewer parameters than networks using fully connected layers, so the parameter space we are searching in is smaller. Convolutional networks could be "simulated" by dense networks where some weights are set to 0. In Figure 4 we compare PureConv and WithDense with $w = 79$ and $d = 22$ and find that PureConv is faster to train and archieves a higher score. We note again however, that the training curves for WithDense can vary widely as can be see when comparing with the training curves in Figure 4.

The way convolutions work has implications for our representations. Because of the locality, even after multiple layers of convolutions, the entries will still to some degree represent information about local areas in the window. The depth of the network and kernel sizes determines how much "diffusion" of information



(a)

(b) Alternative faster training curve for WithDense

Figure 4: Comparison of training curves for two architectures

there is across the components of the vector. This is of course not a problem for the decoder, but ideally we would want representations which contain abstract global features, and not simply a reconstruction-relevant compression. Perhaps this indicates limitations of the autoencoder approach to learning of representations.

3.3.1 Evaluating the autoencoder

To evaluate the autoencoders we use models where $w = 79$ and $d = 22$ with MNPs above 0.7. We sample $n = 10,000$ windows uniformly at random from the chromosome and encode the windows using the encoder part of the autoencoder. We then apply the non-linear dimensionality reduction technique UMAP to attain a scatter plot of the representations.

Figure 5 shows PCA and UMAP plots of the encodings for a network trained with learning rate 0.01 and using ReLU activation in the last encoder layer, where clusters are visible. When removing the ReLU activation function this cluster structure almost disappears and in Figure 6 we show PCA and UMAP plots for a network with no ReLU in the last encoder layer and a learning rate of 0.001 where no clusters are apparent. Since ReLU activation results in representations where negative entries are set to zero it seems most reasonable to assume that the clusters are not biologically meaningful.

From UCSC Table Browser we attained a tabular file specifying coding regions on chromosome 22. By processing this data we can determine whether a window is coding or not, if it is within an exon and within the coding region. Roughly 1% of sampled windows will be coding. These windows are colored yellow, and looking closely, both clusters in the plots above contain coding windows.

Further arguments for viewing the clusters as merely technical are that when looking at individual windows in the small cluster with UCSC Table Browser no

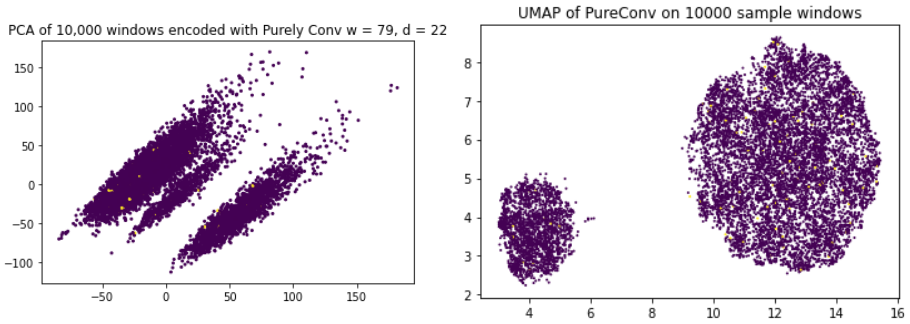


Figure 5: Encodings when network has ReLU activation in last encoder layer

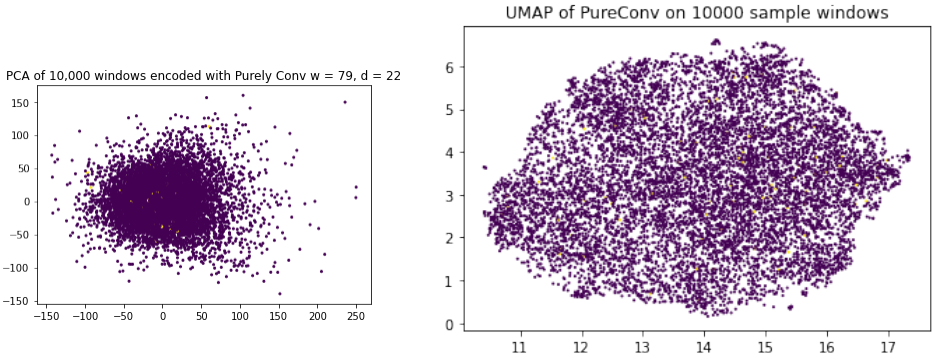


Figure 6: Encodings when network has no activation function in last encoder layer

obvious pattern stands out. We also notice that the windows in the small cluster are relatively uniformly distributed across the chromosome in terms of location. Models with WithDense architecture, same w, d and MNP performance likewise give plots like in Figure 6.

