Evaluation of suitability of conditional generative models for classification

(Zbadanie przydatności warunkowych modeli generujących dane do klasyfikacji)

Wojciech Pratkowiecki

Praca magisterska

Promotor: dr hab. Jan Chorowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

22 września 2020

Abstract

As a part of an autoencoder, a conditional generative model is supposed to reconstruct some complex samples conditioned on a low dimensional latent representation produced by encoder. This compressed form of data can be used for input values classification. With the Bayes' theorem, a generative model can be converted into a classifier, what allows to assess whether and how does it use the latent representation. As the reconstructor has a direct impact on features stored in the latent, a proper model architecture may be crucial for accurate autoencoder's input classification. This thesis describes experiments and analyzes aimed at selecting a reconstructor best suited for this task.

Rolą warunkowego modelu generującego w autoencoderze jest możliwie dokładna rekonstrukcja skomplikowanych danych na podstawie utworzonej przez enkoder niskowymiarowej reprezentacji ukrytej. Postać ta może zostać również użyta do klasyfikacji wartości podanych na wejściu autoencodera. Przy pomocy twierdzenia Bayesa modele generatywne mogą zostać przekształcone w klasyfikator, co pozwala ocenić na ile i w jaki sposób wykorzystują one reprezentację ukrytą. Jako, że rekonstruktor ma bezpośredni wpływ na informacje w niej zapisane, dobór odpowiedniej architektury modelu może być kluczowy w klasyfikacji danych wejściowych autoencodera. Ninejsza praca opisuje eksperymenty i analizy, mające na celu dobór rekonstruktora, który możliwie ułatwiałby to zadanie.

Contents

Chapter 1

Introduction

Generative models have achieved great results in modeling complex structures from various domains. For example, they can produce realistic samples of human faces and landscape photos [1], [2] or even write a novel like GPT-3 language model [3]. A high quality output is obtained from a little input information - we can for instance feed a generative model with a conditioning c that determines hair color of sampled face or a sequence of letters which should be "written" by a model generating images of characters sequence.

Autoencoder architectures use generative models to turn a relatively small latent representation z into a sample $\hat{X}$ as similar to the input X as possible. In semi-supervised learning the same latent z is treated as a new data representation and is used to classify the input. The encoder's task is to find the underlying structure of the data, so that mapping to the latent space would allow to effectively classify the unlabeled samples.

In this thesis our goal is to build a model that takes scans of handwritten documents and produce their transcripts, and train it in a semi-supervised way. We would like to have an encoder, that maps the letters from the image to some latent vectors related to the written characters and possibly independent of features like the author's writing style or the paper's structure. We may wonder what's the decoder's impact on the latent space in such a scenario. If some reconstructor architecture would guide the encoder to produce a latent representation, that would be more related to the original transcript, we should use it in our model, hoping that the classification would be more accurate.

Chapter 2 of the thesis introduces some background deep learning knowledge. The next chapter contains more detailed information about used approaches and our goals. The following chapters describe three groups of conducted experiments - in chapter 4 we use the Bayes' theorem to evaluate a reconstructor as a classifier and compare different decoder architectures. Chapter 5 shows how to improve a reconstructor's performance with additional network that models writer's style. The

6th chapter explores a decoder's behaviour when it is able to modify latent vectors and tune the features they store.

Chapter 2

Background

2.1 Artificial Neural Networks

Artificial Neural Network (ANN) is a broadly used computational model, inspired by neurons in the human brain. ANN is composed of units called neurons, that are stacked into layers of the network (see figure 2.1). In a classic setup, each neuron is a non-linear function applied on weighted sum of the input values. The very first layer performs its computation on the whole model's input (e.g. RGB image), while each of the following layers (hidden layers) process the previous layer's output. The value returned by the last layer is the model's final result and can be interpreted as e.g. the probability distribution over possible answers (e.g. is the image presenting a dog?).

Neural networks are a fundamental concept of deep learning - machine learning techniques based on models composed of numerous neural layers. To obtain a successful model we have to provide a *loss* function - a magnitude determining, how well (or how bad) is a network performing. It should be computed as a function of model's parameters Θ. During the learning process we evaluate a network on input values, compute the loss function that measures the model's error and compute its gradient. The gradient value is pointing in the direction that maximizes the loss function, therefore changing the model's parameters with respect to the opposite of the gradient, during a process called *error backpropagation*, should lead to minimizing the error of our network.

Over the recent decades researchers have invent many improvements of neural networks and demonstrated how to apply them to various tasks. Convolutional neural networks (CNN) [4] allow us to build models with reasonable number of parameters that can comprehensively process images. Recurrent neural networks (RNN) brought great results in sequence processing. A huge growth of neural networks capabilities and applications may have been observed in last few years. Development of distributed computations, GPU acceleration and effective learning algorithms al-

lowed to apply complex neural network architectures in many real world problems.

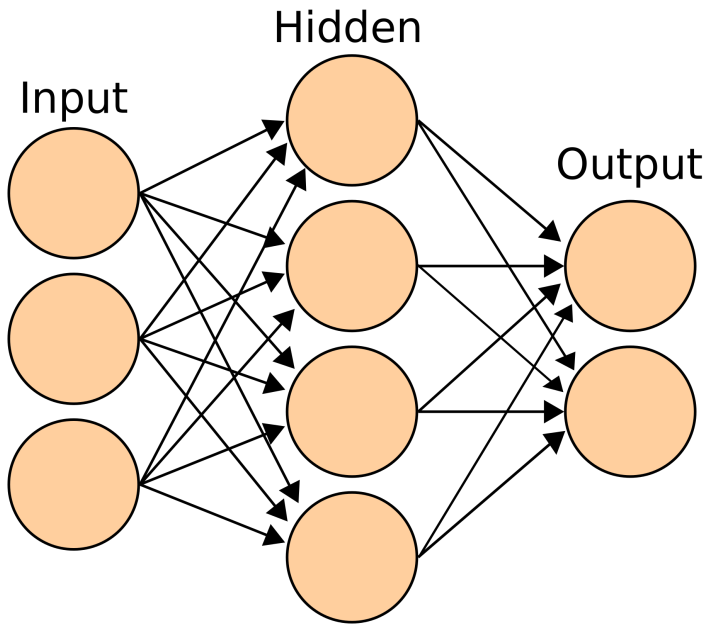


Figure 2.1: Example of neural network architecture composed of three layers - input layer, one hidden layer and an output layer. Picture from Wikipedia.

2.2 Generative models

We can divide machine learning models into two types - discriminative and generative ones. Discriminative models process an input and compute scores assigned to possible labels with respect to the input value. Commonly the model's output is a vector of each label's probability. If we denote data instances as X and labels set as Y, a discriminative model computes a conditional probability $P(Y|X)$.

On the other hand, generative models compute how likely is a given data sample. If labels set Y is provided they model a joint probability $P(X,Y) = P(X|Y) \cdot P(Y)$ or alternatively, when a a target label $y \in Y$ is given, probability of observation X, what can be denoted as $P(X|Y = y)$. Generative models are therefore able to produce new data instances, as they capture a distribution of the dataset. With a model trained on labeled images of animals, we may hence ask the network to generate a new image presenting a cat or a dog.

Generative models' task is much more challenging than the one faced by discriminative networks. When processing some images, a discriminative model may take an accurate decision by capturing few core features for each class - it may dis-

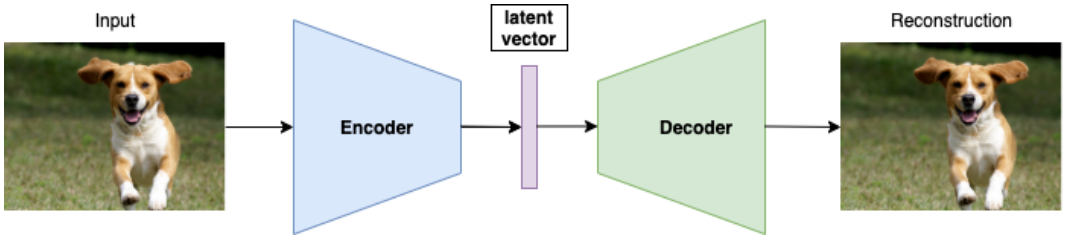


Figure 2.2: A visualization of autoencoder's mechanism. The encoder takes input value x and converts it to the latent vector z. Afterwards, the decoder tries to reconstruct the original image from the z vector

tinguish a cat from a crocodile by looking for a water-like patterns etc. An efficient generative model needs to capture many correlations - a head should be drawn with an ear on both sides, a mouth under a nose and so on.

2.3 Autoencoders

Autoencoders are a group of neural network architectures learned to effectively compress and reconstruct a model's input. An autoencoder is constructed with two neural networks - an encoder E that processes the input value x and provide its new, smaller representation $z = E(x)$, and a generative decoder/reconstructor R which takes the z value and produces $\hat{x} = R(z)$ - a reconstruction of x. The decoder's goal is to deliver $\hat{x}$ as similar to x as possible, therefore its learning process is focused on optimizing a loss function $L(x, R(E(x)))$. Figure 2.2 presents the autoencoder's workflow.

The autoencoder's training approach forces the encoder E to capture underlying form of the data samples, as the latent vector z needs to contain only crucial information about the input. E must therefore reject all the redundant features and noise when processing the data. Meanwhile, the reconstructor R may be used as a generative tool producing new data samples. Employing the decoder as a data generator is especially effective in case of Variational Autoencoder (VAE) [5], as its encoder maps input values into a latent space distributed possibly close to the standard Gaussian thank to a Kulback-Leibler term in the loss function. A VAE's reconstructor is therefore used to produce realistic samples from the normal distribution, so obtaining a new data instance may be done by evaluating it on a vector sampled from the $\mathcal{N}(0, I)$.

2.4 Semi-supervised learning

When building a classifier the most needed thing is labeled data. Unfortunately, often the amount of labeled samples is painfully smaller than the amount of available

unlabeled data for almost every domain. Imagine training a classifier C distinguishing images of road signs. You may build a valuable dataset $\mathcal{X}$ by manually labeling thousands of pictures. However, there will still be many more images $\overline{\mathcal{X}}$ presenting road signs that you won't manage to designate, but using them during the training process would improve the classifier's performance.

In such scenario the semi-supervised learning approach steps in. When learning a model this way we evaluate a classifier C not directly on a data sample x, but on its latent representation $z = E(x)$ produced by an encoder E trained as a part of an autoencoder. As described in the previous section, such an encoder is able to produce a latent representation much more condensed than the original input value. Moreover, we can train the autoencoder on both labeled and unlabeled data $\mathcal{X} \cup \overline{\mathcal{X}}$, therefore the latent representation could capture much more universal features of the road sign images. If we classify sample $\overline{x} \in \overline{\mathcal{X}}$ afterwards, the $C(E(\overline{x}))$ prediction should be significantly more accurate.

Chapter 3

Concept

3.1 Manuscript transcription

Experiments made within this thesis relate to the "Distant supervision for representation learning" project, developed during JSALT 2019, by a team lead by Jan Chorowski. The team was researching some models and approaches to effectively produce a transcript of some very challenging documents - 16-18th century Dutch manuscripts [6]. Samples from the dataset (see figure 3.1) are difficult to classify for many reasons - not only are the characters very spiral, but also it's usually impossible to segment a text line into individual characters, as many of them overlap. The team's segmentation approach was described in an individual paper [7].

The number of available scanned and transcribed manuscripts is remarkably small. Labeling data of this kind is exceptionally demanding, therefore extending the labeled dataset with new samples seems inefficient. However, with the semi-supervised approach we can make use of the unlabeled documents. As mentioned in the previous section, we may build an autoencoder learned on both labeled and unlabeled documents, that would process following text lines and encode them as latent vectors. A single line from the ScribbleLens dataset is a 1 channel, 32 pixels high image. While encoding it to the latent space, we slide through the consecutive picture columns and represent each $32 \times k$ block of pixels as a single latent vector z. A line of width W is therefore encoded to $z_0, z_1, ..., z_{\frac{W}{k}-1}$ vectors, that are are classified as characters afterwards, what produces a prediction of line's transcript.

3.2 Reconstructor's role

Having the $Z = \{z_0, z_1, ..., z_{\frac{W}{k}-1}\}$ latent vectors a reconstructor R tries to recreate the original picture of a manuscript's line l and maximize the $p(l|Z)$ probability. The

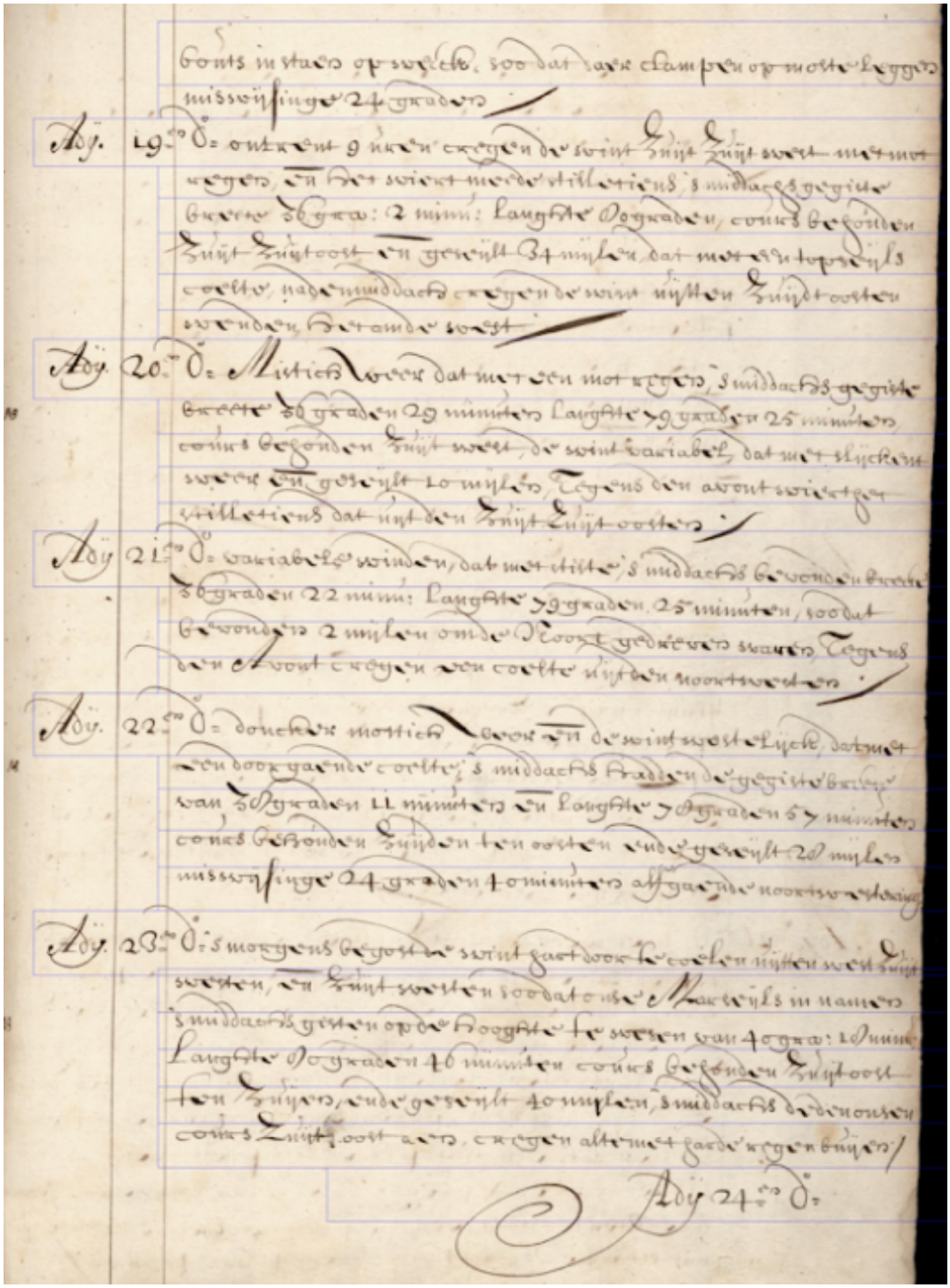


Figure 3.1: Example page from the ScribbleLens dataset.

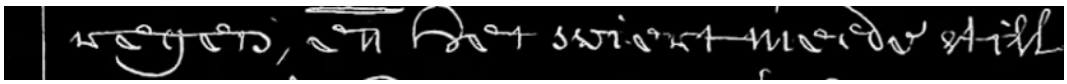


Figure 3.2: A single line of manuscript from the ScribbleLens dataset. When creating a transcription model, images of lines are used as an input.

loss function minimized by R can be described as:

$$L(l, R(E(l; \Theta_E); \Theta_R))$$

During the training a reconstructor R is therefore tuning parameters Θ_E of the encoder E. We can interpret the decoder R to guide the encoder forward a representation useful for the reconstructor. Choosing a proper decoder's architecture is then a crucial aspect when building a successful transcript model - we would like it to demand the z vectors to include information about the encoded characters rather than properties like sum of pixels values in each $32 \times k$ pixels block. If a reconstructor forced the encoder to produce something like one-hot vectors of the following characters in a line, our global classification task would become trivial.

