technische universität
dortmund

Bachelor Thesis

Sequence synthesis with Conditional Generative Adversarial Recurrent Neural Networks

Alexander Becker

March 7, 2018

Department of Computer Science
Chair of Artificial Intelligence (LS8)
TU Dortmund
`http://www-ai.cs.uni-dortmund.de`

Contents

1 Introduction **3**

2 Preliminaries **3**
2.1 Artificial Neural Networks . . . 3
2.2 Long Short-Term Memory . . . 7
2.3 Generative Adversarial Networks . . . 10
2.4 Conditional Generative Adversarial Networks . . . 12
2.5 Conditional Generative Adversarial Recurrent Neural Networks . . . 12

3 Experiments **13**
3.1 MNIST . . . 14
3.2 Football Events Data . . . 26
3.2.1 Data set . . . 27
3.2.2 Preprocessing . . . 28
3.2.3 Experiment I . . . 30
3.2.4 Experiment II . . . 41
3.2.5 Experiment III . . . 43
3.2.6 Experiment IV . . . 44
3.2.7 Experiment V . . . 46

4 Conclusion **48**

Bibliography **51**

1 Introduction

The task of sequence generation is relevant in many fields of application, e.g. text and speech generation, music compositions, machine regulations over time and prediction of protein structures. At the same time it is often important to depend on a certain context in addition to the preceding sequence such as a certain information that should be phrased, the scale of a song or a machine's target temperature. The most common and successful model used for sequence generation tasks is the Long Short-Term Memorie (LSTM). However, LSTMs are limited in the sequences they can generate since the training process always depends on a certain sequence that was drawn from the original data. This means that if we use the LSTM to generate a sequence with respect to a yet unknown context vector without having an example sequence it will not be able to do so. For this purpose we introduce the Conditional Generative Adversarial Recurrent Neural Networks (CGARNNs) that combines the advantages of LSTMs and Generative Adversarial Networks (GANs). Since the CGARNN's generative part only depends on the feedback of the discriminative part, we expect it to learn how to generate sequences with respect to valid context vectors that are not included in the training data set but are element of the set from which the context vectors are drawn. In section 3 we will investigate the behavior regarding the above-mentioned property in multiple experiments. We start with a proof of concept on the MNIST data set and continue with a text corpus containing descriptions of football game situations. Beforehand we explain the necessary preliminaries in section 2. In section 4 we will draw a conclusion and describe how to continue the work on the CGARNN model in the future.

2 Preliminaries

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a well-known and widely used model in machine learning, common in nearly every field of application and occurring in many different variations while the basic mechanisms stay similar. Basically, an ANN is a concatenation of biologically inspired neurons (Fig. 1) that take vectors x as input and calculate output values regarding neuron specific weight matrices W and biases b. Therefore the neurons' output scalar y will be calculated by (Eq. 1). The weights

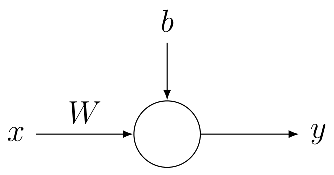


Figure 1: Neuron with input x, weight matrix W, bias b and output y.

are necessary to incorporate the different significance of the inputs and form the neurons' "knowledge".

$$y = xW + b \qquad (1)$$

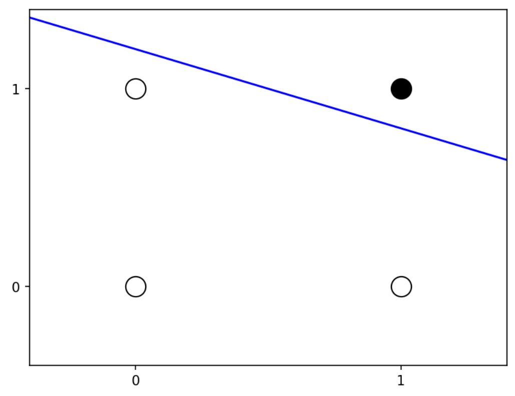


Figure 2: The blue line describes the linear function represented by the neuron. The black point is the value for which the neuron returns 1 ($\geq \theta$), the white points are the values for which the neuron returns 0 ($< \theta$). Thus this neuron represents the logical *and*.