3.4 Convolutional Variational Autoencoder

We use settings from the purely convolutional encoder with $w = 79, d = 22$ but modify the architecture and the loss function in order to turn it into a variational autoencoder [10]. In variational autoencoders, the encoder outputs parameters for a distribution which approximation of the posterior $p(z|x)$, where z is a vector of hidden variables and x are the observed data, in our case windows. This approximation is called the variational distribution $q_\phi(z|x)$. A prior distribution $p(z)$ is defined on the hidden variables. The decoder is a generative

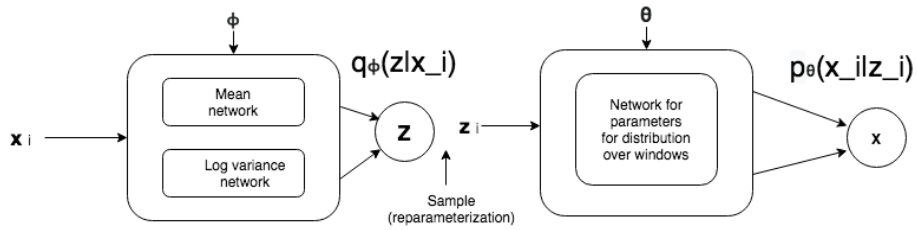


Figure 7: Diagram of variational autoencoder

distribution $p_\theta(x|z)$. The variational autoencoder maximises the evidence lower bound ELBO:

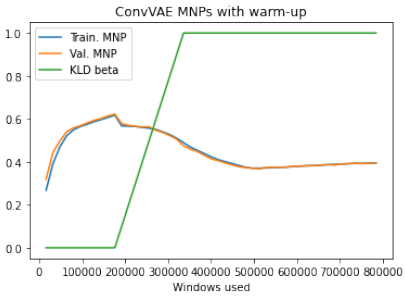
$$\text{ELBO} = \mathbb{E}_{q(z|x)}[\log p(x|z)] - D_{KL}(q(z|x)||p(z))$$

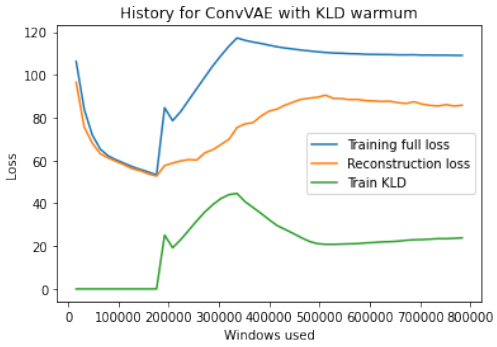
As proposed in [10], we let $q_\phi(z|x)$ be a factored multivariate Gaussian and the prior $p(z)$ be the standard multivariate Gaussian. The expectation is approximated using sampling using the "reparameterization trick" which allows for backpropagation. We take a window $\mathbf{x}$ forward it through to get $q_\phi(z|x)$, make a single sample, forward it through the decoder to compute $\log p(x|z)$ using the log-likelihood loss we already used (cross-entropy). The KL-divergence term can be computed analytically as

$$D_{KL} = -\frac{1}{2}\sum_{i=1}^{d}(1 + \ln(\sigma_i^2) - \mu_i^2 - \sigma_i^2)$$

To the encoder we add two layers mapping what was previously the final encoding $\mathbf{z} \in \mathbb{R}^d$ to a mean vector and a log-variance vector. Both of these encoders need to be linear, so that the means and log-variances can be negative. To the decoder we add a linear layer between the sample and the original decoder.

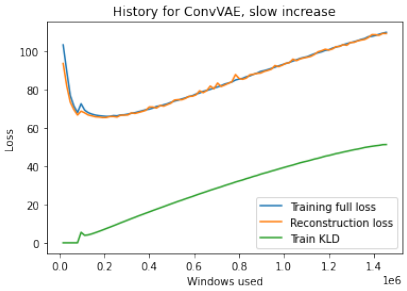
We train with a warm-up so that the KL-term is weighted by $\beta = 0$ for 20 pseudo-epochs and then β is increased by 0.1 for each pseudo-epoch until reaching $\beta = 1$. We find that when starting KL-activation the MNP falls from 0.6 to 0.4 and only improves very slowly after $\beta = 1$. We can choose to only increase β by 0.01 for each pseudo-epoch, then the number of windows used before $\beta = 1$ will is much larger. We then find that we can increase β gradually to 0.5 without MNP falling, but beyond that MNP will begin to fall.
One reason why it would be interesting to get the variational autoencoder to work, is that the KL-term encourages that the variational distribution has low divergence from a standard multivariate normal, where the hidden variables are independent. So called β-VAEs use $\beta > 1$ to further encourage this, and supposedly leads to disentanglement of the factors of variation which is arguably at the core of representation learning [12] [11].