3.3 Using reconstructors as transcript classifiers

By replacing the latent vectors z with some manually provided values, we are able to evaluate reconstructors separately from any encoder. We may even use the characters' one-hot vectors as input values, so a reconstructor models $P(X|Y)$, where Y is a characters sequence. If we use a manuscript line's x ground-truth transcript y_T, we can obtain decoders that will reconstruct the line's image from its transcript and model $P(X = x|Y = y_T)$ probability. As already mentioned, a desired reconstructor would reproduce the image x most accurate, when $Y = y_T$. However, the Bayes theorem says that:

$$P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$$

moreover, for any value of Y (any sequence of characters) the $P(X)$ term is constant. If we assume that each combination of characters is equally likely (what is not true in general, however simplifies the computations), we may say that:

$$P(Y|X) = P(X|Y) \cdot \alpha$$

so $P(Y|X)$ is directly proportional with $P(X|Y)$ - a probability modeled by a reconstructor. With appropriate reconstructor's architecture changes we are able to obtain the $P(X = x|Y = y)$ value. From the Bayes' theorem for any input characters sequence y_i we get:

$$P(y_i|x) = \frac{P(x|y_i)}{\sum_j P(x|y_j)}$$

However a computational cost of sum in the denominator is exponential - we would have to obtain a probability of each possible characters sequence. Instead of predicting a whole transcript we may therefore model a probability of a single character - if we denote the transcript y with i-th letter substituted to the j-th element of the alphabet as $y\{c_{i_j}\}$ we can obtain a prediction for each possible j as:

$$P(y\{c_{i_j}\}|x) = \frac{P(x|y_i)}{\sum_k P(x|y\{c_{i_k}\})}$$

This allows us to determine the most likely characters sequence as decoder's R input for target image x. If the highest score is obtained for y_T (i.e. the most likely c_{i_j} is the true one), R would demand an encoder to produce a latent representation close to the y_T, what would be very beneficial. We may therefore order all the reconstructors, by using them as Y sequences classifiers and find an architecture best suited for our task separately from the remaining part of the transcript model.

Chapter 4

Reconstructors comparison

4.1 Computation process

All the computations were implemented in Python, the PyTorch [8] framework was used for deep learning facilities. Most of the reconstructor architectures implementation used during experiments had been developed by the JSALT team.

ScribbleLens dataset contains images of manuscripts lines. Every line is a $H \times W$ 1-channel picture represented as a tensor L, whose elements describe brightness of a pixel in 16 values scale. A reconstructor R produces an output basing on input values - sequence of vectors z computed by an encoder or set manually.

The alphabet of ScribbleLens samples contains 68 characters, including space or comma sings. The L line's conditioning in a form of one-hot transcription can be therefore represented as $68 \times W$ matrix, where w-th column is a one-hot vector v_w of a character presented in w-th column of the image L. As already stated, each vector z can be assigned to multiple columns of the input - it may represent k following columns of the picture. When combining consecutive image columns $[x_i, \ ... \ x_{i+k}]$ into z_i, the element-wise logical OR of $[v_i, \ ... \ v_{i+k}]$ one-hot vectors is used (see figure 4.1).

An auto-regressive reconstructor R fed with $[z_0, ..., z_{\frac{W}{k}}]$ conditioning vectors produces a prediction of the image L reconstruction. The decoder's output is a $H \times W \times 16$ tensor and each of $H \cdot W$ values $s \in \mathbb{R}^{16}$ determines likeliness of pixel brightness value quantized to 16 levels. If we apply the Softmax function on the s, we can obtain a $H \times W \times 16$ tensor T representing probabilities of each possible pixel value at every image position $[h, w]$. As an auto-regressive reconstructor is used, probability of each pixel $L_{[w,h]}$ is computed based on all the previous pixels,

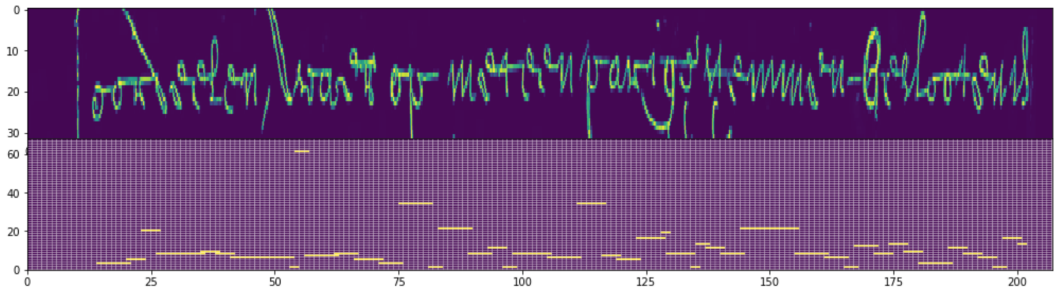


Figure 4.1: An exemplary line from ScribbleLens dataset and a visualization of its encoding - element-wise logical OR of k ($k = 4$) one-hot vectors assigned to corresponding columns of the image. The line is presenting *oordeelen Waerop met eenparige stemmen beslooten is* text.

so consecutive s vectors are computed as:

$$s_{[0,0]} = P(L_{[0,0]})$$
$$s_{[0,1]} = P(L_{[0,1]}|L_{[0,0]})$$
$$...$$
$$s_{[w,h]} = P(L_{[w,h]}|L_{[0,0]}, L_{[0,1]}, ..., L_{[w,h-1]})$$

With this kind of tensor T we are able to compute a probability of any $H \times W$ image by combining its pixel values. It allows us to determine $p(x|y)$ - the probability of image x modeled by the reconstructor R conditioned on the characters sequence y.

To test the relatedness between image x and its transcript y_T, we compute $P(x|y\{c_{i_j}\})(j = 0...67)$, where $y\{c_{i_j}\}$ is the transcript of x with i-th letter substituted to each of 68 possible characters - if a line presents a single word *dog*, the $y\{c_{1_j}\}$ would be each of *dAg*, *dBg*, ..., *dOg*, ... sequences. In the previous section we argued that a good reconstructor would perform best, when a true transcription y_T is used as its input and $p(y_T|x)$ would be maximized. To obtain the probability distribution $p(y\{c_{i_j}\}|x)$ of transcripts, we apply the Softmax function on $[p(x|y\{c_{i_0}\}), ..., p(x|y\{c_{i_{67}}\})]$ values. Figure 4.2 visualizes the described computation process.

4.2 Architectures results

The following sections describe results of training different reconstructor architectures in the described manner and testing them as transcript classifiers. All tested architectures had been developed and tested in a full transcript model before, with encoder modeling the latent vectors and a classifier producing the transcript's prediction. During this experiments the reconstructors were separated from the rest of the whole setup. In few cases some additional experiments were conducted. Table 4.1 presents the summary of obtained results.

Model	Accuracy	Average probability of the true character	Average probability of the most likely character when classified correctly	Average probability of the most likely character when classified wrongly
PixelCNN	59%	55%	86%	53%
PixelCNN Large	55%	53%	88%	57%
PixelCNN Lookahead various number of skipped columns	26.6%	24.2%	83.1%	71.9%
PixelCNN Lookahead fixed number of skipped columns	47%	43%	85%	52%
PixelCNN right-to-left	48%	42%	81%	52%
WaveNet	39%	39.2%	94%	81.4%
WaveNet Large	32%	32%	92%	82%
WaveNet lookahead various number of skipped columns	21%	20.9%	94.5%	86.7%
WaveNet noisy labels	11.5%	—	—	—
Bidirectional: PixelCNN + PixelCNN	**71%**	68%	94%	66%
Bidirectional: PixelCNN + PixelCNN R_{LR}/R_{RL}	**63%** / 46%	59% / 40%	88% / 80%	55% / 51%
Bidirectional: PixelCNN + Wavenet	57%	55%	94%	81%
Bidirectional: PixelCNN + Wavenet R_{LR}/R_{RL}	57% / 29%	55% / 28%	88% / 86%	53% / 74%
Bidirectional: Wavenet + PixelCNN	59%	58%	95%	79%
Bidirectional: Wavenet + PixelCNN R_{LR}/R_{RL}	39% / 47%	39% / 42%	91% / 81%	75% / 50%
Bidirectional: Wavenet + Wavenet	51%	51%	96%	83%
Bidirectional: Wavenet + Wavenet R_{LR}/R_{RL}	43% / 25%	41% / 25%	90% / 86%	75% / 72%

Table 4.1: Accuracies of different reconstructor architectures when used as transcripts classifiers. The first column is a reconstructor type, the rows with R_{LR}/R_{RL} annotation present classification results of both left-to-right and right-to-left reconstructors of a bidirectional model evaluated separately. The second column is a model's accuracy - how many times the $p(y\{c_{i_T}\}|l)$ was the highest probability over all $p(y\{c_{i_j}\}|l)$ values. The third row presents an average probability assigned to the $y\{c_{i_T}\}$ over all test samples, while the last two columns show the average probability assigned to the most likely characters when a prediction was correct and wrong respectively.

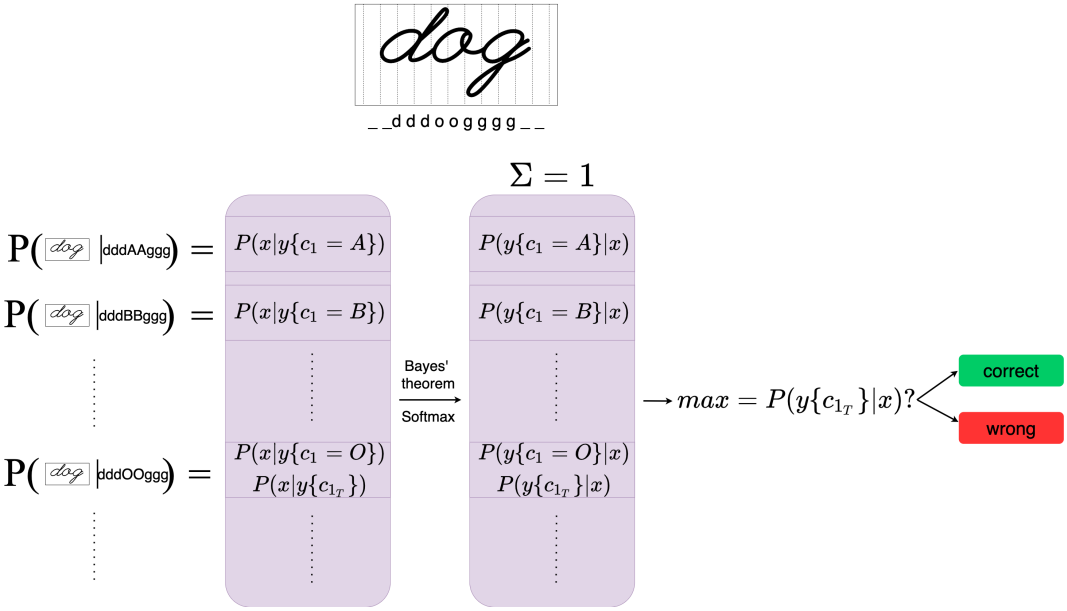


Figure 4.2: A visualization of the classification process. At the top we can see the image x and its ground truth transcript y_T. Below is the computation process: firstly we compute each of $P(x|y\{c_{i_j}\})$ probabilities. Afterwards we use the Bayes' theorem to convert it into $P(y\{c_{i_j}\}|x)$. If this probability is greatest for $c_{i_j} = c_{i_T}$the classification is correct.

4.2.1 Gated PixelCNN

Gated PixelCNN [9] is an auto-regressive generative model, using the chain rule to model consecutive pixels:

$$p(x) = \prod_{i=0}^{H \cdot W} p(x_i | x_0, ..., x_{i-1})$$

and if a conditioned probability is computed:

$$p(x|\mathbf{h}) = \prod_{i=0}^{H \cdot W} p(x_i | x_0, ..., x_{i-1}, \mathbf{h})$$

The "gated" term stand for the activation units used in the model, that were designed to resemble the multiplicative units from the LSTM [10]:

$$\mathbf{y} = \tanh\left(W_{k,f} * \mathbf{x}\right) \odot \sigma\left(W_{k,g} * \mathbf{x}\right)$$

If a conditioning is used, the $\mathbf{h}$ vector is added before the nonlinear functions:

$$\mathbf{y} = \tanh\left(W_{k,f} * \mathbf{x} + V_{k,f}^T \mathbf{h}\right) \odot \sigma\left(W_{k,g} * \mathbf{x} + V_{k,g}^T \mathbf{h}\right)$$

PixelCNN is an improvement of PixelRNN [11] proposed by the same authors before. One of the biggest advantage of the PixelCNN are the convolutional layers used in the architecture, that significantly speed up the training process. To prevent a model

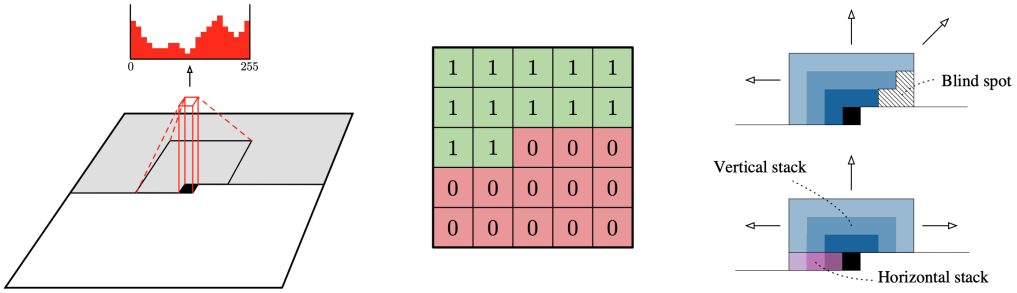


Figure 4.3: **Left:** visualization of the pixel prediction process. The already modeled pixels are used to determine the probabilities of each possible value for the current pixel. **Middle:** a mask used for masked convolutions - the kernel is multiplied by 0 and 1 values to hide the following pixels. **Right:** vertical and horizontal stacks used to capture the neighbourhood.

from reading the current and following pixels while making a decision, the masked convolutions were used. Also vertical and horizontal stacks allow to capture the receptive field, so the pixel's neighbourhood is taking into account during prediction. Figure 4.3 presents these main concepts of the Gated PixelCNN.

After a PixelCNN training, the model was evaluated as a transcript classifier. It was tested on following lines l_j from the test set of the ScribbleLens, that contained 300 letters c_i. During each test case a c_i letter in the y_j transcript was set to every of 68 possible values $(c_{i_0}, ..., c_{i_{67}})$. Afterwards the picture l_j probability conditioned on the modified transcript was computed. The LogSoftmax function was then applied to a $H \times W \times 16$ tensor produced by the reconstructor. Finally, a sum of the real image l_j pixel values was taken, what gave us the log probability of the true picture. Having 68 such sums (one for each c_i) the Softmax is applied to obtain the probability $p(y_j\{c_{i_k}\}|l_j)$ of each possible transcript.

Figure 4.4 presents the histogram of probabilities assigned to the true character c_{i_T} for each of 300 classified letters $c_0, ...c_{299}$. Over 30% of c_i characters were classified correctly with a probability close to 1, i.e. probability of the unchanged transcript $p(y_j\{c_{i_T}\}|l_j)$ was definitely the highest one from $p(y_j\{c_{i_0}\}|l_j), ..., p(y_j\{c_{i_{67}}\}|l_j)$. However, over 15% test samples were strongly misclassified - the $p(y_j\{c_{i_T}\}|l_j)$ was much smaller than some other values.

Overall 59% of samples were classified correctly. Table 4.1 presents the results compared to other models. The average probability of the true character $p(y_j\{c_{i_T}\}|l_j)$ for each $i = 0, ..., 299$ was 55.9%, the same value computed only for the correctly classified samples was 86.7%. When the PixelCNN was wrong, it was not such confident about its choices - in the badly classified test cases the average probability of the most likely letter was just 57.4%. The bottom histogram on figure 4.4 shows the

highest computed probability for each test case ($\max_k p(y_j\{c_{i_k}\}|l_j)$ for $i = 0, ..., 299$).