When we think of biological neurons, there is one important aspect we have not considered yet: the impulse propagation. A biological neuron receives an impulse and passes it if it was strong enough, which allows a more complex behavior than simply passing all impulses no matter how strong they are to the next neurons. The simplest way to achieve this behavior is by adding a threshold θ to every neuron. The neuron will pass the "impulse" when the output is greater or equal to the threshold, otherwise it will not (Eq. 2). Comparing the output to the threshold results in either 1 or 0 depending on whether the equation is true or false. This result will then be used as the neuron's actual output.

$$\begin{aligned} & xW + b \geq \theta \\ \Leftrightarrow \quad & xW + b - \theta \geq 0 \end{aligned} \tag{2}$$

Even if adding a threshold to each neuron expresses the above-mentioned behavior, we are only able to describe linear functions because every neuron that can be described by equation 2 represents a line that divides the plane into two parts (Fig. 2). Unfortunately, most problems are very complex and therefore cannot be represented by a linear function such as the *xor* function (Fig. 3). To describe non-linearity we use activation functions which serve the same purpose as the comparison to a threshold does - the regulation of the propagated impulses. The difference between those two approaches is that the comparison to a threshold always results in 0 or 1 and is not continuously, whereas most activation functions map into the interval of [0,1] and are continuous. Figure 4 shows a graphical representation of the threshold comparison and one of the most basic activation functions: the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

Usually, networks contain neurons arranged in multiple layers (see Fig. 5). We differentiate between the input layer that takes the network's input, any number of hidden layers that are responsible for the main processing and the output layer that

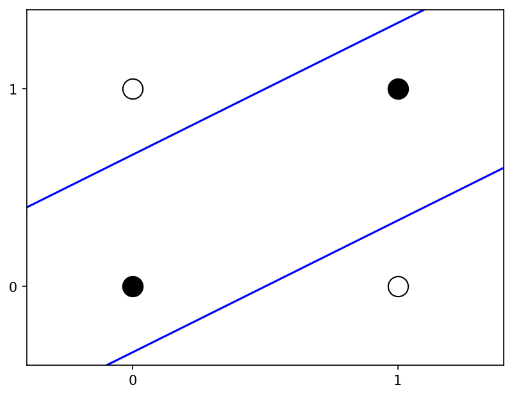
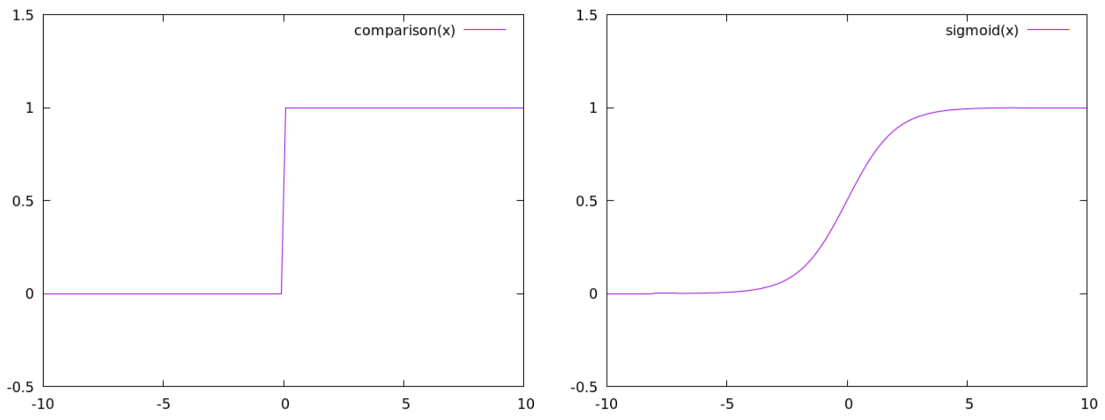


Figure 3: *xor* function. The black and the white points cannot be separated by a single line and therefore the *xor* function cannot be represented by a neuron as described above.