(a) Here the orange curve is the validation loss without including KL-loss.

Figure 8



(a) The orange curve is wrongly labeled "reconstruction loss", it should be "validation loss".

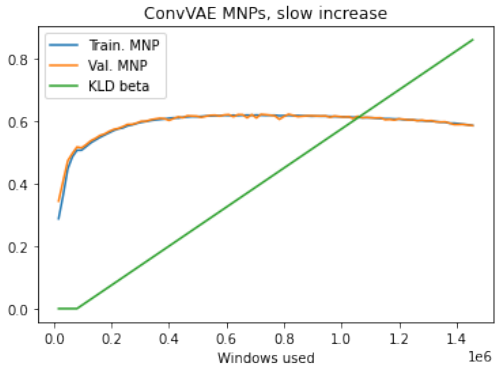


Figure 9

4 Smooth trajectories

When producing vector representations of windows, it might be beneficial for downstream tasks, if the representations where such that moving with window of length w through a genome with a low stride, fx stride-1, would result in the sequence of vector representations forming a "smooth trajectory" in $\mathbb{R}^d$. We note that the trajectory is of course discrete.

For finite vector sequences $\mathbf{a}_i$, $i \in \{1, ..., n\}$ and $\mathbf{b}_j$, $j \in \{1, ..., m\}$ we say that $\mathbf{a}_n$ is smoother than $\mathbf{b}_n$ if the average of Euclidean distances between $\mathbf{a}_i$ and $\mathbf{a}_{i+1}$ is lower than the average distances between $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$. However, as we will see this way of looking a smoothness ignores aspect of scaling. In some sense, we would think that if $b_i = ca_i$ where c is some scaling constant, then

b_i is just as smooth as a_i. In this case we can use cosine similarity, since this involves a normalization of the vectors.

Using a purely convolutional autoencoder with $w = 79$ and $d = 22$ with test MNP at 0.70 we investigate smoothness. We select a random start position in the chromosome (where streaks of n's are removed) and move with stride 1 giving $10,000$ windows and encode each window. We look at the distances between encodings for adjacent windows and find that the mean Euclidean distance is 45.9 and if scaling the vectors by some constant $c = 50$ the mean Euclidean distance becomes roughly $c = 50$ times as large 2296.2. We get mean cosine similarity of 0.39.

For comparison we sample $10,000$ windows i.i.d. and look at distances between adjacent windows. We get a mean Eucldiean distance of 60.64 and a mean cosine similarity of 0.19. This could indicate that the first trajectory is a bit smoother, however when moving with stride equal to the window length so that there is no overlap, we get mean Eucldiean distance 58.85 and mean cosine similarity 0.18, so the higher lower distance and higher similarity has something to do with the overlap and the network does not give smoother trajectories than a random one, at least only marginally.

In the following we describe experiments where we include additional terms to the training objective with the aim of producing representations which give smooth trajectories.

4.1 d-loss

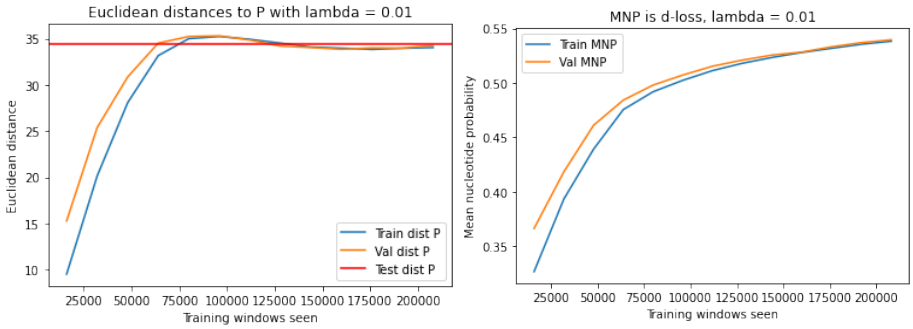
In this experiement, we use the purely convectional architecture with 79/22. We take a window a and an adjacent window p (there is no overlap between the two windows). The encodings are $a' = f_\theta(a)$ and $p' = f_\theta(p)$. If the trajectory is to be smooth, we somehow want the distance between a' and p' to be small.

Thus we add to the loss a term $\lambda \cdot d(a', p')$ where d is the Euclidean distance and λ is a hyperparameter specifying the importance of the distance aspect. We call this new loss the d-loss. The reconstruction loss is now the mean of the cross-entropy loss for a and p.