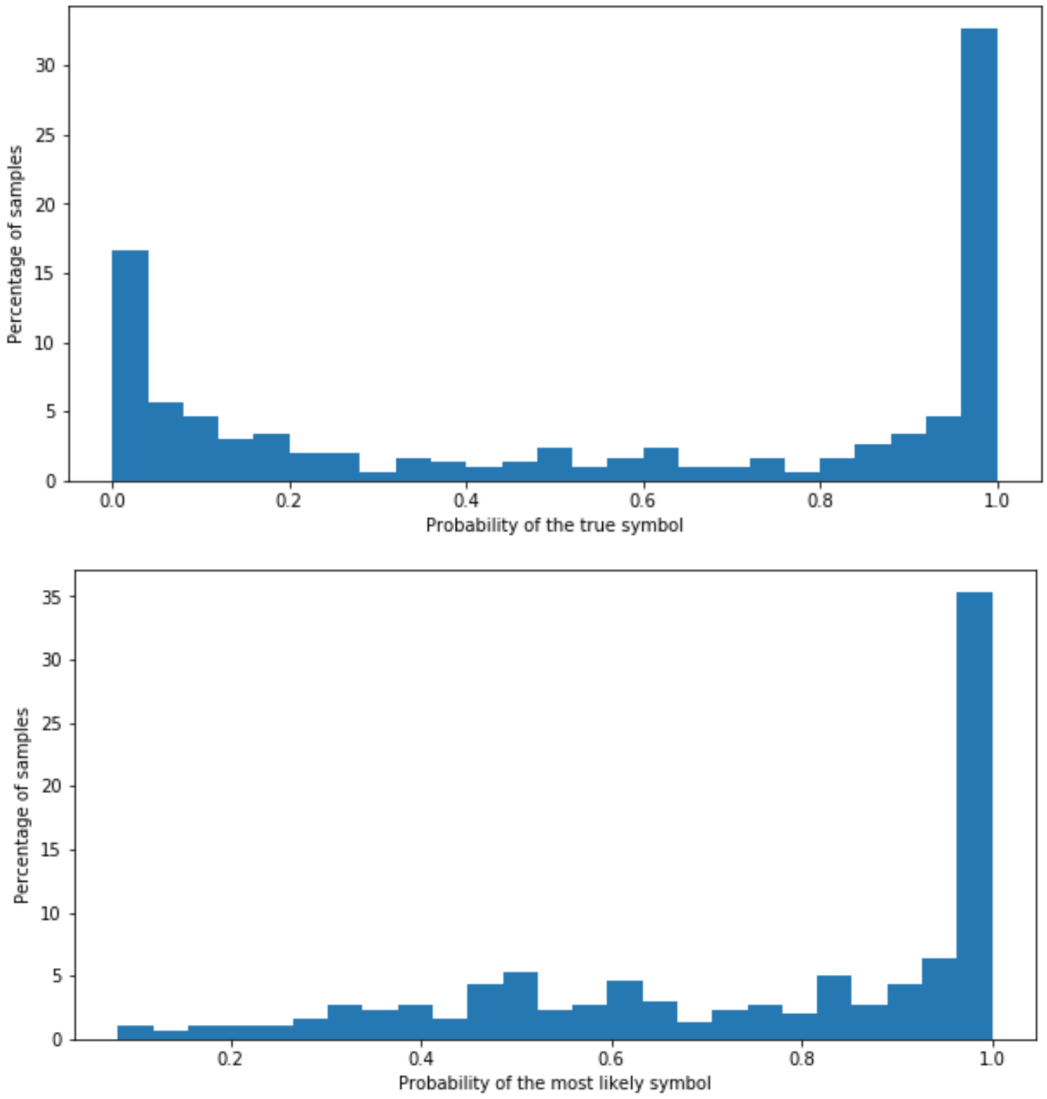


Figure 4.4: **Top:** Histogram of probability of the true character - $p(y_j\{c_{i_T}\}|l_j)$. **Bottom:** highest probability $\max_k p(y_j\{c_{i_k}\}|l_j)$ computed by the PixelCNN for each c_i character.

Having all these computations done we can investigate the results a bit. First of all we can check which one of 68 symbols was badly classified most frequently. Over 28% of these samples were representing a blank space (" ") - when processing pixel columns with the blank space assigned, PixelCNN often considered them as some letters. This weird behaviour is more understandable when we display a line segmented into individual characters, like on figure 4.5. We can see that many characters overlap, and the blank spaces may contain some portions of neighbouring letters.

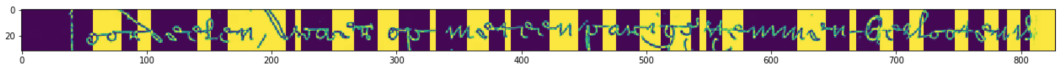


Figure 4.5: A ScribbleLens line segmented into individual characters (according to alignment from the dataset). The background color separates consecutive characters. We can clearly see, that the segmentation is not detailed and many characters overlap.

As classifying the blank space is problematic, we can remove them from the 300 test cases and recompute the results. 65% of 263 remaining samples were correctly classified, while the average probability $p(y_j\{c_{i_T}\}|l_j)$ was 61.5%. Finally we can display some of the misclassified samples, however it doesn't provide any valuable information, we can at most once again notice the imprecision of letters segmentation (see figure 4.6).

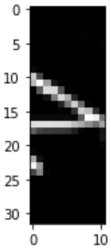


Figure 4.6: One of the wrongly classified character. The ground truth label is *d*, the $p(y_j\{c_i = \text{"}d\text{"}\}|l_j)$ was 10%, while the most likely symbol according to the model was *9* with 46% probability.

4.2.2 Larger Gated PixelCNN

When a PixelCNN was extended with additional hidden layers and extra channels per layer, no improvement was noticed. As presented in the table 4.1, the model predicted correctly 55% of test cases with the 53% average probability of ground truth label over all the samples. Once again it was confident of its choices when the most probable character was the true one - an average probability of the letter was 88% in these cases.

Just like the smaller PixelCNN, the model did not cope well with blank spaces - the space was the true symbol in 42 test cases, 39 of them were wrongly classified. After removing the spaces from the tests, 63% of remaining samples were correctly classified and the average probability of the true label increased to 60%.

4.2.3 Lookahead Gated PixelCNN

The Lookahead variant of the Gated PixelCNN computes a pixel's value basing on the past ones, but it did not consider values from n neighbouring columns on the left. The n could either be constant or it could be randomly picked from given range. Omitting the neighbouring column was introduced to force a reconstructor to focus on the input vectors z, rather than on the image. However, the Lookahead PixelCNN has failed, especially, when the number of skipped columns was randomly chosen.

If we compare a reconstruction loss over 20 ScribbleLens lines computed for the different PixelCNN variants (see figure 4.7), the Lookahead with randomly picked number of skipped columns performs much worse than the others. It shows, that this network was not able to model the manuscript lines.

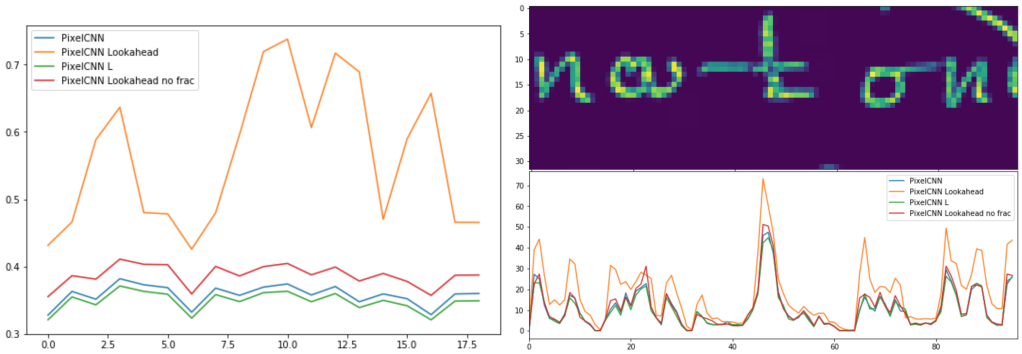


Figure 4.7: PixelCNN reconstruction comparison. The orange line is representing the Lookahead with various number of skipped columns. **Left:** average reconstruction loss (nats/pix) over 20 lines from ScribbleLens (X axis is a sample number). **Right:** the average nats/pix loss over consecutive columns of the image.

4.2.4 Wavenet

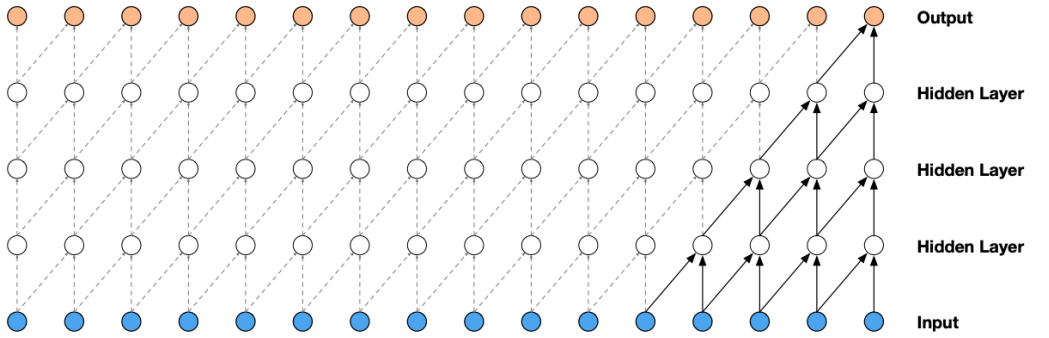
The next tested reconstructor architecture was the Wavenet model [12] that was originally developed for audio generation that modeled the joint distribution of the waveform $x = \{x_0, x_1, ..., x_T\}$ as a product:

$$p(x) = \prod_{t=0}^{T} p(x_t | x_0, x_1, ..., x_{t-1})$$

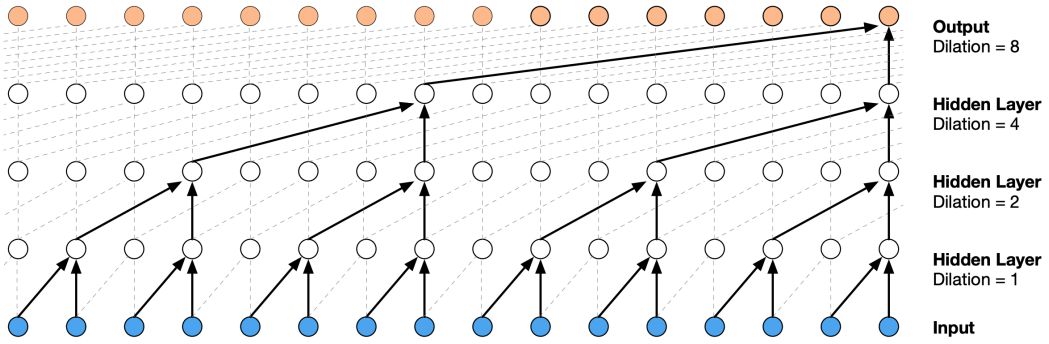
therefore the same approach as in PixelCNN is used to compute the probabilities. Wavenet uses the convolutional layers to model the audio wave forms, what makes it much faster than RNNs, however, architectures with stacked convolutional layers have a relatively small receptive field. The Wavenet's author used dilated convolutions to increase the field exponentially - figure 4.8 visualizes this approach. To ensure that a single prediction $p(x_t|x_0, ..., x_{t-1})$ does not depend on anu of $\{x_t, x_{t+1}, ...\}$

values, the masked convolution are used. Just like the PixelCNN, Wavenet is using gated activation units, as they perform significantly better in sequence modeling than some traditional activation functions.

Although the Wavenet is a model adjusted to audio wave forms generation, it can be used for image modeling. In such configuration it would produce a whole column of pixels at a time. As auto-regressive, models like PixelCNN, process each consecutive pixel individually, a computational cost of predicting an entire column is much lower, what is one of the Wavenet's advantages.



(a) The receptive filed of casual convolutional stacked layers.



(b) The receptive field when dilated convolutions are used.

Figure 4.8: The exponential difference in receptive fields.

The Wavenet reconstructor was tested the same way as PixelCNN - it was evaluated on manuscript lines from test set of the ScribbleLens. The number of lines was selected to contain 300 letters. Afterwards each letter c_i was set to every of 68 possible values $c_{i_0}, ..., c_{i_{67}}$ and the probability of the line image $p(l_j|y_j\{c_{i_j}\})$ was computed conditioned on transcript with modified i-th letter. Applying the Softmax function on $[p(l_j|y_j\{c_{i_0}\}), ..., p(l_j|y_j\{c_{i_{67}}\})]$ gave the probability distribution over possible $y_j\{c_{i_j}\}$ transcripts.

Wavenet performed significantly worse as a classifier than the PixelCNN. Only 39% of samples were classified correctly, with 39% average probability of the true symbol $p(y_j\{c_{i_T}\}|l_j)$. Wavenet was also much more convinced about its choices - when the classification was accurate, the average probability $p(y_j\{c_{i_T}\}|l_j)$ reached

over 94%, in case of badly classified samples, the average probability assigned to the most likely c_{i_j} symbol was 81.4%. Figure 4.9 presents histograms of obtained result.

Just like the PixelCNN, WaveNet was wrong in almost all samples of blank spaces. If we evaluate the reconstructor on samples without the blanks, 111 from 260 remaining samples (43%) were classified correctly, with 43% average probability of the true label. Average probabilities of the most likely symbol was 94.8% when the model was right and 83.9% when it was not. We tried to improve the Wavenet's performance by increasing its size, both by adding some extra hidden layers and by increasing the number of channels, however we did not manage to obtain any better results.

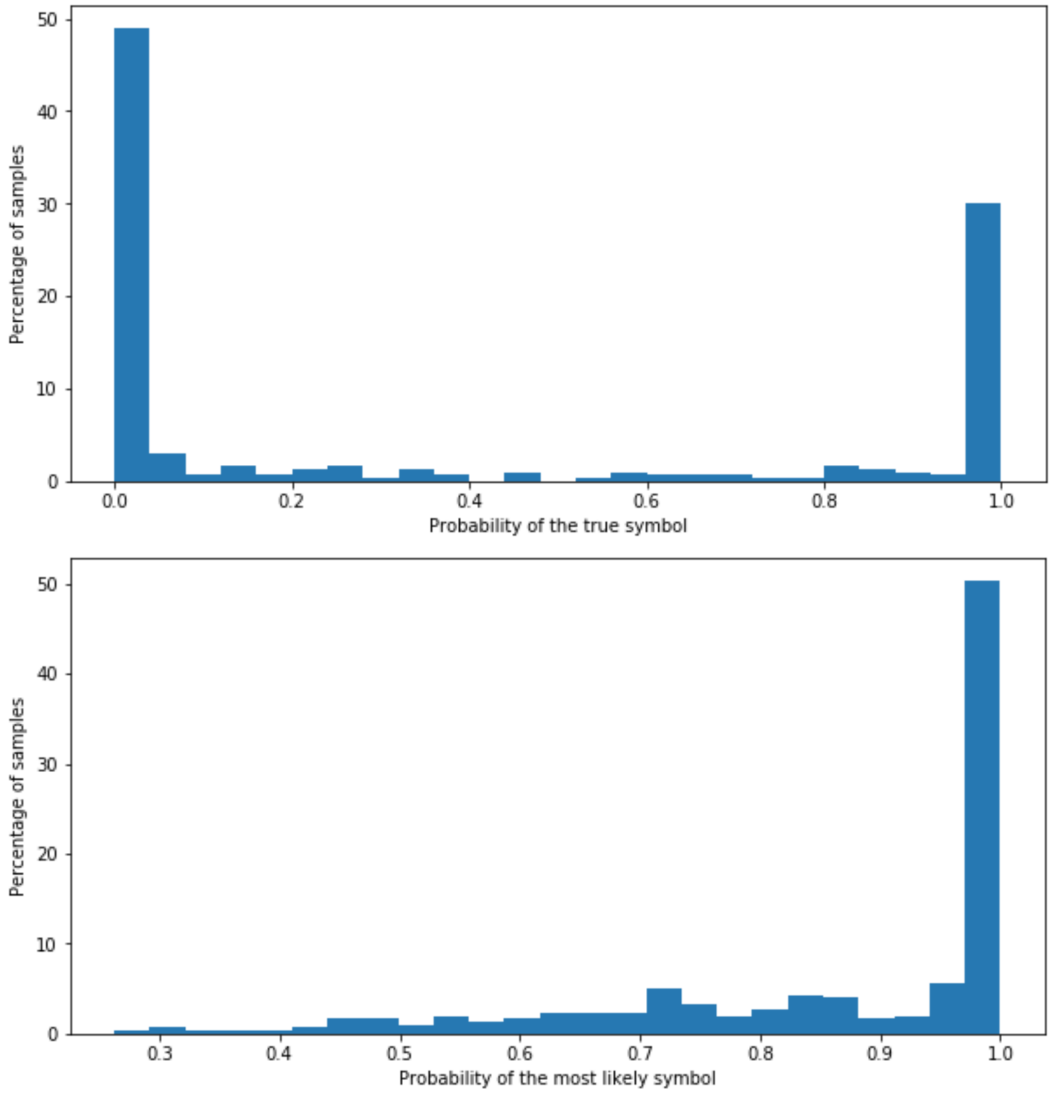


Figure 4.9: **Top:** Histogram of probability of the true character - $p(y_j\{c_{i_T}\}|l_j)$. **Bottom:** highest probability $\max_k p(y_j\{c_{i_k}\}|l_j)$ computed by the Wavenet for each c_i character. We can see, that in most cases the model assigned very high score to the most likely symbol.

As reconstructors are generative models we can compare the capabilities of Wavenet and PixelCNN in sample modeling. Figure 4.10 shows three line images. The first one is an original sample l from the ScribbleLens. The two others are the generated samples, both conditioned on the ground truth transcript of l, the image produced by PixelCNN is above the Wavenet's output. The result of PixelCNN generation is quite satisfying, the reconstructor was even able to notice, that a line of text has a margin mark somewhere at the beginning. The sample produced by Wavenet is not as accurate, however it was more "even" - the characters in the PixelCNN image overlap, while the letters in the Wavenet sample are well segmented. It seems to be an effect of the Wavenet's generation approach - it produces a whole column of pixels at each step. Moreover a circled segment of generated lines shows that the reconstructors indeed use the conditioning while modeling the images - both models produced characters similar to the original ones there.