(a) comparison **(b)** sigmoid

Figure 4: Threshold comparison vs. sigmoid

provides the network's output. In order to propagate the impulses from the neurons in layer $l-1$ to all the neurons in layer l, we provide the $l-1$-th layer's output as the input for layer l by combining the outputs from all the neurons to a single vector. Equation 4 describes the computation of a layer's output element-wise and recursively. In case we reached the first layer in our recursion (base case), we will not be able to access a previous layer's output, because there is none, so we will use the network's input instead. Each column of $W^{(l)}$ represents a weight vector for each neuron in layer l, whereas the vector $b^{(l)}$ contains the bias values. f denotes the activation function. Consider that it is possible to use a different activation function for each neuron and that equation 4 is just a simplified representation.

$$y_i^{(l)} = \begin{cases} f(xW_i^{(0)} + b_i^{(0)}), & l = 0 \\ f(y^{(l-1)}W_i^{(l)} + b_i^{(l)}), & \text{otherwise} \end{cases} \tag{4}$$

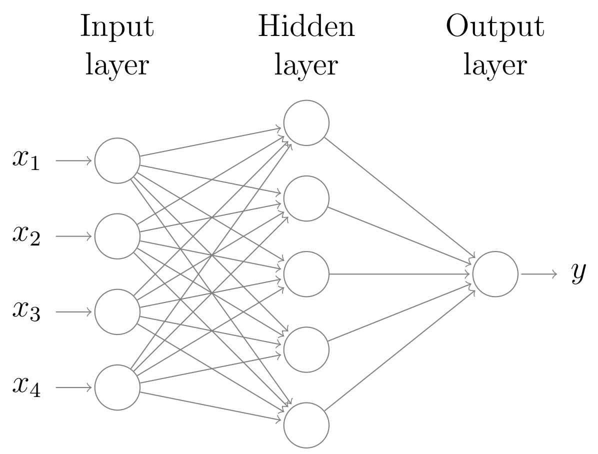


Figure 5: Artificial Neural Network with multiple layers

Now that we know how to calculate the output of an ANN, we can focus on the learning process. Therefore we will look at the most common learning approach, backpropagation, as described by Rumelhart, Hinton and Williams in 1985 [18]. When starting the training of an ANN for a specific problem, its outputs will be rather random. The reason for this is that the network's weights were not adjusted to process the input data to the desired output. As mentioned above, the network's weights represent its actual knowledge. So in order to make the network learn how to solve the specific problem, we need to adjust the weights in a way that the network's "knowledge" is increased. Instead of defining the network's "knowledge", we rather define its error, since a smaller error can be achieved by weights that are adjusted properly. The network's error is always determined as a function of a processed input. Depending on the specified problem, the error is defined by different loss functions. Equation 5 shows a basic example of a loss function - the mean squared error

$$\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - f(x_i))^2 \tag{5}$$

where N is the number of training samples, $\hat{y}_i$ the label and $f(x_i)$ the output for the i-th sample. $\hat{y}_i - f(x_i)$ describes the difference between the label and the actual output. In order to receive positive error values only, the difference is squared. Then the mean of the errors for all inputs in the batch is calculated. The result describes the network's error for a batch of inputs. Next, we need to use the information about the network's error to adjust its weights. This procedure is rather complex due to the many connections between the neurons and the missing information about the influence of a single neuron on the resulted error. We therefore need a function that assigns blame to a single neuron depending on the network's error. This function is known as the delta rule (Eq. 6).