When $\lambda = 0$ we have a purely convolutional autoencoder, and we already know this can attain an MNP of 0.7 but when $\lambda = 1$ no improvement of MNP beyond 0.25 occurs: The dynamic is that that $d(a', p')$ converges to 4.6904e-06 after training the second minibatch of 32 windows. For $\lambda = 0.1$ a similar dynamic happens, for $\lambda = 0.01$ the MNP rises slowly but the distances are growing instead of diminishing, the same goes for $\lambda = 10^{-5}$. The distances are supposedly rising because the reconstruction is given greater relative importance.

4.2 d-loss with warm-up

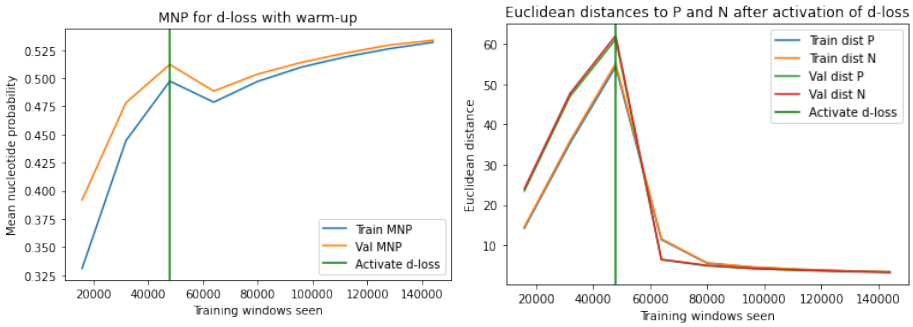
With "warm-up" we train using reconstruction loss only, that is $\lambda = 0$. Then when validation MNP passes 0.5 we set $\lambda = 1$. Figure 11 shows how the MNP will increase after activating the distance term and how the distances



(a) (b)

Figure 10

will fall. However in these graphs instead of including only $d(a', p')$ we also feature $d(a', n')$ where n' is the encoding of a "negative" sample window n sampled from a uniform distribution over all the windows in the chromosome. The figure shows that while distances to adjacent window falls so it does for a randomly chosen window which is contrary to our desired outcome.



(a) (b)

Figure 11

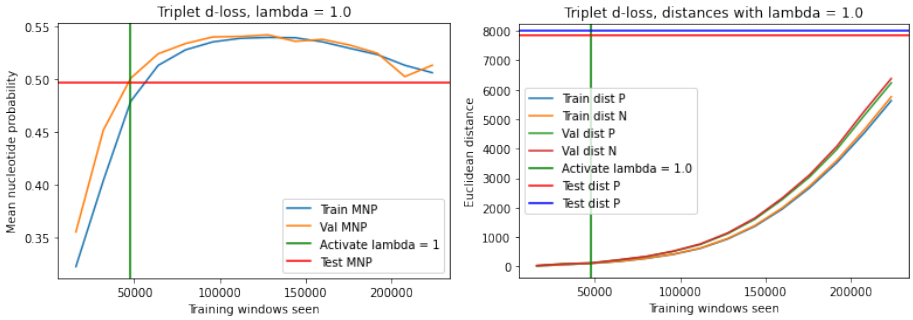
Supposedly, the dynamic is that the network will scale the representation vectors to have smaller norms while maintaining the structure.

4.3 Triplet d-loss

Given that the dynamic we just presented is undesired we experiment with a what we call a triplet d-loss where the extra term is

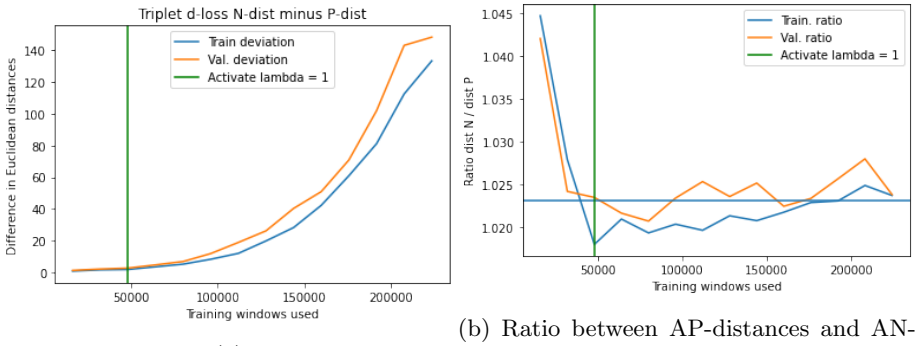
$$\lambda \cdot (d(a', p') - d(a', n'))$$

here the network is penalized if the distance to remote windows is smaller than distance to adjacent window but without applying $\max(0, \cdot)$ this term can contribute negatively to the loss meaning that the total loss can become negative. The reconstruction loss is the mean of the cross entropy for a, p, n. We find that the AP and AN distances increase and that the difference (AN distance minus AN distance) also increases. However, looking at the ratio AN distances to AP distances there is not much improvement. Thus we assume that a form of scaling is occurring, where representations generally have larger norm giving larger differences.