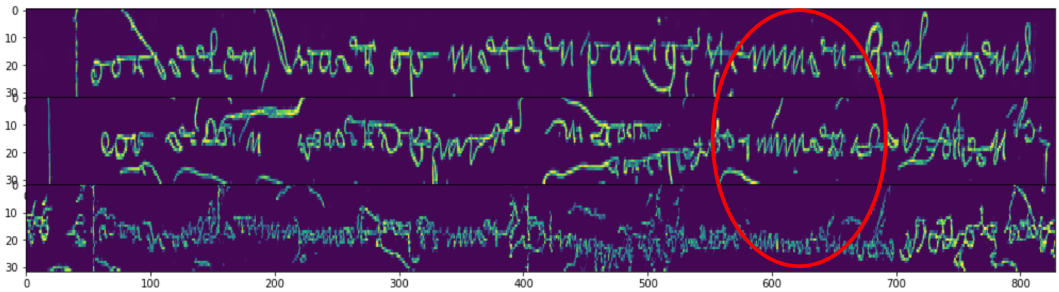


Figure 4.10: Generation of manuscript line image by the reconstructors. **Top:** a line from the ScribbleLens dataset, whose transcript was used as a conditioning for the models. **Middle:** a sample generated by the PixelCNN. **Bottom:** A sample generated by the Wavenet. The red circle points a segment where we can see, that the reconstructors generate characters related to transcript.

4.2.5 Lookahead Wavenet

Just like in the PixelCNN case, skipping the most recent pixel columns significantly degrades the Wavenet performance. From 300 tested samples only 21% were classified well, with just 20.9% average probability of the true character $p(y_j\{c_{i_T}\}|l_j)$ over all samples. Just like the casual Wavenet, the Lookahead one was assigning high probabilities for the most likely letters - 94.5% when the classification was correct and 86.7% when the reconstructor was wrong.

4.2.6 Wavenet with noisy conditioning vectors

The conditioning vectors $z_0, ..., z_{\frac{W}{k}}$ are constructed by taking the element-wise logical OR of k one-hot vectors assigned to following columns of an input image. We could

try to introduce some noise to the z vectors by flipping its single value at a random dimension with probability p. When we tried this kind of conditioning with $p = 0.1$, i.e. for any 86 dimensional conditioning vector z_i there is a 10% chance, that one of its 86 values will be changed (see figure 4.11), the reconstructor's performance was very poor - the classification accuracy dropped to just 11.5%, therefore we did not decide to continue experiments with the noisy conditioning.

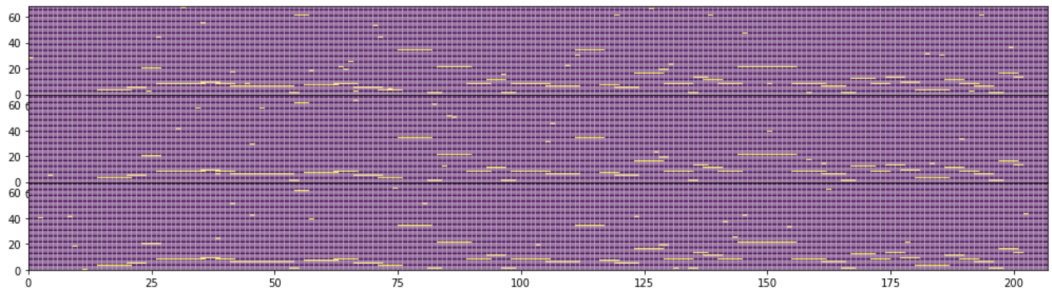


Figure 4.11: Three examples of noisy conditioning. Each picture presents a visualization of vectors $z_0, ..., z_{\frac{W}{k}}$, that were randomly modified. A small difference can be seen, however the performance drop was significant.

4.2.7 Reconstructors with different reduction length

When transforming a line's transcript y into conditioning vectors we combine k one-hot vectors $v_0, ..., v_k - 1$ into a single vector with the element-wise logical OR. In the foregoing computations we used $k = 4$, however we can train the reconstructors with other values of k. The experiments were repeated in two new configurations - with $k = 1$ and $k = 8$. When $k = 1$ the v vectors are not combined and each column of the image is conditioned on an individual vector. When $k = 8$ the v vector may contain information about many characters, as 8 pixel columns is enough space for few letters. However, in both cases we did not spot any difference in reconstructor's performance, therefore the reduction length parameter seems to be a secondary issue.

4.2.8 Bidirectional PixelCNN

Both PixelCNN and Wavenet process the input image from left to right, and none of the pixels on the right side is taken into the account during prediction. However, some knowledge about this part of a picture may be very useful - for instance when processing a first character of a line the reconstructor may be able to model the author's writing style much more accurate if the rest of the image was visible. To enable the two-way image processing, without exposing too many information during pixel prediction, we may use two reconstructors at once. The first one would be a casual left to right model R_{LR}, while the R_{RL} would process the image and

conditioning from right to left.

The R_{LR} models $p(x|y)$, while R_{RL} computes $p(x_{RL}|y_{RL})$, where x_{RL} and y_{RL} are mirror reflections of the x image and its transcript y respectively. We can assume that these models are conditionally independent, therefore we may combine their results and approximate $p(x|y) \approx p(x|y;\Theta_{R_{LR}}) \cdot p(x_{RL}|y_{RL};\Theta_{R_{RL}})$ i.e. probability of image x for y transcript is a product of probabilities computed by both models.

To evaluate this bidirectional reconstructor and obtain the $p(l_j|y_j\{c_{i_j}\})$ probability, we use the R_{LR} to reproduce the l_j from y_j transcription, while the score of R_{RL} is computed for flipped l_j and y_j. Afterwards we add these values, as they represent the log probabilities. The rest of the evaluation process remains unchanged.

The first tested bidirectional reconstructor was using PixelCNN for both left-to-right and right-to-left models. Again we tested it on some lines from test part of the ScribbleLens dataset that contained 300 letters c_i. The classification accuracy of the dual model was significantly better than the result of a casual PixelCNN - 71% of samples were classified well. The average probability of the true symbol $p(y_j\{c_{i_T}\}|l_j)$ was 68%. Also the bidirectional model was much more convinced on its predictions - the average probability assigned to the true symbol when the model was right was as high as 94%.

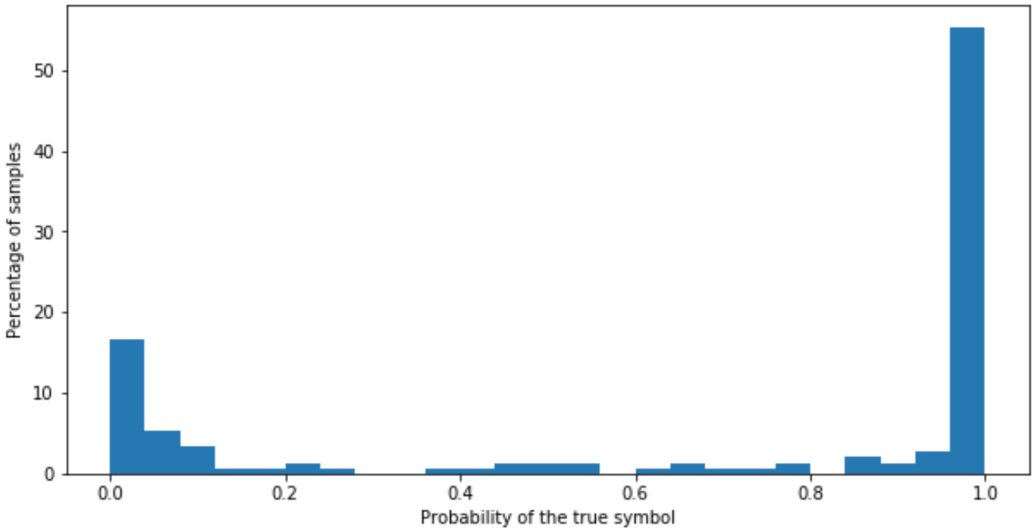
We could also compare the classification accuracy of both models separately. It turned out, that the left-to-right reconstructor was significantly much more accurate than the "mirror" model. Moreover, the left-to-right PixelCNN from the bidirectional model performed better than the PixelCNN trained in a casual way - the classification accuracy of the standalone R_{LR} was 63%. We may therefore assume, that bidirectional training is also a good regularization approach. Figure 4.12 presents the probabilities assigned to the true character for both R_{LR} and R_{RL} separately and combined.

The difference in classification performance between R_{LR} and R_{RL} is clear. However, both models obtain similar results in reconstruction tasks - figure 4.13 shows the change of reconstruction loss value during the training.

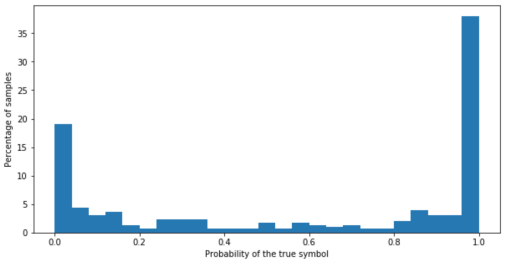
4.2.9 Right-to-left PixelCNN

As results obtained by right-to-left part of all tested bidirectional models were poor, we can construct a single right-to-left reconstructor and verify, whether its performance would be also so unsatisfying. After training a PixelCNN in the right-to-left manner, we evaluated all the tests. The reconstructor was again very inaccurate, it classified correctly only 48% of samples. The average probability of the true character was just 42% what made it the worst of all tested PixelCNN configurations.

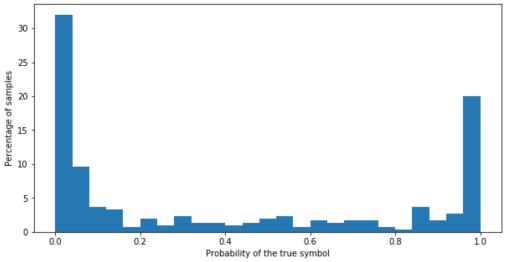
Figure 4.14 presents the generated samples by both left-to-right PixelCNN (middle) and the right-to-left one (bottom). The top image presents the target image.



(a) $p(y_j\{c_{i_T}\}|l_j)$ by combined model



(b) $p(y_j\{c_{i_T}\}|l_j)$ by left-to-right model



(c) $p(y_j\{c_{i_T}\}|l_j)$ by right-to-left model

Figure 4.12: **Top:** Histogram of $p(y_j\{c_{i_T}\}|l_j)$ for each i computed by combined model. **Bottom:** the same probability computed by R_{LR} and R_{RL} separately.

We can clearly see, that the right-to-left reconstructor models the transcript lines much worse, also it did not manage to capture the margin.

4.2.10 Bidirectional reconstructor - PixelCNN + Wavenet

We may combine different architectures into a bidirectional model. If a left-to-right reconstructor is a PixelCNN and the right-to-left one is a Wavenet, the classification result is not as good, as in double PixelCNN case. Just 57% of 300 samples were classified correctly, with 55% average probability of the true character. This bidirectional reconstructor also tended to assign high probabilities - when the prediction was correct, the average probability of the true symbol was 95%, when a decision was wrong the probability of predicted character was by average 81%.

Figure 4.13: The loss value of bidirectional reconstructor during the training. Two plots at the top presents the losses of R_{LR} and R_{RL} respectively, while the charts below represent the combined loss.

4.2.11 Bidirectional reconstructor - Wavenet + PixelCNN

When we swapped the reconstructors and use Wavenet as R_{LR} and PixelCNN as R_{RL}, the combined model was slightly better. This was a bit surprising, as we could have noticed, that the left-to-right reconstructor had a bigger impact on the overall result, and that PixelCNN was performing definitely better than the Wavenet. This bidirectional model classified correctly 59% of 300 samples. The probability of the most likely symbol when a prediction was right/wrong was by average 95%/79%.

4.2.12 Bidirectional reconstructor - Wavenet + Wavenet

The last tested bidirectional configuration was composed of two Wavenet architectures. The accuracy of this model was 51%, what was significantly higher than the score of a standalone Wavenet. Also, if the left-to-right reconstructor was used for classification it performed better than the single model - the R_{LR} accuracy was 43%. As in every bidirectional model, the right-to-left model was performing poorly.

4.2.13 Other tested models

During the experiments we also evaluated the tests on architectures like unsampling convolutional model. However, they did not manage to capture the complex distribution of the ScribbleLens samples. As the JSALT team was considering PixelCNN and Wavenet in various configurations for most time, we have also focused on these two architectures.

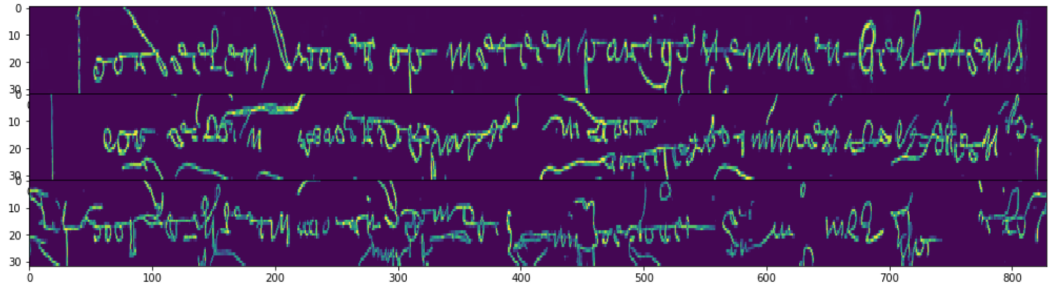


Figure 4.14: Generation of manusctipt line by two-way models. **Top:** a line from the ScribbleLens dataset, whose transcript was used as a conditioning for the models. **Middle:** a sample generated by the left-to-right PixelCNN. **Bottom:** A sample generated by the right-to-left PixelCNN.

4.3 Summary

From all the evaluated reconstructors, the bidirectional PixelCNN obtained the best result in the test. It means that the PixelCNN could be a reconstructor that would be able to guide an encoder to produce a latent representation accessible for a transcript classifier. It was also confirmed by the tests evaluated by the JSALT team - their best semi-supervised model was also using a bidirectional PixelCNN as the reconstructor. Consistency of the results shows that using a Bayes' theorem to assess a reconstructor is a well-based approach.

Chapter 5

Style modeling

Classifying reconstructors with the proposed approach seems to be a good benchmark for decoder architectures. We may wonder if we can somehow improve a reconstructor's performance in manuscript line modeling. A feature that strongly differentiates the pictures is the author's handwriting style - the same word written by various people may look completely different. If we were able to convey the information about the style to the reconstructor, it might be easier to model the manuscripts. Moreover, if a decoder would already know, that the author of the processed line has cursive writing, the latent representation would not need to include this information, and could be more focused on the characters. This idea prompts us to introduce a new element to the network - the style model.

When dealing with a dataset like ScribbleLens we have an access to many samples of a processed character's author - many of them are situated in the same line. Before making a prediction we may therefore show the line l to the style model that would produce a *style vector* s. The s could be then used as an additional conditioning, and the reconstructor would model $p(l|y, s)$, where y is the line's transcript. Figure 5.1 visualizes the described reconstruction process.

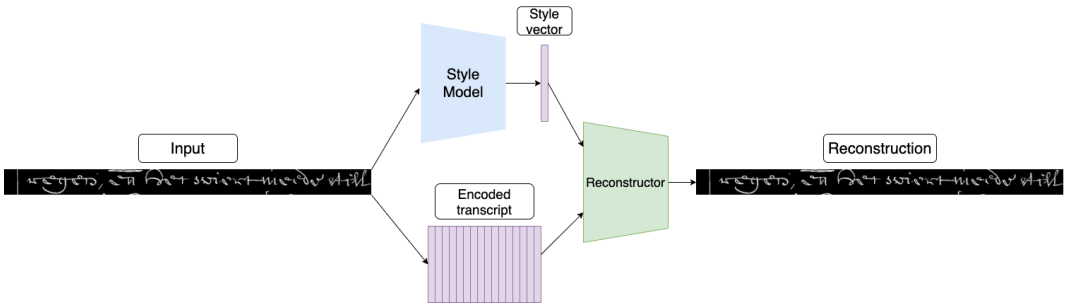


Figure 5.1: The reconstruction process with style model.

Once again we could use the Bayes' theorem to convert the distribution modeled by a reconstructor to the $p(y|l, s)$ probability. This allows us to continue using the developed architecture testing method and verify, whether a reconstructor with style

model could perform better in modeling transcript from a manuscript's line.

5.1 Simplified setup

Processing an image of entire line would be problematic - the ScribbleLens line samples vary in width and number of contained letters. Adjusting the dataset to our task would be a demanding task that might prove to be useless, if the style model did not benefit. Therefore we decided to use the QMNIST [13] dataset to simplify the reconstructor's task. QMNIST is composed of single channel, 28×28 images of handwritten digits (see figure 5.2), from the well known MNIST [14] set, but it contains some detailed metadata about each picture. Primarily, every image is labeled with the id of its author. We can therefore take k fixed size samples written by author a and pass each of them to the style model to get k style vectors $s_0, ..., s_{k-1}$. By computing the mean of these vectors, we may obtain an overall style vector s_a describing the author.

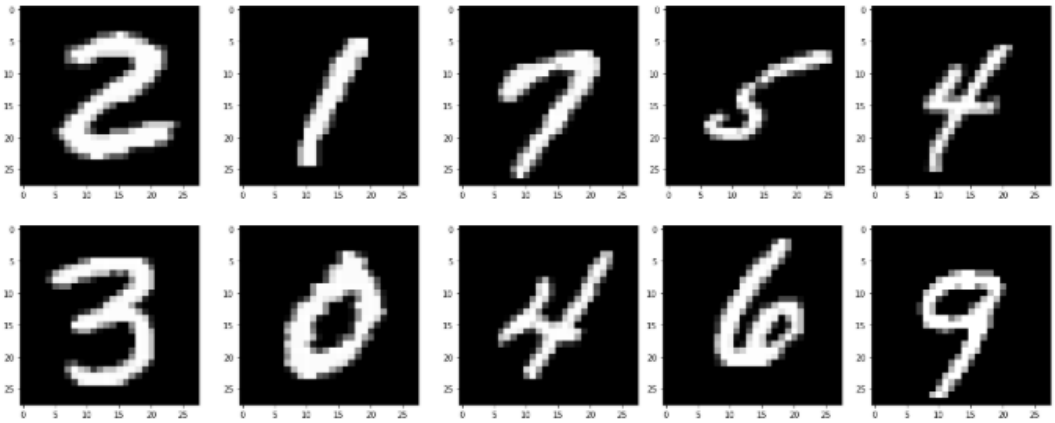


Figure 5.2: Samples from QMNIST dataset of digits written by the same author.