$$\delta_i^{(l)} = \begin{cases} f'(net_i)\lambda'(\hat{y}_i, y_i), & \text{output layer} \\ f'(net_i)\sum_j \delta_j^{(l+1)} w_{ij}, & \text{otherwise} \end{cases} \tag{6}$$

$f'(net_i)$ denotes the derivative of the activation function, $\lambda'(\hat{y}_i, y_i)$ denotes the derivative of the loss function, net_i is the non-activated output of the i-th neuron, $\hat{y}_i$ is the label's i-th value, y_i is the output of the i-th neuron and w_{ij} the weight between neuron i in the current layer and neuron j in the next layer. For the output layer, the error is determined by the multiplication of the activation function's derivative and the loss function's derivative. These describe the direction towards the minimum error and the degree of the error and if we should go further in the direction that is indicated by the activation derivative or if we should go in the opposite direction in order too reach the minimum. For all remaining layers we multiply the activation derivative with the sum of all errors of the neurons in the next layer that are connected to the current neuron. Thereby the sum describes the influence of the current neuron's output on the error of the following neurons. By means of the delta rule we are actually able to define the weight updates for all neurons, even those that are not part of the output layer. The weight updates are defined by

$$\Delta w_{ij} = \alpha \delta_j y_i \tag{7}$$

where Δw_{ij} is the difference of the weight between neuron i in the current layer and neuron j in the next layer, α is the learning rate that describes the size of the weight's difference before and after updating and y_i is the i-th neuron's activation. The backpropagation algorithm then works like this: First, a certain number of inputs are given to the network one after another. For each input the loss and error values are calculated using the loss function and the delta rule. Then the weights are updated by using the errors and the neurons' activations. In order to obtain a network with an error near 0 these steps are repeated for several iterations.

2.2 Long Short-Term Memory

In 1997, Hochreiter and Schmidhuber introduced a novel approach for handling long term dependencies within the input data and therefore provided a model that is able to process sequential data with regard to earlier sequence elements called Long Short-Term Memory [12]. The family of networks whose outputs depend on some earlier calculations represented by additional time-delayed inputs for the neurons of the hidden layers are called Recurrent Neural Networks (RNNs). To explain the properties of RNNs, we will first look at Elman nets. Then we will discuss possible problems and explain Long Short-Term Memories (LSTMs) which solve these problems.

Elman nets [5] were described by Elman in 1990 and are one of the most basic forms of RNNs. The problem Elman wanted to solve with his recurrent nets is the representation of time which is a natural property of sequences since they can be seen as data that is divided into n time steps where n is the number of sequence elements. The idea of Elman nets is to represent time implicitly as the effect it has on the data processing by slightly modifying vanilla ANNs (section 2.1). The neurons of the hidden layer now have two different kinds of inputs (Fig. 6). The first and already known kind of input is the output that comes from neurons of the previous layer. The second input is a time-delayed input - the neuron's output at the last time step - that is represented by an additional context neuron for each neuron

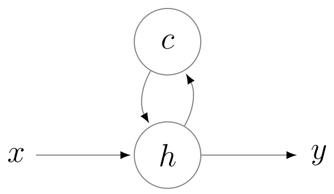


Figure 6: Elman net with one hidden layer. The neuron of the hidden layer receives an additional input from the context neuron that saves the hidden neurons output from the last time step. x and y denote the input and output values, h the hidden neuron and c its correpsonding context neuron.

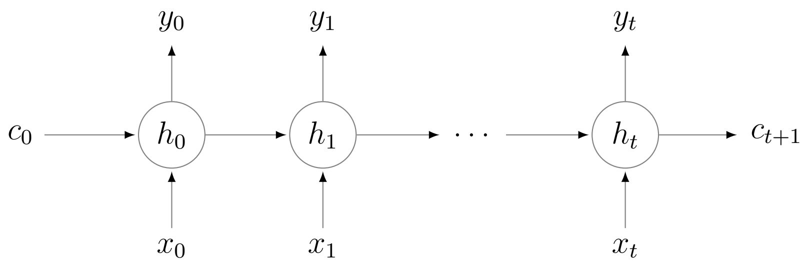


Figure 7: Unrolled Elman net.

in the hidden layers. The context neuron stores the hidden neuron's output and feeds it back in the next time step. Therefore, information about the last processing step is stored and used to make each output of the hidden neuron dependent on the previous sequence element.