(a) (b)

Figure 12



(a)

(b) Ratio between AP-distances and AN-distances.

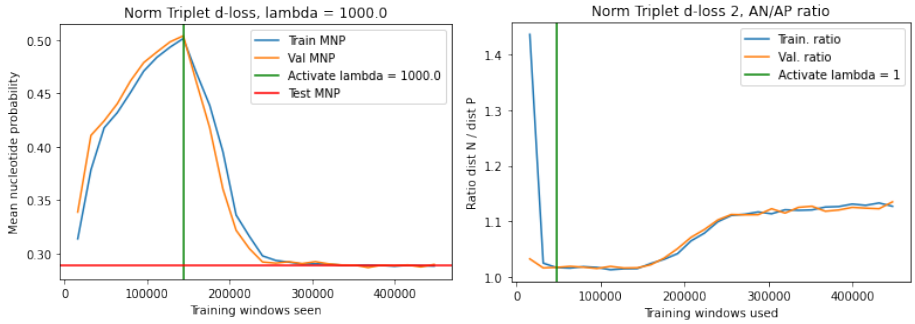
Figure 13

4.4 Triplet d-loss with normalization

To avoid the dynamic where representations are scaled to increase the difference between AP and AN distances we experiment with normalization of the representations before computing the triplet d-loss $\lambda(d(a', p') - d(a', n'))$ with

Euclidean distance function. When normalized, distances will become smaller, thus we use a large $\lambda = 1000$ chosen to give enough importance to the triplet d-loss. We active the normalization triplet d-loss when validation MNP exceeds 0.5 and the MNP continues to rise but the ratio does not increase, staying at 1.03 before and after activation.

Previously, I thought that normalization of the representation within the autoencoder, that is, decoding of normalized representations, hindered reconstruction. It is however possible to get MNP improvements with normalization autoencoder without any extra loss term. Therefore we also try to do train with normalization and $\lambda = 0$ and activate $\lambda = 1000$ when MNP exceeds 0.5. In Figure 14 we see that the MNP falls but the ratio rises, with test AP/AN ratio being 1.13 which means that the AN distance is 13% larger than AP distances. Since the MNP does not fall to 0.25 we also try to train with only normalized triplet d-loss, without reconstruction, this also leads to test AP/AN ratio on 1.13.



(a) (b)

Figure 14

4.5 Triplet loss

We will use the idea of *triplet networks*, which fx have been used for face identity recognition [17]. Here the anchor a can be a picture of some person (fx Obama) and the postive p is a picture of the same person (also Obama) and the negative n is a picture of another person (fx Macron). The triplet refers to the triplet of a, p, n. In triplet networks we want representations where a and p are close while a and n be more distant.

Thus, the goal is to have

$$d(a', p') \leq d(a', n')$$

These distances should be separate by some margin α, and by moving $d(a', n')$ to the left side of the inequality we want representations fulfilling this inequality

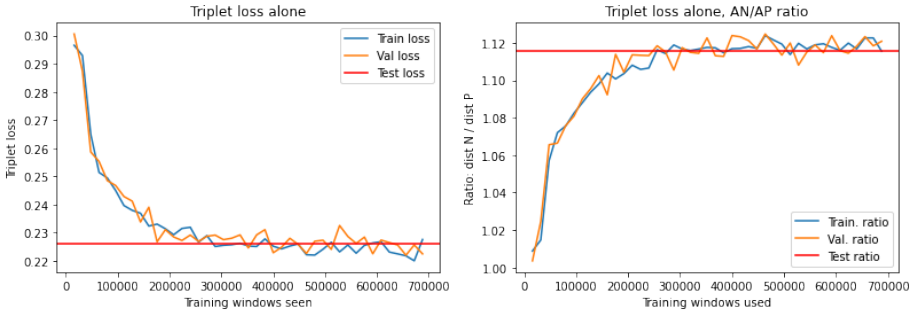
$$d(a', p') - d(a', n') + \alpha \leq 0$$

Based on this, the extra term to the triplet loss (t-loss) is

$$\lambda \cdot \max(0, d(a', p') - d(a', n') + \alpha)$$

The idea of using max is that we only want to penalize if the inequality does not hold.

We use the normalized representations without using reconstruction with a margin of $\alpha = 0.3$ and get a similar result to the triplet d-loss on normalized representations, a test AN/AP ratio of 1.115. I have not succeeded in finding a way of successfully combining reconstruction and triplet loss.