Not only can we use the QMNIST dataset to simplify the computations, but also we can evaluate a reconstructor on an image of single digit instead of a whole line (evaluating on a single digit picture is equivalent to a line with just one character). During all the foregoing experiments we computed $(l|y\{c_{i_j}\})$ - only one symbol was changed to assess the reconstructor's performance. There is no need to adjust the computations to the new setup therefore, moreover processing a 28×28 image is much faster.

The QMNIST dataset contains a lot of samples - there are over 400000 images available. However, the number of images written by each authors varies significantly. During the training an adjusted variant of the set was used to ease constructing a sample - each of N writers was an author of n_a samples (we used $n_a = 2$). When asking for i-th element of the dataset, the author number $j = \lfloor \frac{i}{n_a} \rfloor$ was selected. Af-

terwards from all the digit images $x_{j_0}, x_{j_1}, ...$ written by the j-th author we randomly pick a target image x used as the reconstruction objective and k images used as an input for the style model. In such setup, a reconstructor is fed with the average style vector s_j and the one-hot vector y representing the digit d written on the x picture. The reconstruction process is presented on figure 5.3.

The adjusted QMNIST dataset was split into train and test set by dividing the authors indices. A test sample is therefore written by an author not seen during the training process.

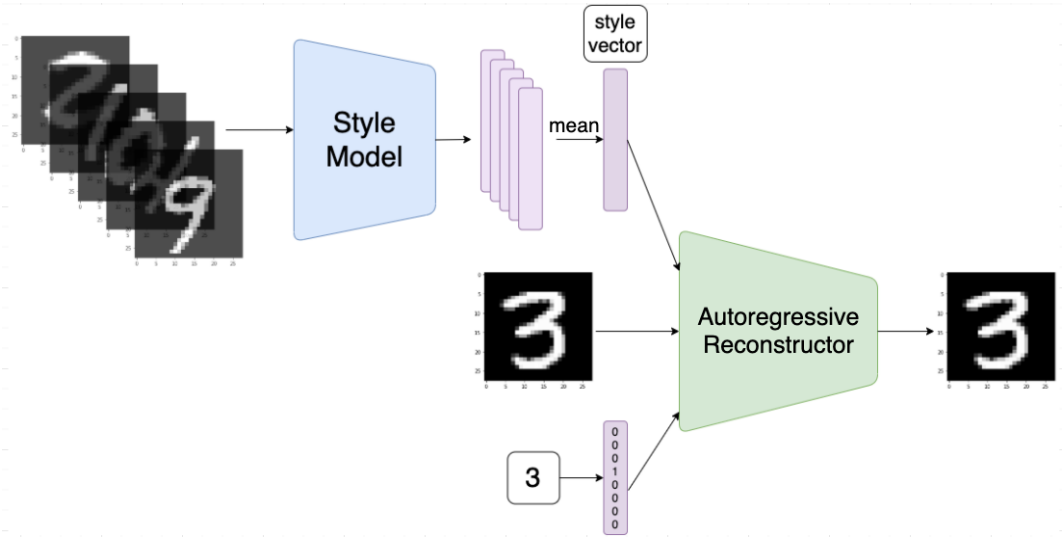


Figure 5.3: The reconstruction process for QMNIST dataset. A style model produces the style vector basing on k random samples of the author. The reconstructor is also fed with the one-hot vector of the digit presented on target image x and models its reconstruction.

5.2 Experiments with style model

To verify whether the style modeling idea could be beneficial, we conducted several experiments. In most of them the style model was a convolutional network with two hidden layers that fed with 28×28 image produced a 144 dimensional vector. The reconstructor used for most of the computations was a PixelCNN, as results from the previous section induced to develop this architecture. Initially we wanted to test whether the reconstructor was able to produce samples related with a particular author's style. The style model was fed with $k = 9$ writer's images and the PixelCNN was optimizing the loss function:

$$L(x_T^i, R(y_T, S(x_0^i, x_1^i, ..., x_8^i; \Theta_S); \Theta_R))$$

to maximize the reconstruction probability. In the equation above R, S are the reconstructor and style model respectively, Θ_R, Θ_S are their parameters, x^i is the

sample written by i-th author and y_T is a transcript (one-hot of the presented digit) of target image x_T^i.

After the training we asked the reconstructor to generate an image of digit d with respect to the style vector s computed for a random author. Figure 5.4 shows an exemplary obtained result, and it is satisfying. The generated samples of digit 7 (bottom rows) are definitely related with each author's style and look similar to samples of 7 written by the author used to compute the style vector.

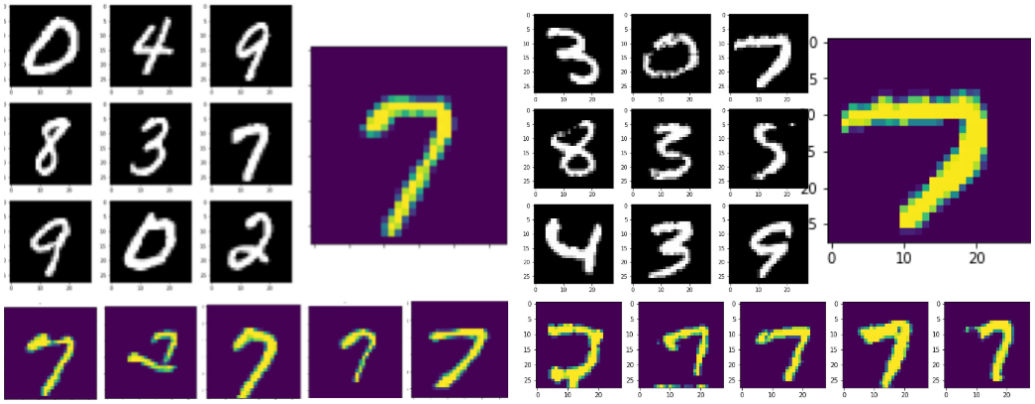


Figure 5.4: Generated samples of digit "7". The squares of 9 pictures present images written by a single author a_i used to compute a style vector s_i. The images on the right are samples of "7" written by the writer a_i. Below there are rows of 5 images generated by the reconstructor with respect to the corresponding style vector s_i.

5.2.1 Style model input images number

The initial computations showed that capturing a style with dedicated model S might be beneficial. A style vector s was obtained by taking a mean of $S(x_0^i), S(x_1^i), ...S(x_8^i)$ vectors. In the next step we trained the architectures $StylePixelCNN_k$ containing a style model S_k and a PixelCNN reconstructor R_k for different number k of images used as the style model's input. For each value of k from $\{1, 2, 4, 8, 16, 32\}$ set a reconstruction training was conducted.

Figure 5.5 presents the reconstruction loss change over the training for each model. The networks' parameters were optimized with Adam algorithm [15] and performed best, when the learning rate for each optimizer was changed twice - at the beginning it was set to 10^{-4}, after 70 epochs of training it was increased to 10^{-3} for another 70 epochs and eventually it was reduced to 10^{-4}. The sudden spikes of reconstruction loss at the chart was caused by reinitialization of each optimizer. Increasing a learning rate is not a common approach, as it usually leads to network deregulation. However, a learning rate that was neither constant (one of $10^{-4}, 10^{-3}$), nor a decreasing-only produced a better model.

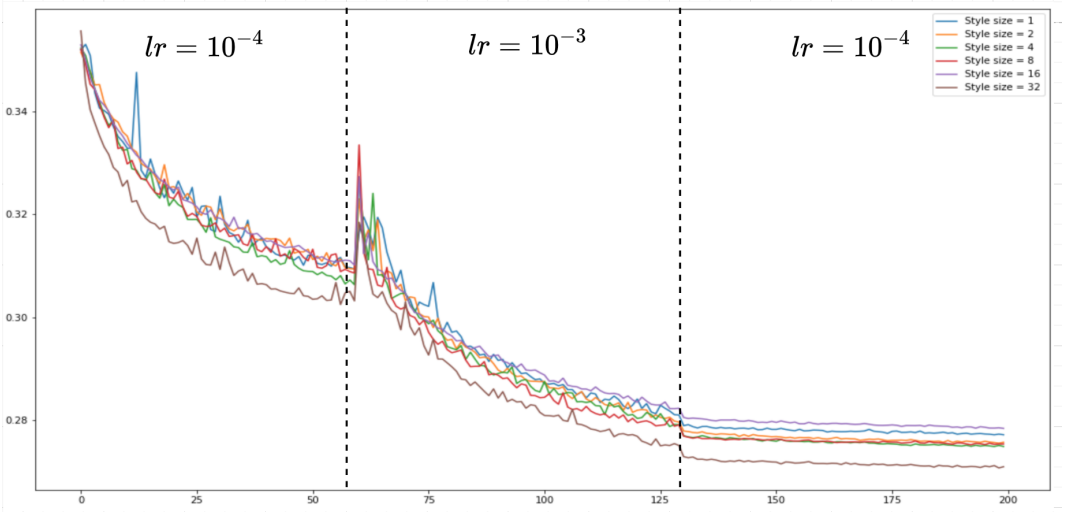


Figure 5.5: Reconstruction loss (nats/pix) over the training for models with different number of images passed to the style model. The learning rate was changed twice during the training, once from 10^{-4} to 10^{-3} and then back to 10^{-4} with fully reinitialized optimizers, hence the sudden change is visible.

Even though one of the reconstructors ($StylePixelCNN_{32}$) seems to be performing better than the others as its reconstruction loss is lower, during each iteration of training the order of models differed, but the nats/pix values were close to each other, therefore we assume that regardless of k value, reconstructors with style model perform similarly. Table 5.1 presents the test samples reconstruction result for each $StylePixelCNN$ model. Addition of the S_k is beneficial, especially for $k \geqslant 2$.

After the learning process all the reconstructors were used to generate a sample of every digit 0...9. For digit d an author a_d was sampled, and each of R_k generated an image with respect to style vector $S_k(x_0^d, ..., x_k^d)$ and a digit's one-hot y. Obtained pictures are presented on figure 5.6.

5.2.2 Every digit sample

Samples used as an input for each of S_k style model have been randomly picked from all the available images for a given author so far. This approach is especially well suited for semi-supervised learning, as no labels are needed to compute a style vector. However, if some labeled data is available, we may select the images passed to the style model. The dataset was therefore modified so each author a_i had assigned images $x_0^i, ..., x_9^i$ presenting each of 0...9 digit. A data sample for the a_i writer contains a randomly picked target image x, its label y, and the $x_0^i, ..., x_9^i$ images, that are used to compute the style vector s_i.

Necessity of having the labeled data is definitely a drawback of such an approach.

Model	Reconstruction loss (nats/pix over 2500 samples)
PixelCNN without style model	0,2785
PixelCNN with S_1 style model	0,2743
PixelCNN with S_2 style model	0,2696
PixelCNN with S_4 style model	0,2689
PixelCNN with S_8 style model	0,2684
PixelCNN with S_{16} style model	0,2681
PixelCNN with S_{32} style model	0,27
PixelCNN with S_{0-9} style model	0,2629

Table 5.1: Reconstruction loss for different reconstructors. All of the models were conditioned on target image one-hot label. The first row presents a value for a casual PixelCNN without a style model and is a baseline for the $StylePixelCNN$ reconstructors.

However, all we require is a single sample of each digit. In case of ScribbleLens dataset we would need 68 labeled character pictures, but this number of images assigned to a writer seems fair, if our goal is to transcript a whole document written by the author.

Figure 5.7 presents the reconstruction loss of $StylePixelCNN_{0-9}$ reconstructor trained on the modified dataset, which had a similar trend to the losses of previously tested models, however, during the final evaluation, the $StylePixelCNN_{0-9}$ performed better than others. The samples of each digit generated by the model resemble the $x_0^i, ..., x_9^i$ images. These results show that passing a sample of every character to the style model improves the reconstruction.

5.2.3 Classification with styled reconstructors

Knowing that the style model improves the performance in reconstruction task we would like to see if it's going to be an advantage in transcript classification. As previously mentioned, we believe that providing a writer's style information by dedicated model would allow the reconstructor to require more character specific features from the latent representation. To verify this concept we should once again use the Bayes' theorem and classify the labels basing on the image.

In foregoing classification experiments the ScribbleLens dataset was used. Adjusting the computation to QMNIST images of a single digit is straightforward - previously $p(y\{c_i\}|l)$ was modeled - a probability of line's transcript with a single replaced character. As the whole transcript of QMNIST image is an id of the digit,

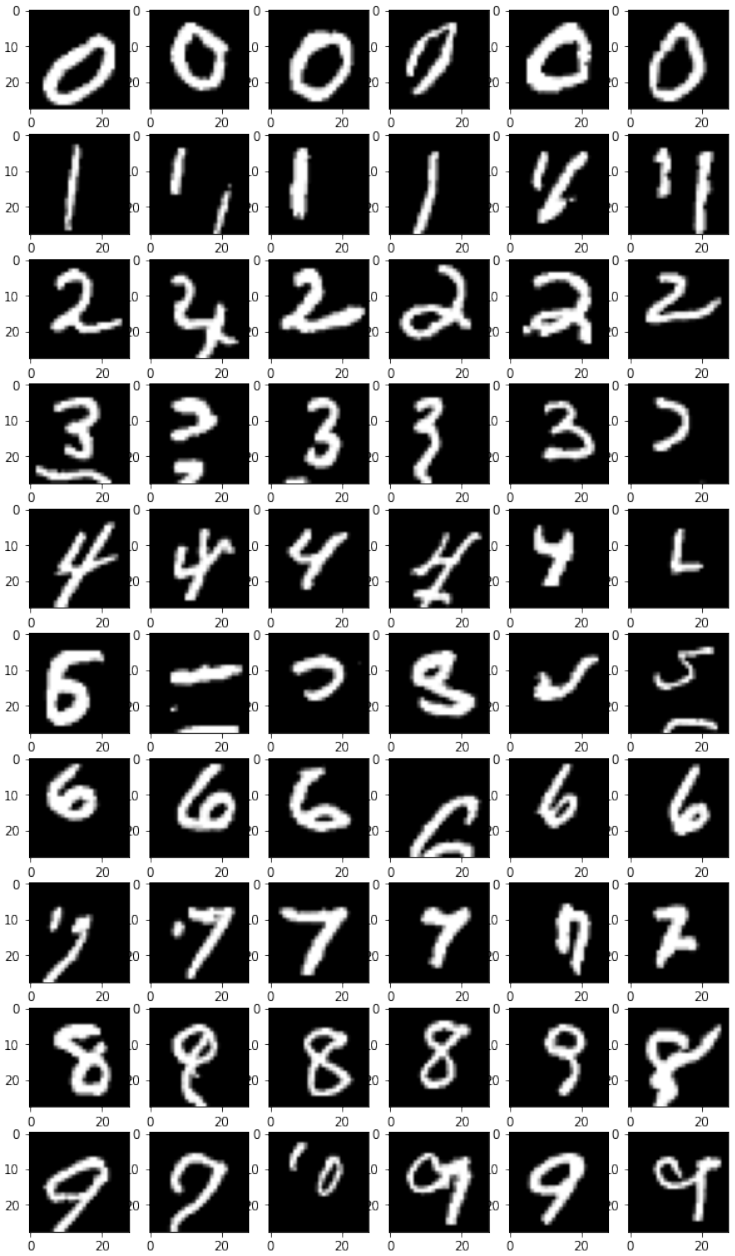
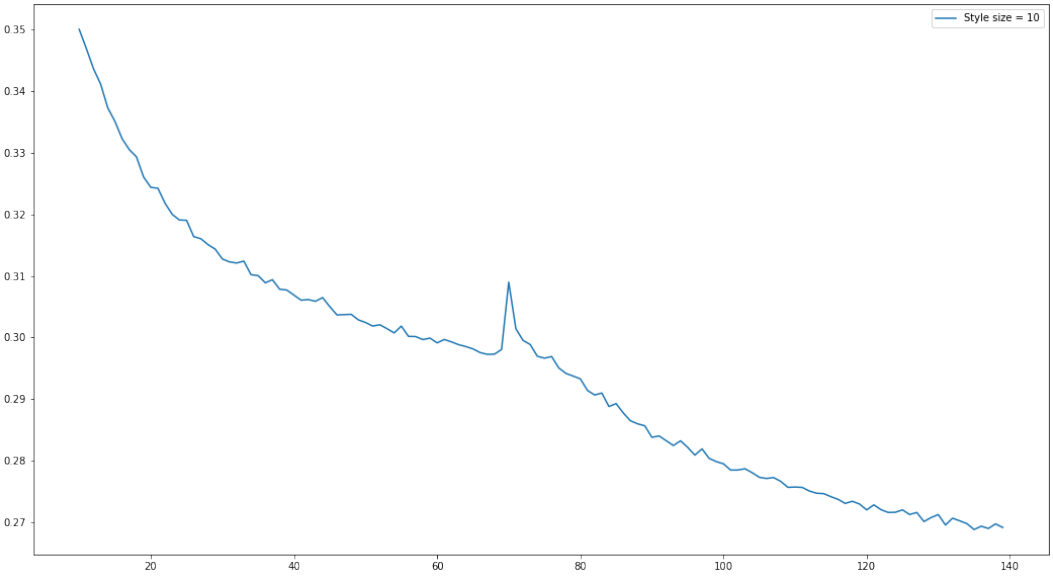


Figure 5.6: Samples generated by each of styled reconstructor $StylePixelCNN_k$. Columns present images produced by each model.