This simple modification gives Elman nets the potential of handling sequential data while the processing of each sequence element depends on the previous elements. In order to minimize the loss of an RNN using a gradient-based optimization function, we have to unroll the network first. This means that the recurrent connections are replaced by multiple copies of the same network. The number of copies is equal to the number of elapsed time steps. Therefore, when t time steps of the Elman net in figure 6 are unrolled, we achieve the vanilla ANN seen in figure 7. Unfortunately, there is a grave problem when using Elman nets to process large sequences. Hochreiter and Schmidhuber described this problem in [12]. When calculating the error values through time, the error decreases exponentially with every step we take. This phenomenon is called the vanishing gradients problem. The error of a nonoutput neuron j at time step t can be described as

$$\delta_j(t) = f_j'(net_j(t)) \sum_i w_{ij}\delta_i(t+1) \tag{8}$$

where $f_j'(net_j(t))$ is the derivative of the activation of neuron j at time step t, $net_j(t)$ its non-activated output, w_{ij} the weights to all neurons of the next time step $t+1$ and $\delta_i(t+1)$ the error of neuron i at time step $t+1$. Thus the sum represents the influence of neuron j on the error values of time step $t+1$ at time step t. The i-th neuron's error results from multiplying this error influence with the neuron's activation derivative that determines in which direction the weights must

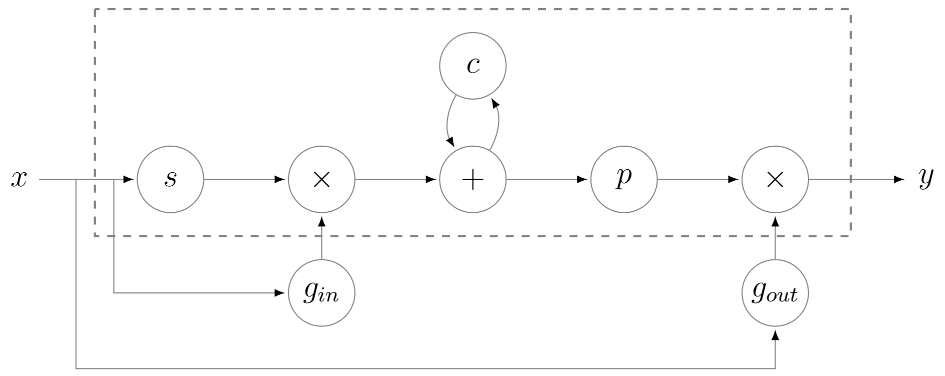


Figure 8: LSTM memory cell and gates. x and y denote the input and output, g_{in} and g_{out} the input and output gates and everything inside the dashed rectangle shows the actual LSTM memory cell. Inside the memory cell the output of s describes the information that wants to be stored in the memory, c describes the context neuron, p's output describes the information that wants to be propagated and the nodes with $\times$ and $+$ describe vector multiplication and addition.

be adjusted. From this follows the update rule

$$w_{ji} = w_{ji} - \alpha\delta_j(t)y^i(t-1) \tag{9}$$

where α is the learning rate and $y^i(t-1)$ is the activation of neuron i at time step $t-1$. The total error flow from an arbitrary neuron $l_0 = u$ at time step t to another arbitrary neuron $l_q = v$ at time step $t-q$ is defined as

$$\sum_{l_1=1}^{n} \cdots \sum_{l_{q-1=1}}^{n} \prod_{m=1}^{q} f'_{l_m}(net_{l_m}(t-m))w_{l_m l_{m-1}} \tag{10}$$