(a) (b)

Figure 15: Triplet loss for normalized representations without using reconstruction

5 Word2Vec

Word2Vec is a method used in natural language processing (NLP) presented by [1]. Given a corpus of text, Word2Vec learns vector representations for each word. Word2vec and similar methods are based on the distributional hypothesis of semantics, which is related to idea that "You shall know a word by the company it keeps". If we can train a network which given a one-hot encoding of a word w outputs a probability distribution $P\theta(c|w)$ giving the probability of word c occuring in the context of the word w, then this is one interpretation of having learned the semantics of w. This is the "skip-gram" version of Word2Vec.

Word2vec has two matrices, one with with vector representations for centre words w and one with representations for context words c. For a given context word, dotting the vector representation of w with context vectors as a measure of similarity and applying softmax can yield one such probability distribution $P_\theta(c|w)$. Because of the computational cost of doing this, other objectives can be used: Some are more similar to a classfication task, where given a centre-context pair (w, c), the network should learn to classify such a pair correctly as either a real pair or a fake pair, where a fake pair can be attained by sampling the context word c from the vocabulary using some distribution [8].

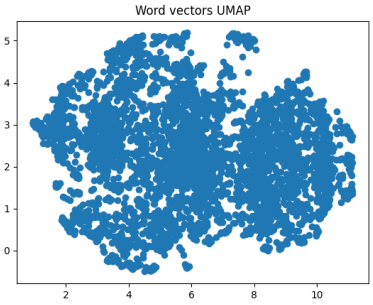


Figure 16: UMAP of word vectors with $k = 6$

Section 2.2 describes how we will choose to view the chromosomes as text corpora, viewing k-mers as words and sentences a lists of length d of k-mers. Since practical implementations of Word2Vec can be quite involved, we use an existing package called Gensim which implements a version of Word2vec [2].

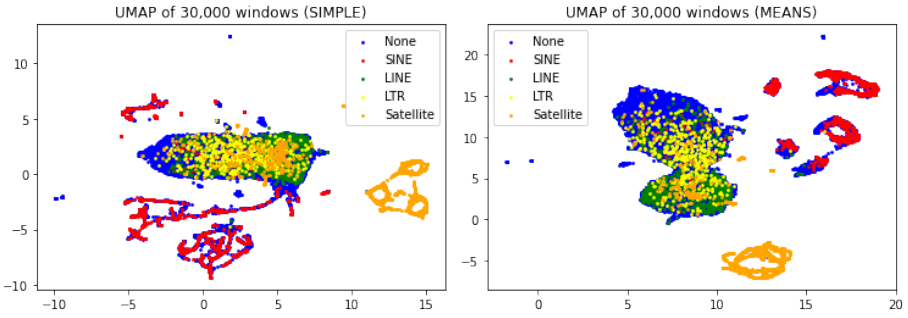
5.1 Word vectors from chromosomes

We train on chromosomes 20, 21, and 22 with $k = 6$ and $d = 15$. The training is performed in 5 epochs, total training time is around 20 minutes. The output is a model containing word vectors in $\mathbb{R}^{100}$ for all the k-mers. The UMAP of these word vectors is presented in Figure 16, and while some structure is visible, it does not seem that there are any parts specifically worth investigating.

From chromosome 22 we sample $30,000$ windows of length $d \cdot k$, split the window into d different k-mers and get the associated d word vectors and then sum them. This means that for each sampled window we have a representation in $\mathbb{R}^{100}$. The PCA plots of these representations are not interesting, but in 17 we present scatter plots of the UMAP of the representations where clusters appear. Looking manually at these clusters we found some clusters associated with repeat elements. From the UCSC Table Browser we get a table containing information about repeat areas, and use this to color the points in the plot if the window is within some region. A few hundred points belonging to minor repeat classes have been left out. In 17b we do not only encode the $d \cdot k$ window but compute the $k - 1$ representations derived by skipping initial nucleotides and average these representations. We see a cluster with many Satellite repeats and some clusters with many SINE repeats. When using the averagin approach the large cluster is divided into two areas, and the LINE repeats are present only in one of these two areas.

We also tried $k = 3$ where no structure in UMAP was visible, and $k = 9$ which looked similar to $k = 6$ but with more outliers. The best choices of k will supposedly depend on the amount of training data, since we want to have k-mers occur relatively frequent for Word2Vec to work.

That the cluster structure apparent in the plots are in particular related to



(a) (b)

Figure 17: Encodings of samples from chromosome 22 colored by repeat class.

repeats can perhaps be understood by looking at the Word2Vec method: Since repeats occur multiple times in the chromosome, the k-mers occuring within the repeats will occur most often within the context of other k-mers in the repeats. When adding the word vectors together they would cluster together.

It could perhaps be interesting to explore encodings where not all k-mer vectors are given equal weight.