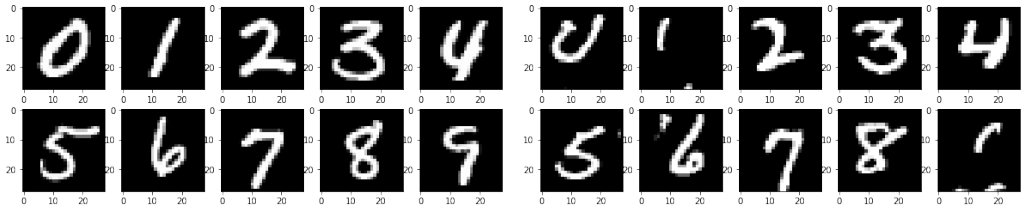
we are going to predict $p(d|x, s)$, where d is each of 0...9 digit, x is a QMNIST sample and s is a style vector.

Computing the probability is made in the same way as for ScribbleLens line - for the input image x^i presenting digit d_T written by author a_i we compute each of $p(x^i|d, s_i)$ values, where s_i is a style vector assigned to a_i writer. Afterwards using the Bayes' theorem we transform these values to $p(d|x^i, s_i)$ with the Softmax function. If the most likely digit d is equal to d_T, the classification is assumed correct.

Classification results for each reconstructor are presented in table 5.2. The



(a) Reconstruction loss (nats/pix) over the training



(b) Images used as the style model input.

(c) Generated samples of all digits.

Figure 5.7: **Top:** The reconstruction loss during $StylePixelCNN_{0-9}$ training. **Bottom:** Images used to compute the style vector s and samples by $StylePixelCNN_{0-9}$.

baseline is a PixelCNN conditioned on digit's one-hot vector during the learning process and trained on the QMNIST adjusted to $StylePixelCNN$ models - for each author it processed just n_a samples. All the $StylePixelCNN_k$ models perform better than a baseline, therefore modeling writer's style could be beneficial in our global goal - producing a transcript of the written text. Somewhat surprisingly, despite the best result in reconstruction, the accuracy of $StylePixelCNN_{0-9}$ is the lowest one, so it is not suited for our task.

The last column of table 5.2 contains the accuracy of each model for "fake author" style vector. In this experiment, a prediction was made basing on $p(d|x^i, s_j)$ value, where s_j is a style vector of randomly picked author $a_j \neq a_i$. A classification results are significantly worse, especially for models using many images to compute a style vector. Such a drop of accuracy is a bit of surprise, we could expect that style vectors s_i and s_j are rather close, as they are result of processing some similar data. It also suggests that a style vector is an important part of the computation process. We will take a closer look at this issue in the following section.

Model	Classification accuracy	Classification accuracy when "fake author" style vector was used
Standard deviation of the experiment	*0,0321*	*0,1722*
PixelCNN without style model	98,032%	—
PixelCNN with S_1 style model	98,11%	96,28%
PixelCNN with S_2 style model	98,07%	94,23%
PixelCNN with S_4 style model	98,19%	91,31%
PixelCNN with S_8 style model	98,18%	89,17%
PixelCNN with S_{16} style model	98,41%	87,87%
PixelCNN with S_{32} style model	98,1%	85,81%
PixelCNN with S_{0-9} style model	97,75%	87,27%

Table 5.2: Classification accuracy for $StylePixelCNN$ reconstructors. The standard deviation is common for all the models, computed with results from 4 test iterations. The PixelCNN without a style model was evaluated on QMNIST adjusted to $StylePixelCNN$ computation process.

5.2.4 Multiple "fake" authors style

We have already seen that style vector is relevant for the reconstructors - during the "fake author" test, accuracy of $StylePixelCNN_{32}$ reconstructor taking 32 samples as an input for the style model S_{32} dropped from over 98% to 86%. Such a behaviour shows the importance of a style vector and stimulates to analyze the issue.

In all the foregoing "fake author" evaluations there have been just one fake author - samples $X_S^i = \{x_0^i, ..., x_{31}^i\}$ passed to a style model were replaced with samples written by the "fake" author a_j. We could check whether replacing the X_S^i images with samples from many "fake" authors also lowers a model's performance. When all the 32 samples written by the a_i author were replaced with 32 samples $x_0^{j_0}, ..., x_{31}^{j_{31}}$, each one written by a different authors $a_{j_k} (k = 0, ..., 31)$, the model's classification accuracy was just slightly lower - it dropped to 96,64%.

Style vector used in this setup may be interpreted as an "averaged style" over several authors. A small accuracy drop may indicate that there are many authors represented with close style vectors and some of randomly picked writers a_{j_k} have style vectors s_{j_k} similar to the s_i. The constructed, averaged style vector is therefore closer to the s_i and doesn't damage the model's performance so much.

5.2.5 Probability heatmaps

When modeling a QMNIST sample each of reconstructors produces a $28 \times 28 \times 16$ tensor P, representing a probability of each pixel's value. During digit classification, we compute $p(x|y_d, s)$, where y_d is set to each 10 dimensional one-hot vector.

Obtained tensor represents pixels probabilities if the x is presenting digit d. We can visualize the likeliness of the x image as a 28×28 heatmap H_d^x, by taking $H_d^x[w,h] = P[w,h,x_{w,h}]$ value for each position $[w,h]$.

Such a heatmap would present how a model is "surprised" with the image at each position. If it would capture some digit specific regions, we could notice that one of H_d^x is significantly different. We computed the heatmaps for each of $StylePixelCNN_k$, they are shown on figure 5.8. Each row represents one of the reconstructors, while columns are the H_d for consecutive $d = 0, ..., 9$.

Regardless of d digit and k parameter, a reconstructor assigned low probability to the digit's contour. As PixelCNN is an auto-regressive model - it predicts a pixel based on the preceding ones, such a behavior could be expected - the reconstructor expects following pixels to be consecutively dark or bright, as they construct either background or the digit, and only the change regions are surprising.

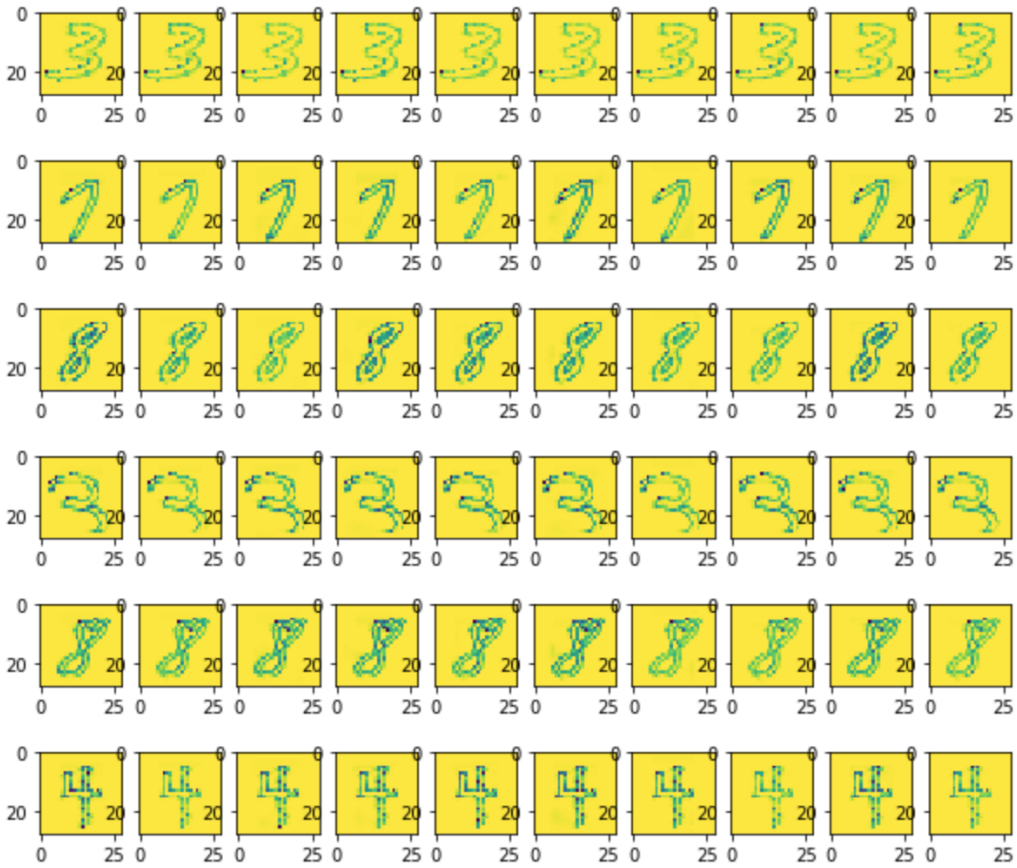


Figure 5.8: Heatmaps H_d^x of image probability $p(x|y_d, s)$ for different $StylePixelCNN_k$ reconstructors. Each row presents a $p(x|y_d, s)$ computed by one of the models for randomly sampled x image. Columns correspond to consecutive d conditioning.

5.2.6 Style vectors from all author's samples

All the $StylePixelCNN$ models we have developed have been trained to compute a style model for k given images. The maximal value of k was 32, however for most writer there are much more samples available. As obtaining a style vector is done by processing each of the k input images individually, and computing the mean of k produced values, every style model S_k can easily compute a style vector s from $m \neq k$ pictures.

It seems that each style model should produce the most valuable style vector s_i, when all the available writer's a_i samples are used as an input. Such a s_i vector would most accurately represent the author's style. We decided to take all the trained $StylePixelCNN_k$ models and evaluate them on label classification task in such scenario.

Table 5.3 presents the results of digits classification by each of $StylePixelCNN_k$ reconstructors on a test set containing samples written only by authors not used during the learning process. Passing all the author's samples turned out to be beneficial, especially for models that used few images to compute a style vector during training - $StylePixelCNN_1$ and $StylePixelCNN_2$ are performing much better in this setup. Obtained results were more varied, even though a style vector for each author was constant, the only difference between test iterations were target samples x. Moreover, during one of the tests evaluation the $StylePixelCNN_1$ reached the highest obtained accuracy - 98,72%.

Model	Classification accuracy
Standard deviation of the experiment	*0,0755*
PixelCNN with S_1 style model	98,41%
PixelCNN with S_2 style model	98,33%
PixelCNN with S_4 style model	98,23%
PixelCNN with S_8 style model	98%
PixelCNN with S_{16} style model	98,23%
PixelCNN with S_{32} style model	98,24%

Table 5.3: Classification accuracy for $StylePixelCNN_k$ reconstructors, when a style vector s_i is computer for author a_i all samples. 4 test iterations were evaluated.

5.2.7 Accuracy change for different style model input images

The next experiment tested the effect of style model's input images selection on a sample classification accuracy. For a QMNIST image x^i presenting a digit written by a_i author, each styled reconstructor $StylePixelCNN_k$ models:

$$P(x^i|y, s_i)$$

where y is a one-hot vector of the digit d presented on the x^i picture and s_i is a style vector computed as a mean of $S_k(x_0^i), ..., S_k(x_{k-1}^i)$ values. The $x_0^i, ..., x_{k-1}^i$ images are randomly picked from all the writer's samples, therefore a style vector s_i describing the a_i author differs every time one of the writer's sample is classified.

To see how the classification results may vary for different style vectors, we can compute n_s style vectors:

$$s_i^{[0]}, s_i^{[1]}, ..., s_i^{[n_s-1]}$$

where each of $s_i^{[j]}$ vectors is representing a_i author, but different input images $x_0^i, ..., x_{k-1}^i$ were used every time. Afterwards we compute:

$$\begin{aligned} p^{[0]} &= p(y|x^i, s_i^{[0]}) \\ p^{[1]} &= p(y|x^i, s_i^{[1]}) \\ &... \\ p^{[n_s-1]} &= p(y|x^i, s_i^{[n_s-1]}) \end{aligned}$$

probabilities and take the maximal difference $maxdiff = p^{[j]} - p^{[l]}$. The difference determines how much (in percentage points) the classification accuracy may vary, when different style model describing the same author is used. For each $StylePixelCNN_k$ reconstructor we computed *maxdiff* value for 150 different samples from QMNIST dataset with parameter $n_s = 10$.

Table 5.4 presents the obtained results. The average *maxdiff* value is rather small - for every model but $StylePixelCNN_1$ the style vector selection may change the final probability of the ground truth label by no more than 1,53%. The value for $StylePixelCNN_1$ shows that the reconstructor is sensitive to the style vector change. The maximal *maxdiff* value shows, that regardless of the k, with a proper style vector each of $StylePixelCNN_k$ may change the prediction.

The highest *maxdiff* - 99.67% was obtained during image x presenting the 0 digit classification (presented on figure 5.9). The highest $p^{[j]}$ probability for this image was 99,77%, while the lowest one was just 0,09%. The $PixelCNN_1$ was 99,25% sure, that the x is presenting the 6 digit in that case.

Number k of samples used as style model input	Average *maxdiff* value	Maximal *maxdiff* value
1	8,03	99,67
2	0,72	47,45
4	1,50	54,13
8	1,53	76,15
16	1,49	73,76
32	1,38	50,46

Table 5.4: *maxdiff* values for different $StylePixelCNN_k$ reconstructors. Each model computed the *maxdiff* 150 times with number of style vectors for each sample $n_s = 10$

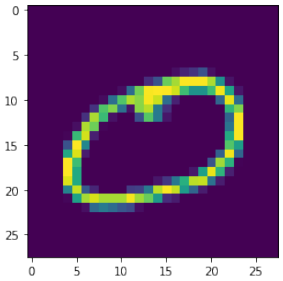


Figure 5.9: A sample for maximal *maxdiff*.

5.2.8 Style vectors visualization

To get a better understanding of the style vectors we may extend the style model S architecture with a single linear layer, that would map s_i to a 3 dimensional $s_i^3 \in \mathbb{R}^3$. Vectors obtained in this way may be visualized in the coordinate system. The style model S was replaced with the new architecture S^3 in the styled PixelCNN reconstructor. Just like before, the S_k^3 may compute a s_i^3 vector based on k input images. We decided to train the new $StylePixelCNN_k^3$ models for 3 values $k = \{1, 8, 32\}$.

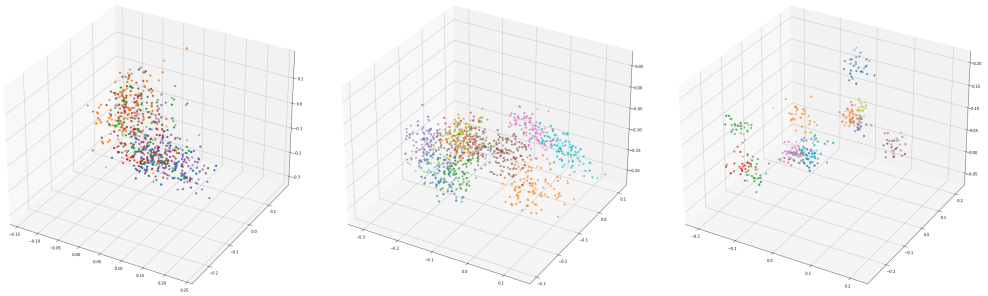
When the learning process was completed, first of all, we had to verify whether the accuracy drop also occurs in case of 3 dimensional style vector, as it could be too small to capture some valuable features. Table 5.5 shows the QMNIST samples classification accuracy (predicting $p(y|x, s^3)$ on the basis of Bayes' theorem) and reconstruction loss for each $StylePixelCNN_k^3$. Despite the style vector's dimensionality reduction, replacing a true s_i^3 vector with a s_j^3 representing a fake author a_j is still harmful for the model's performance. Moreover, obtained classification and reconstruction results are as good as before, therefore the style information can be captured in a much simpler space.

Model	Classification accuracy	Classification accuracy when "fake author" style vector was used	Reconstruction loss (nats/pix)
$StylePixelCNN_1^3$	98,28%	96,52%	0,2716
$StylePixelCNN_8^3$	97,8%	89,4%	0,2695
$StylePixelCNN_{32}^3$	98,4%	89,04%	0,2638

Table 5.5: Accuracy of QMNIST images classification and reconstruction for each of $StylePixelCNN_k^3$ reconstructor.

As style vector's dimensionality reduction did not cause any drastic change in reconstructor performance, we can go through visualizations. For each of $a_0, ..., a_{12}$ authors 30 style vectors $s_{i_0}^3, ..., s_{i_{29}}^3$ were computed by each of S_k^3 style models. Afterwards the obtained s^3 vectors were treated as points and placed in the coordinate

system. Figure 5.10 presents the distribution of style vectors produced by each style model. Each color represents the $s^3_{i_0}, ..., s^3_{i_{29}}$ vectors obtained for the same writer a_i.



(a) S^3_1 (b) S^3_8 (c) S^3_{32}

Figure 5.10: 3 dimensional style vectors s^3_i produced by each of style model S^3_k. Each color represents style vectors s^3_i computed for the same a_i author.