The sums represent all paths from v to u and the product describes the error flow for each path. In case that all errors on a path are smaller than 1, the total error will vanish due to the multiplication of the single errors which results in exponential error decreasing. Therefore the weight changes per update are insignificant. In order to solve the vanishing gradients problem, Hochreiter and Schmidhuber introduced LSTM networks. The neurons of an LSTM network are called memory cells and are provided with an input and an output gate (Fig. 8). An LSTM memory cell takes an input x, which is either the initial input at time step 0 or the output of the last time step concatenated with the input at time step t, and processes it by a basic neuron s. The result is then the new information that wants to be stored in the cell's memory. The input gate g_{in} also takes x as an input and returns a vector of values from [-2,2]. These values describe how much of the new information should be stored and which parts of the stored information should be forgotten. Thus these two vectors will be multiplied and the result is added to the current memory. The resulting memory will be processed by another basic neuron p. Its result describes the information that wants to be propagated to the next time step. The output gate g_{out} then decides how much of the memory should be propagated. Therefore, it also takes the input x and returns a vector with values from [-1,1]. The multiplication of

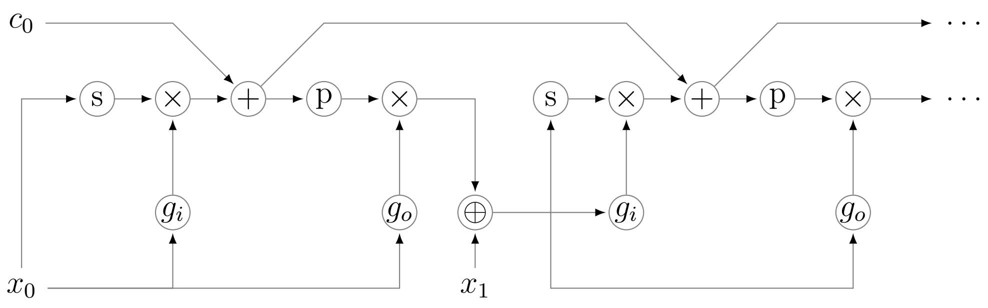


Figure 9: Unrolled LSTM memory cell. c_0 describes the initial memory state and $\oplus$ is the vector concatenation. Each repetition of the LSTM cell represents one time step.

the gate's result and the information that wants to be propagated is then the actual output of the memory cell. Unrolling the LSTM network leads to a representation that reveals how the concept of the memory cell solves the problem of vanishing gradients (Fig. 9). To avoid the vanishing gradients problem, we want to have a constant error flow for the recurrent connections. In other words, we want the following equation to be true.

$$\begin{aligned} & \delta_i(t) &&= \delta_i(t+1) \\ \Leftrightarrow\quad & f_i'(net_i(t))w_{ii}\delta_i(t+1) &&= \delta_i(t+1) \\ \Leftrightarrow\quad & f_i'(net_i(t))w_{ii} &&= 1 \end{aligned}$$

In order to satisfy the above equation, a proper activation function f must be chosen. Integrating this equation leads to the result that the activation function must be linear.

$$\begin{aligned} & \int f_i'(net_i(t))w_{ii} &&= \int 1 \\ \Leftrightarrow\quad & f_i(net_i(t))w_{ii} &&= net_i(t) \\ \Leftrightarrow\quad & f_i(net_i(t)) &&= \frac{net_i(t)}{w_{ii}} \end{aligned}$$

This means that we can achieve a constant error flow by choosing a linear activation function for the connection between the hidden and the context neurons. This is why the identity function is used in the LSTM memory cells. In summary, LSTMs are able to process sequences and learn long time dependencies because they are not affected by the vanishing gradient problem due to their linear memory.