5.2 Alignment

Sequence alignment can be used to understand the relationship between sequences. In pair-wise sequence alignment we have a pair of sequences $x = x_1x_2...x_n$ and $y = y_1y_2...y_m$ and want the symbols in the sequences to align by inserting gaps. The alignment is scored by some objective function which rewards matching symbols and penalizes mismatching symbols and gaps. Alignment can be understood within the context of evolution where mutations, insertions and deletions in DNA sequences happen over time.

Smith-Waterman is a dynamic programming algorithm which can be used to find a *local* alignment of x and y by constructing at $n+1 \times m+1$ matrix $\mathbf{C}$ and filling in the entries of in $\Theta(nm)$ time by making a $O(1)$ computation in each step. The first row and first column are filled with zeros and the rest of the entries are filled using the rule

.

$$\mathbf{C}[i,j] = \max \begin{cases} \mathbf{C}[i-1,j] + m \\ \mathbf{C}[i,j-1] + m \\ \mathbf{C}[i-1,j-1] + s(x_i, y_j) \\ 0 \end{cases}$$

Here m is the gap penalty and $s(x_i, y_j)$ is a function which gives the reward or penalty for aligning x_i with y_j. Alignments can be attained from the matrix by tracing backwards from a entry with greatest value until reaching 0. The

greatest value in the matrix can also be used as a measure of how well the sequences can be aligned.

We want to evaluate the use of k-mer word vectors for alignment. Given x and y we truncate them from the right so that k divides the lengths of x and y and then split x and y into k-mers. Now the problem becomes one of aligning sequences of vectors instead of sequences of symbols. For the scoring function s we propose the cosine-similarity which is in $[-1, 1]$.

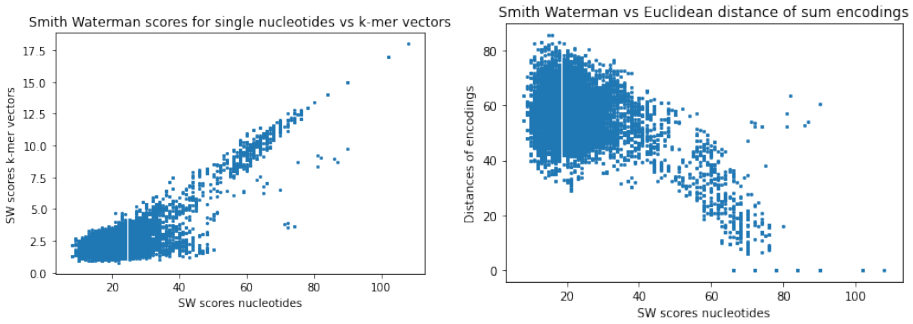
We have attained a FASTA file with 432 tRNA sequences from the human genome from UCSC Genomic tRNA database. We select the tRNA sequences on chromomsome 6 and 17. There are 187 such sequences and they have lengths 7l-107. We perform alignment of these sequences using both Smith-Waterman on the sequences of nucleotides with setting (match: 1, mismatch: -1, gap: -1) the "nucleotide-SW" and Smith Water with the just mentioned k-mer vector version with $k = 6$, the "vector-SW". These settings for nucleotide-SW might not be optimal, and if changing them, we would also need to change the vector-SW scoring system to ensure that the ratios between scaling gap penalty and match/mismatch scoring s are matching for the two methods (Anders Krogh, private correspondence). We also tried settings (match: 1 mismatch -2, gap: -2) for nucleotide-SW and vector-SW with gap -2 and a scoring function with negative cosine similarities are scaled by a factor 2 so that the lowest score is also -2. This gives very similar results.

In the scatter plot of Figure 18a each point is an alignment of two distinct sequences and we compare the alignment scores from nucleotide-SW and vector-SW. We find that there is a relatively high correlation (0.89) and that the correlation is stronger when just looking at the alignments where the nucleotide-SW is high. However, it should be noted that some of the tRNA sequences are very similar, and so this might not generalize. Further work could look into the individual alignments, especially the alignments which appear as outliers on the scatter plots.

In Figure 18b we also compare nucleotide-SW alignment scores to the Euclidean distances of the encoding of the sequences, when a sequence is encoded by summing of the k-mer vectors. The correlation is -0.66. If there was a strong connection here, then the advantage would be that comparison of Euclidean distance is much faster to compute than Smith-Waterman score.