The style vectors computed by S^3_1 (first picture on the figure) compose an unstructured blob. It seems impossible to divide them by authors, therefore replacing a s^3_i with s^3_j is slightly disturbing. Style vectors produced by the S^3_8 (the middle image on the figure) are much more structured, and each author's vectors are gathered. The S^3_8 is indeed modeling some features related to every author's writing style, even though any style related term was not introduced to the $StylePixelCNN^3_8$ reconstructor's loss function.

The style vectors' division is even more visible at the visualization of vectors constructed by the S_{32} (right picture on the figure). Vectors computed for each authors are placed in some individual space, separately from the others. It explains the significant performance degradation after replacing the a_i writer's style vector with a "fake" one.

Chapter 6

Learned samples embedding

All the foregoing reconstructors were modeling a x image given its transcript y - either $p(x|y)$ in case of simple decoders or $p(x|y,s)$ when styled reconstructors were used. The transcript was passed in form of one-hot vectors representing the respective characters. Such a reconstructor's input was chosen, as it is our desired latent representation produced by an encoder, as our global goal is to use the latent vectors for characters classification task. Evaluating the decoders on these fixed conditioning vectors showed how each model processed such input, and whether it could expect similar representation from the encoder.

To verify whether a reconstructor would actually demand a conditioning related to character's labels, we can allow it to assign an individual vector $e \in \mathbb{R}^{n_e}$ for each QMINST sample, that would be used instead of a digit one-hot conditioning. This embedding would be tuned by the reconstructor's gradient, similarly to the approach used in word2vec [16] to construct words' representation. Before the training a $N \times n_e$ tensor E is allocated, where N is the used QMNIST dataset strength. When i-th sample is processed, $e_i = E[i]$ is used instead of one-hot vector. Afterwards, a styled reconstructor R tries to recreate the input image by modeling:

$$p(x_i|e_i, \overline{s_i})$$

where $\overline{s_i}$ is a style vector computed for the author of i-th sample x_i. During back-propagation the e_i is optimized then. Figure 6.1 visualizates the described process.

In a perfect scenario, the reconstructor would tune the embeddings to represent a label of digit presented on each image - the label would be the most desirable feature of each sample. If $n_e = 10$ it could even turn the e_i into a one-hot vector. A reconstructor would have to figure out, that each of the N images represents one of 10 classes, what's more than unlikely. However, if the model would cluster the N images in $c \geqslant 10$ groups containing images of only one digit, we could easily determine a sample's label basing on its embedding. Such situation is also exactly what we want, as predicting an image label would be trivial.

To determine, whether a reconstructor succeeded in tuning embeddings to represent digits labels, after the learning process we construct a very simple (one hidden layer) embedding classifier. The classifier predicts $p(d_i|e_i)$, where d_i is a label of the x_i picture. The embeddings e_i remain unchanged during this process, we use the values optimized during a reconstructor training. If such a small classifier would be capable of distinguish digits from the embeddings, a reconstructor would definitely focus on storing d_i information in the e_i vector.

Reconstructor type	Reconstruction loss (nats/pix on training test)
PixelCNN with no conditioning	0.2936
PixelCNN conditioned on one-hot vector	0.2843
$StylePixelCNN$ conditioned on one-hot vector	0.2811
$StylePixelCNN^E$ conditioned on $e \in \mathbb{R}^{10}$	0.2824
$StylePixelCNN^E$ conditioned on $e \in \mathbb{R}^{32}$	0.2751
$StylePixelCNN^E$ conditioned on $e \in \mathbb{R}^{128}$	**0.2362**
$StylePixelCNN^E$ conditioned on $e^S \in \mathbb{R}^{10}$	0.2832
$StylePixelCNN^E$ conditioned on $e^S \in \mathbb{R}^{32}$	0.2771
$StylePixelCNN^E$ conditioned on $e^S \in \mathbb{R}^{128}$	0.2734
$StylePixelCNN^E$ conditioned on $e^{GS} \in \mathbb{R}^{10}$	0.2861
$StylePixelCNN^E$ conditioned on $e^{GS} \in \mathbb{R}^{32}$	0.2869
$StylePixelCNN^E$ conditioned on $e^{GS} \in \mathbb{R}^{128}$	0.2841

Table 6.1: Reconstruction loss on the QMNIST subset used during training process of embedding-based reconstructors.

6.1 Computation process

During all the upcoming embedding experiments the $StylePixelCNN_8$ reconstructor was used. We can denote the embedding-based variant of the model as $StylePixelCNN_8^E$. The dataset had to be adjusted, so 60000 samples from QMNIST were chosen as the target images. To compute a style vector $\overline{s_i}$, 8 random images from all the digits written by a_i were picked. Moreover, an embedding classifier could be only learned

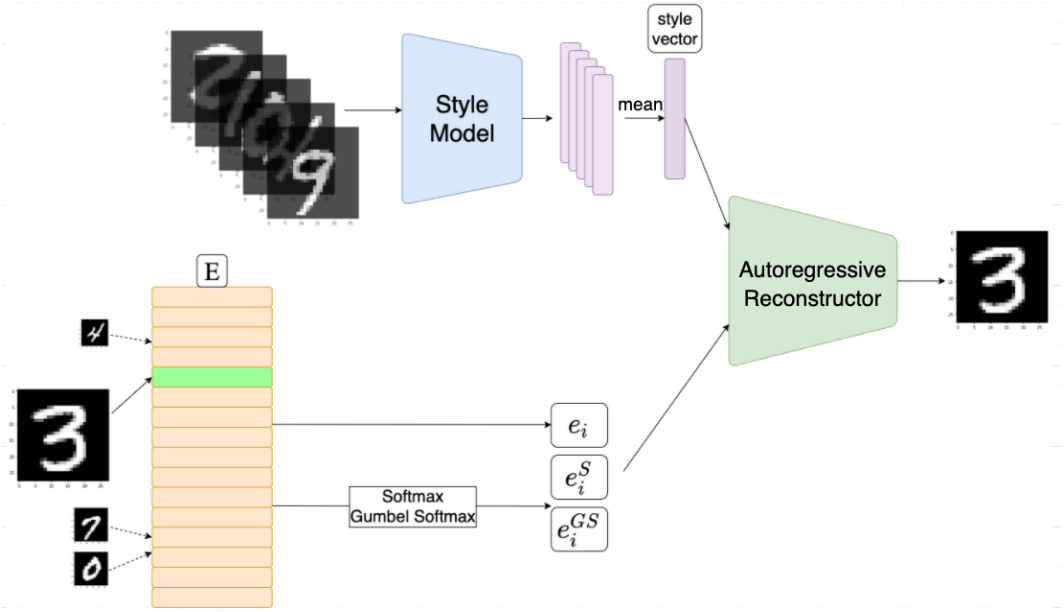


Figure 6.1: A visualization of embedding-based reconstructor architecture. Tensor E keeps the learnable embedding weights, that are either passed directly as e_i or a Softmax/Gumbel Softmax is applied before.

on the mentioned 60000 QMNIST samples, as the E embeddings contained only their e_i vectors. The E was therefore divided into a train and test set, containing 50000 and 10000 elements respectively.

To compare a reconstructor using the embeddings with a one conditioned on digit's one-hot vector, the only comparable metric was a reconstruction loss on train set images. We computed this value for the already developed models then - they can be found in table 6.1. The embedding-based reconstructor cannot be used in the digit classification task, however modeling an image from a label is not the objective now, in fact it's quite opposite - we want to model a label from the image.

6.1.1 Embedding dimensionality

In the first experiment different values of n_e parameter were tested. We trained a $StylePixelCNN_8^E$ for $n_e \in \{10, 32, 128\}$ and classified the obtained embeddings afterwards. Results are presented in table 6.2. The embeddings classification accuracy is rather disappointing, the reconstructor did not discover the underlying division of the samples into the digit classes.

A reconstruction loss for each trained $StylePixelCNN_8^E$ can be found in table 6.1. The value for $n_e = 128$ is incomparably better than a result of any other reconstructor. It could mean that the $StylePixelCNN_8^E$ decided to use the e_i vector for some kind of x_i compression. When reconstructing the image as $p(x_i|e_i, \overline{s_i})$ it had an access to many features of the target image then.

Embedding dimensionality	Embedding classification accuracy
10	27%
32	39%
128	32%

Table 6.2: Accuracy of predicting labels d_i from e_i embeddings and the reconstruction loss for different values of n_e parameter.

6.2 Constraining embeddings

We could constrain the embedding, so the reconstructor could not store so many target image features in it, and try to obtain e_i embedding more similar to a one-hot vector. For this purpose we apply a Softmax function on the embedding weight, so elements of the e_i would sum to 1. The embedding for the i-th dataset sample would be then:

$$e_i = Softmax(E[i])$$

In fact applying the Softmax dosn't limit the information that can be stored in the vector, however a reconstructor could not figure it out, and would focus on other features of a sample.

If we want to make the e_i vector look like one-hot even more, we can replace the Softmax with the Gumbel Softmax [17] - a function that takes the vector of probabilities/scores v_p and produces an approximation of discrete sampling v_s with respect to the v_p values. Just like in the classic Softmax, the elements of v_s sum up to 1. Most importantly, the function is differentiable. Gumbel Softmax is parameterized with τ - a temperature value, that determines how close the output should be to real discrete sampling. Figure 6.2 presents the Gumbel Softmax output vectors for different τ values.

Gumbel Softmax would actually limit the capability of embedding vectors, as e_i would be sampled with respect to the $E[i]$ weights. If a low τ value is used, the e_i would resemble a one-hot vector.

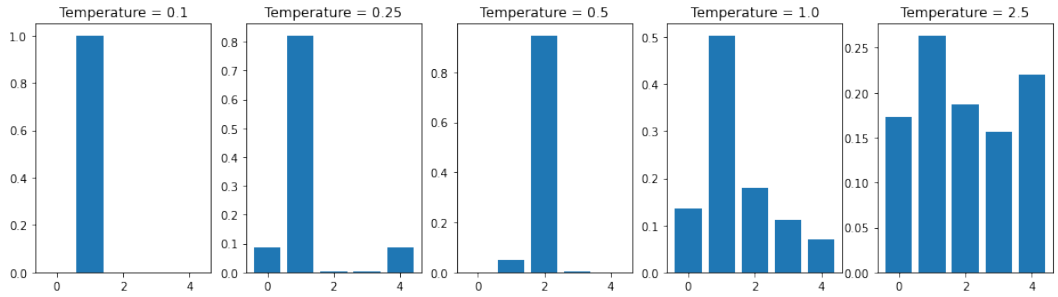


Figure 6.2: Gumbel Softmax output vectors for the same input $v_p = [1, 2, 1, 1, 1]$ when different values of temperature τ is used.

6.2.1 Softmax and Gumbel Softmax embedding evalutation

For embedding $e_i^S = Softmax(E[i])$ three variants of e_i dimensionality were tested - $n_e \in \{10, 32, 128\}$. For each embedding size we learned a $StylePixelCNN_8^E$ that modeled $p(x_i|e_i^S, \overline{s_i})$ probability. After the training process we tried to predict the d_i labels for embedding vectors e_i^S. The embedding classifier was performing poorly (see table 6.3), the accuracy for $n_e \in \{32, 128\}$ is a random one.

The reconstruction loss was the lowest for $n_e = 128$ (table 6.1), therefore $StylePixelCNN_8^E$ made use of e_i^S vectors. Even though a simple classifier did not manage to find any pattern in these vectors, digit labels were a relevant part of image reconstruction task.

As the embedding classifier managed to capture some label related features only for $n_e = 10$, we used this value when training a reconstructor conditioned on $e_i^{GS} = GumbelSoftmax(E[i])$. We tested 3 values of temperature $\tau \in \{0.1, 0.5, 1.0\}$. Unfortunately, an embedding classifier was unable to model the $p(d_i|e_i^{GS}, \overline{s_i})$ for any of the τ value. The embedding weights E were optimized with Adam algorithm with learning rate set to one of $\{10^{-3}, 10^{-4}, 10^{-5}\}$, however we did not managed to improve the performance. Moreover, obtained reconstruction scores were worse than results of a model conditioned on one-hot vectors (table 6.1).

Constraining the embedding e_i values did not force a reconstructor to store more label related features in them. Nevertheless the vectors were still beneficial during reconstruction task.

Embedding type	Embedding classification accuracy
Softmax, $n_e = 10$	28%
Softmax, $n_e = 32$	11%
Softmax, $n_e = 128$	11%
Gumbel Softmax, $\tau = 0.1$	10%
Gumbel Softmax, $\tau = 0.5$	12%
Gumbel Softmax, $\tau = 1.0$	15%

Table 6.3: Accuracy of predicting labels y_i from e_i^S and e_i^{GS} embeddings for different values of n_e and τ parameter.

6.2.2 Author's digit embeddings

Embedding vectors constrained by Softmax or Gumbel Softmax function did not store any features related to sample's label. However, if we limit the number of embeddings, we could check if the e_i^S and e_i^{GS} are able to capture this information at all.

Instead of keeping an individual e_i vector for each x_i sample, we assign 10 embedding vectors (one per each digit) to each author a_j. Obtaining the embedding vector for sample x_i that presents digit d written by author a_j could be done as:

$$e_{d,j} = E[10 \cdot j + d]$$

as writer's a_j samples begin at position number $10 \cdot j$ of embeddings tensor E. Figure 6.3 presents the computation process in this scenario.

In this experiment only 10 dimensional embeddings were used. For Gumbel Softmax once again we tested $\tau \in \{0.1, 0.5, 1.0\}$. Results of embedding classifiers for each variant are shown in table 6.4. The model predicting d labels basing on $e^S_{d,j}$ reached 91% accuracy, therefore embedding vectors representing the same digit were similar regardless of the author. As $e^S_{d,j}$ was not storing any sample-related features, the reconstructor could not use it to compress the target image and information kept in $e^S_{d,j}$ had to be more generalized.

Results of classifying the $e^{GS}_{d,j}$ vectors show, that the $StylePixelCNN^E_8$ reconstructor is not able to perform well when Gumbel Softmax with low temperature τ is used to construct an embedding vector.

Embedding type	Embedding classification accuracy
Softmax	91%
Gumbel Softmax, $\tau = 0.1$	15%
Gumbel Softmax, $\tau = 0.5$	42%
Gumbel Softmax, $\tau = 1.0$	60%

Table 6.4: Accuracy of predicting labels d from $e^S_{d,j}$ and $e^{GS}_{d,j}$ embeddings for different values τ parameter.

6.3 Using pretrained models to tune embeddings

We have seen that a reconstructor modeling $p(x_i|e_i, \overline{s_i})$ conditioned on learned embedding is rather not interested in storing features related to image's label in the embedding vector. However, the reconstruction error gets lower, when e_i vectors were introduced, especially if their dimensionality was higher.

Instead of tuning the reconstructor's weights form the scratch, we can use one of previously learned $StylePixelCNN$ models. It was trained to reconstruct a x_i image from a style vector of its author $\overline{s_i}$ and a one-hot vector y_i of the presented digit. Providing the label information is beneficial for the decoder, as the reconstruction loss gets much lower than the result of non-conditioned reconstructor (table 6.1). $StylePixelCNN^E$ initialized with parameters of such model could therefore optimize the E weights to make the e_i vectors closer to corresponding one-hot vectors.

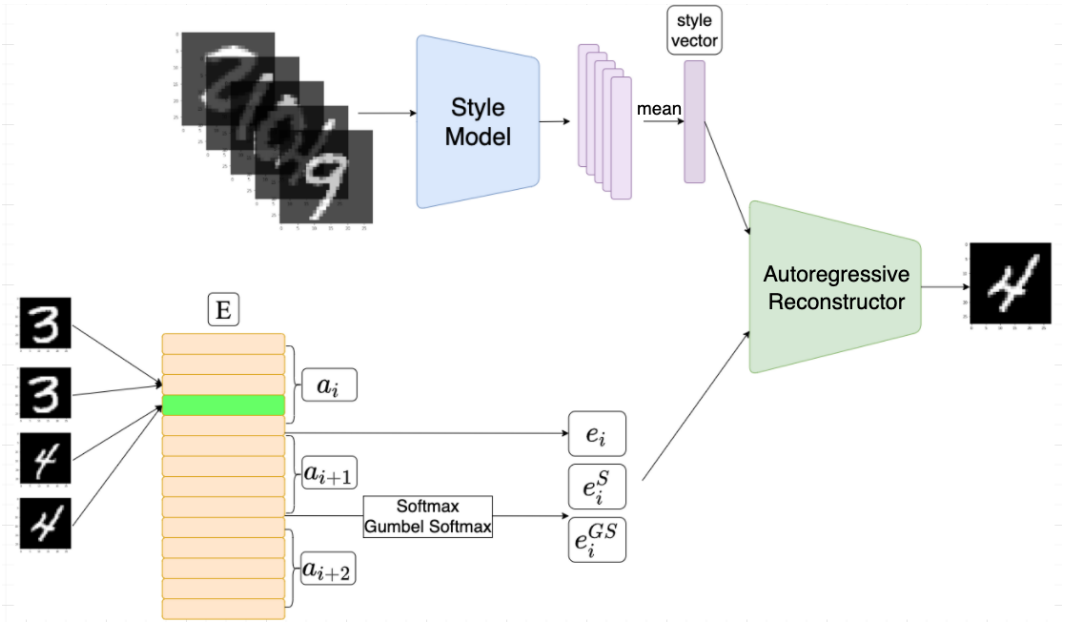


Figure 6.3: A reconstructor with per-author embedding. Images of the same digit written by the same author are represented by single embedding vector.