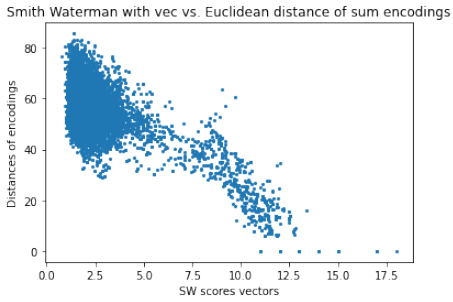
6 Discussion and conclusion

Unsupervised learning, including representation learning, has been successful in the realm of natural language data and image data, where neural network methods scale well with the large amount of data available when no labels are needed. Given that the amount of biological data is growing exponentially and that many species have yet to be sequenced, systems which automatically identify patterns and structure in genomes seem desirable. Given the large set of annotations already existing, there is rich opportunity to evaluate the performance of such systems.



(a) (b)

Figure 18



(a)

Figure 19

In this project we were not able to show that simple autoencoders using convolutional layers were able to identify biologically meaningful aspects about sequences. We found that it was possible to learn vector representations where areas which are close to each other on the chromosome have vector representations which close to each other. We suspect that an approach which only uses unsupervised autoencoders are not sufficient to learn interesting representations, and that a semi-supervised approach where at least some amount of prior knowledge is included in order to guide the system to focus on relevant patterns.

The Word2Vec system which has been successful in natural language processing showed some usefulness when applied to chromosomes, in that it could be used to produce representations where sequences belonging to *repeats* cluster together. The reason why Word2Vec could be used to produce these results and our previous autoencoders could not, is perhaps that the autoencoders look at windows independently of their context, while Word2Vec inherently focuses on context. In general the question of scale is relevant for looking at sequences,

since the biological "meaning" of the sequences exist at certain, possibly multiple, scales, where our focus has mostly been at the scale below 90 nucleotides. This is simply a reflection, as I do not have much knowledge about molecular biology.

We have briefly investigated the possibility of using the vector representation of k-mers from Word2Vec for use in sequence alignment. While it does seem possible use it, it is not clear from this work, that it is any real use. The general idea is that if sufficiently good encodings were attained, perhaps such an approach to alignment could be used to make more abstract alignments taking into account a form of "meaning" not necessarily relationships apparent at the level of nucleotides.

It also seems relevant for unsupervised methods for genomic data to take into account the special nature of this type of data which has been produced or been affected by evolutionary processes. If small sequences such as the windows we have used are to be treated as samples from some distribution, it seems relevant to consider that this underlying distribution is of a very special kind.

References

[1] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.

[2] Řehůřek, Radim, and Petr Sojka. "Gensim—statistical semantics in python." Retrieved from genism. org (2011).

[3] Venter et al. The sequence of the human genome. Science. 2001 Feb 16;291(5507):1304-51. doi: 10.1126/science.1058040.

[4] UCSC Genome Browser. Sequence data by chromosome. Sequence and Annotation Downloads. Retrieved from *http://hgdownload.soe.ucsc.edu/downloads.html* on October 17, 2020.

[5] Dunham, I., Kundaje, A., Aldred, S. et al. An integrated encyclopedia of DNA elements in the human genome. Nature 489, 57–74 (2012). *https://doi.org/10.1038/nature11247*

[6] Guo Yan et al., Improvements and impacts of GRCh38 human reference on high throughput sequencing data analysis, Genomics, 2017, *https://doi.org/10.1016/j.ygeno.2017.01.005*.

[7] Smith-Waterman Algorithm. wikipedia.org. Retrieved from *https://en.wikipedia.org/wiki/Smith-Waterman_algorithm* on November 6, 2020.

[8] Manning, Christopher. Word Vectors I: Introduction, SVD and Word2Vec. CS224n: Natural Language Processing with Deep Learning. Stanford. Re-

trieved from *http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes01-wordvecs1.pdf* on October 23, 2020.

[9] Peter J. A. Cock et al., Biopython: freely available Python tools for computational molecular biology and bioinformatics, Bioinformatics, Volume 25, Issue 11, 1 June 2009, Pages 1422–1423, https://doi.org/10.1093/bioinformatics/btp163

[10] Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." arXiv preprint arXiv:1312.6114 (2013).

[11] Bengio, Yoshua, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives." IEEE transactions on pattern analysis and machine intelligence 35.8 (2013): 1798-1828.

[12] Higgins, Irina, et al. "beta-vae: Learning basic visual concepts with a constrained variational framework." (2016).

[13] Autoencoder. wikipedia.org. Retrieved from *https://en.wikipedia.org/wiki/Autoencoder* on November 6, 2020.

[14] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[15] Conv1D. PyTorch 1.7 documentation. Retrieved from *https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html* on November 6, 2020.

[16] ConvTranspose1D. PyTorch 1.7 documentation. Retrieved fomr *https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose1d.html* on November 6, 2020.

[17] Ng., Andrew. C4W4L04 Triplet loss, November 7. 2017. Retrieved from *https://www.youtube.com/watch?v=d2XB5-tuCWU* on October 13, 2020.