Both the PixelCNN reconstructor and a style model S_8 were initialized with the pre-trained parameters (Θ_R, Θ_S) then. During the training weights of each model's compontent (Θ_R, Θ_S, E) were optimized. The embedding vectors were the "constrained" ones - the Softmax and Gumbel Softmax with $\tau \in \{0.1, 0.5, 1.0, 1.5, 2.0, 2.5\}$ functions were applied to obtain $e_i \in \mathbb{R}^{10}$.

After each reconstructor training, an embedding classifier was used to verify, whether the label information is present in the e_i vectors. Accuracy for each reconstructor is presented in table 6.5. The obtained result for Softmax embeddings e_i^S is slightly better when pre-trained Θ_R, Θ_S were used. The reconstruction loss of this $StylePixelCNN^E$ model is a bit lower than the value obtained for $StylePixelCNN$ whose parameters were used to initialize the new reconstructor. When it was enabled to tune the embeddings, it found some more valuable features than the digit labels.

Results of classification the e_i^{GS} vectors are once again close to a random prediction. Increasing the τ improved the results, but as we could previously see, Gumbel Softmax output with such value of the τ parameter is not resembling a one-hot vector at all. Also none of the reconstructor performed better than the one-hot conditioned $StylePixelCNN$ in the image recreation task.

6.3.1 Using pretrained reconstructors and embeddings

When a reconstructor trained on one-hot vectors representing digit labels was allowed to optimize the embeddings, it did not decide to store the label information in e_i^S

Embedding type	Embedding classification accuracy	Train set reconstruction loss (nats/pix)
Softmax, $n_e = 10$	38%	0,2808
Gumbel Softmax, $\tau = 0.1$	11%	0,2858
Gumbel Softmax, $\tau = 0.5$	13%	0,2856
Gumbel Softmax, $\tau = 1.0$	15%	0,2844
Gumbel Softmax, $\tau = 1.5$	19%	0,2838
Gumbel Softmax, $\tau = 2.0$	21%	0,2822
Gumbel Softmax, $\tau = 2.5$	20%	0,2825

Table 6.5: Reconstruction loss and accuracy of predicting labels d_i from $e^S_{i,j}$ and $e^{GS}_{i,j}$ embeddings, when $StylePixelCNN^E$ reconstructor was initialized with pre-trained parameters.

vectors. However, we may wonder if it would abandon the label features, if they were already there.

To get the answer, a slightly modified version of the $StylePixelCNN_8$ reconstructor was trained. Instead of passing a one-hot vector y as a conditioning, $Softmax(y)$ was used. As the Sotmax function computes:

$$Softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } z = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

for a 10 dimensional one hot vector y the denominator equals $e+9$, and the function's value is:

$$Softmax(y)_i = \begin{cases} \frac{1}{e+9} & \text{if } y[i] = 0 \\ \frac{e}{e+9} & \text{if } y[i] = 1 \end{cases}$$

where $y[i]$ is a value of the vector at i-th coordinate. Figure 6.4 presents the final form of such vector.

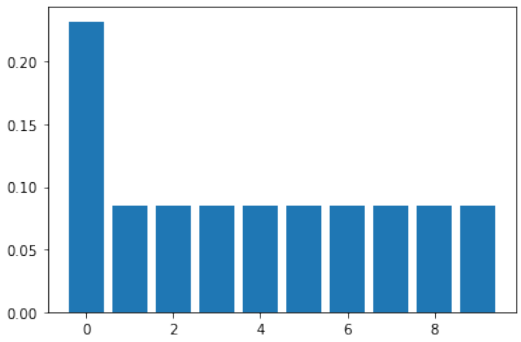


Figure 6.4: Values of one-hot vector with Softmax function applied.

The $StylePixelCNN_8$ conditioned on such vectors perform as good as the previously computed model both in reconstruction task and in modeling the label probability $p(y|x^i, s_i)$. Having this reconstructor, we can initialize the E weights tensor to

represent a one-hot vectors of following N samples. Embedding vector of x_i sample would be then $e_i = Softmax(E[i]) = Softmax(y_i)$, therefore it is exactly the same vector as the one used during $StylePixelCNN_8$ training.

In this approach, switching from $StylePixelCNN_8$ using the fixed conditioning vectors to the $StylePixelCNN_8^E$ is just enabling the reconstructor to modify conditioning vectors at the late stage of training. After the learning process, an embedding classifier converged quickly and reached 99,99% accuracy in mapping e_i vector to the label of x_i image (table 6.6 presents results of experiments from this section). Figure 6.5 presents heatmaps of few e_i vectors grouped by the appropriate digit label. The "spike" at the proper dimension is easily visible, $StylePixelCNN_8^E$ did not remove the label information from the embedding.

It is worth noting that the reconstruction error computed for $StylePixelCNN_8^E$ is significantly lower than this value for both $StylePixelCNN_8$ and all the previously trained $StylePixelCNN^E$ reconstructors that were using a 10-dimensional embedding vector to reconstruct the image. It may suggest, that the label information is a really valuable feature for the decoder, however it is not able to model it in the embedding.

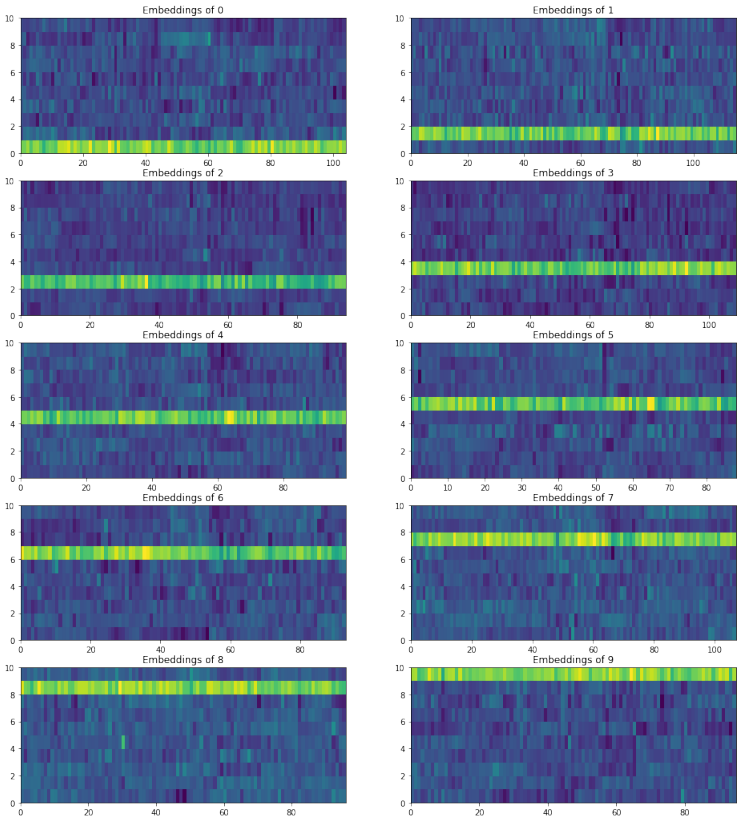


Figure 6.5: Embedding vectors e_i grouped by y_i - the label of respective image, when E weights were initialized as one-hot vectors.

Metric	Score
reconstruction loss (nats/pix) with Softmaxed one-hot conditioning	0,2821
reconstruction loss (nats/pix) with learned Softmaxed embeddings initialized as one-hot vectors	0,2640
Embeddings e_i classification	99,99%
results from table 6.1:	
StylePixelCNN conditioned on one-hot vector reconstruction loss (nats/pix)	0.2811
reconstruction loss (nats/pix) when conditioned on $e^S \in \mathbb{R}^{10}$ without embeddings initialization	0.2832

Table 6.6: Results of embedding classification and reconstruction for models using both fixed value and the learned embedding as conditioning, when the E is properly initialized. Bottom rows present comparable results from table 6.1.

6.4 Using neural network to produce the embeddings

The last experiment showed, that the label information is a valuable feature for the $StylePixelCNN^E$ reconstructor, but it is not capable of optimizing the embedding weights E to represent these labels. We could however replace the E tensor that provides the embedding vectors with a whole neural network, that would be tuned by the reconstructor's gradient. This is equivalent with constructing an autoencoder with additional style model S.

We are going to use an architecture of a neural network performing well in QMNIST data classification task. Such models tend to produce a similar output for similar input values - images presenting the same digit should be assigned a score vectors with maximum in the same position. Such a model could produce embedding vectors that contain some label ralated featuers then.

A reconstructor in such setup is composed of 3 models - a style model S, a neural "embedding model" E^N and the decoding part R (PixelCNN in our case). It models x_i image probability:

$$p(x_i|E^N(x_i), \overline{s_i})$$

where $\overline{s_i}$ is again a style vector representing the a_j - author of x_i. The reconstructor is minimizing a loss function:

$$L(x_i, R(E(x_i; \Theta_E), S(x_0^j, ..., x_k^j; \Theta_S); \Theta_R))$$

Therefore the parameters Θ_E and Θ_S of embedding and style models are tuned only by the reconstructor's gradient.

An architecture used as style model extended with a linear layer mapping the output to a 10 dimensional vector and applying the Softmax function could be used

as QMNIST samples classifier. After a short training it achieved 98,22% accuracy, therefore it may be considered as a good candidate for E^N architecture. For the upcoming computations we used the $StylePixelCNN_8^{E^N}$ reconstructor, composed of a PixelCNN, a style model S_8 taking 8 images to produce a style vector and an embedding model with the Softmax function E^N.

e_i^N dimensionality	Reconstruction loss (nats/pix)	e_i^N classification accuracy
10	0,2538	91,9%
3	0,2634	82%

Table 6.7: Results of $StylePixelCNN_8^{E^N}$ reconstructor performance. Using an embedding model was beneficial for reconstruction task and e_i^N vectors were easy to classify.

Results presented in table 6.7 show that $StylePixelCNN_8^{E^N}$ trained in the described approach achieved the best reconstruction score from all already developed models that use 10 dimensional vectors to reproduce a QMNIST sample. This should come as no surprise, as the model is an example of autoencoder, and the E^N enables to capture some valuable features.

Figure 6.6 presents few samples generated by $StylePixelCNN_8^{E^N}$ (top row) given embedding vectors e_i^N computed for corresponding QMNIST samples (bottom row). The reconstructor was able to produce some quality samples, where digits are easy to distinguish.

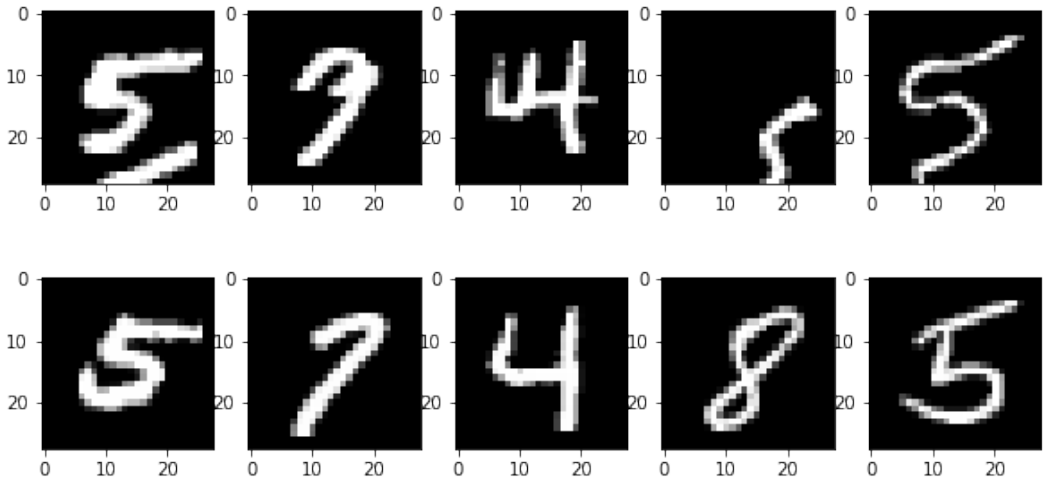


Figure 6.6: **Top:** Samples generated by $StylePixelCNN_8^{E^N}$. **Bottom:** QMNIST test set images used to compute e_i^N vectors used to generate corresponding samples.

In previously developed embedding models, we were able to operate only on embedding vectors assigned to the QMNIST train set, as only for these samples appropriate entries in the E weights tensor were available. With E^N model however

we can evaluate the embeddings also for samples from the test set.

Once again a simple embedding classifier was employed to model $p(d|e_i^N)$ and predict label from the embedding vector. It achieved 91,9% accuracy on the QMNSIT test set. Compared to the accuracy, when embedding was computed as $e_i = Softmax(E[i])$ it's a huge improvement.

If we modify the last, linear layer in E^N to produce a 3 dimensional vector instead of 10 dimensional one we can visualize the obtained embeddings. Table 6.7 shows, that the smaller version of e_i^N vectors were still well classified and provide some valuable features for reconstruction. Figure 6.7 presents the visualization of the $e_i^N \in \mathbb{R}^3$ colored by the represented digit. At the first glance, they are not easy to distinguish. Also they are spread over just 2 dimensions. It may be caused by the Softmax function, that constrains the e_i^N values.

E^N was optimized by the reconstructor to produce vectors that facilitate image reproduction. It turned out that these latent vectors are very easy to divide into clusters representing digit labels. Once again we can see that using vectors that provide the label information is beneficial for PixelCNN reconstruction.

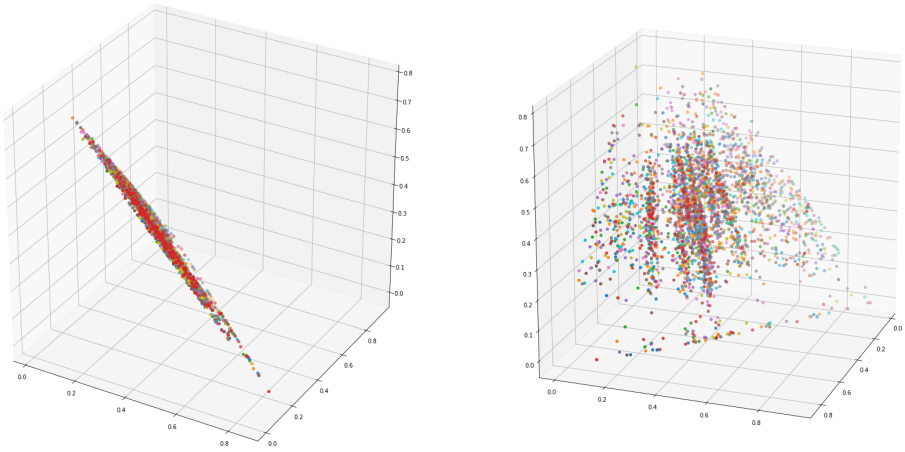


Figure 6.7: e_i^N embedding vectors when E^N maps its input to $\mathbb{R}^3$. Each color represents individual digit label.

Chapter 7

Conclusion

During experiments conducted within this thesis we evaluated a reconstructor in isolation from the encoder, to get a better understanding of its performance. We used the Bayes' theorem to develop an approach enabling us to compare different reconstructor models, by treating them as transcript classifiers.

In the next part of the research we introduced a style model that improves the reconstructor's performance. This simple, additional component actually captures an author's writing style, what was proved by replacing a style vector of the true author with a vector representing a "fake" writer. We have also seen that the selection of samples passed to the style model matters, therefore one should pay attention to this aspect, when using a style model.

The last section of the thesis was devoted to replacing a transcript with learned embeddings. When a reconstructor was able to choose the information stored in the conditioning vectors, the label-related features were not captured. However, when we guide a reconstruction to include such information in these vectors, it was beneficial for the model. The reconstructor was therefore not capable of guiding the embeddings well.

At the very end I would like to thank my tutor dr hab. Jan Chorowski for his patient and commitment during the research. His support and effort put in this thesis was crucial for its success. Also all the machine learning knowledge that was passed to me over my studies is an invaluable advantage.

Bibliography

[1] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018.

[2] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic image synthesis with spatially-adaptive normalization. *CoRR*, abs/1903.07291, 2019.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.

[4] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[5] Diederik P. Kingma and M. Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2014.

[6] Jan Chorowski et al. The "ScribbleLens" Dutch historical handwriting corpus. In *International Conference on Frontiers of Handwriting Recognition (ICFHR)*, page To Appear, 2020. http://www.openslr.org/84/.

[7] Jan Chorowski et al. Unsupervised Neural Segmentation and Clustering for Unit Discovery in Sequential Data. In *NeurIPS 2019 workshop - Perception as generative reasoning - Structure, Causality, Probability*, Vancouver, Canada, December 2019.

[8] Pytorch website. `https://pytorch.org`.

[9] Aaron van den Oord, Nal Kalchbrenner, Lasse Espeholt, koray kavukcuoglu, Oriol Vinyals, and Alex Graves. Conditional image generation with pixelcnn decoders. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett,

editors, *Advances in Neural Information Processing Systems 29*, pages 4790–4798. Curran Associates, Inc., 2016.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *CoRR*, abs/1601.06759, 2016.

[12] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[13] Chhavi Yadav and Léon Bottou. Cold case: The lost mnist digits. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019.

[14] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[17] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. 2017.