2.3 Generative Adversarial Networks

In 2014, Goodfellow et al. introduced a generative adversarial framework [10] that is capable of generating high quality samples and at the same time uses exact gradients for optimizing the network parameters.

The generative adversarial framework consists of two parts. First, the generative part that is trained to generate samples similar to the data of a given distribution. In other words, the generator represents a probability distribution and tries to approximate the distribution of the real data. The second part is the discriminator that is trained to distinguish between samples provided by the generator and samples from the real data distribution. Therefore, both the generator and the discriminator compete against each other. The generator's goal is to outsmart the discriminator such that all the generated samples will be classified as real data. On the other hand, the discriminator wants to distinguish both probability distributions perfectly such that it always relates a sample to the correct distribution. The exact models for the generator and the discriminator are not fixed such that all kinds of generative and classifying models can be used in this framework. Goodfellow et al. recommended using artificial neural networks for both the generator and the discriminator because it makes the implementation and the learning process straightforward. This is why the framework is called Generative Adversarial Networks (GAN)(Fig. 10). The competition between the generative and the discriminative part of the framework can be described by the following minimax game:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{11}$$

where G and D are the generator and the discriminator, x is a sample drawn from the real data distribution p_{data} and z is a noise sample drawn from a noise distribution p_z. $\log D(x)$ describes the discriminator's accuracy on classifying data from the real distribution whereas $\log(1 - D(G(z)))$ describes the accuracy on classifying data provided by the generator. The discriminator wants to maximize its accuracy by relating as much data as possible to the correct data distribution. Thereby the maximum accuracy is 0. The generator wants to minimize the same accuracy the discriminator wants to maximize. Because of the missing influence of the generator on the samples drawn from the real distribution, its goal is just minimizing $\log(1 - D(G(z)))$. Goodfellow et al. noticed that during training the minimax game (Eq. 11) does not provide good gradients for the generator, especially in the first training iterations. Therefore, the objective of G was changed from minimizing $\log(1 - D(G(z)))$ to maximizing $\log D(G(z))$ which is practically the same but leads to stronger gradients.

Having specified the objectives of G and D, the training works like this: In each training iteration we first calculate the gradients of the discriminator for k mini-batches of real and generated data. We then calculate the generator's gradients for a single mini-batch of noise samples. Using a gradient-based optimization method, we then update the network weights. The reason for using k mini-batches for the discriminator is that we want to have a good classifying model in order to provide good gradients for the generator. Training the discriminator in advance is out of the question because this would lead to an overfitted classifier. So, using k mini-batches for the discriminator per mini-batch for the generator is a trade-off between a non-overfitting classifier and a classifier that provides good gradients.

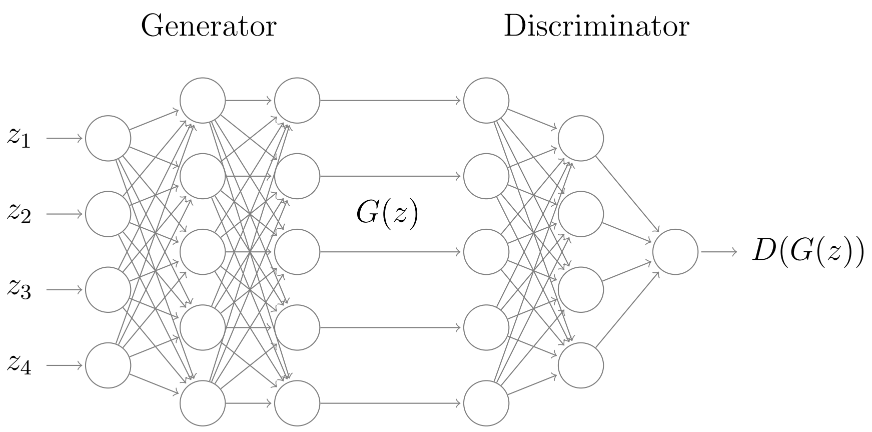


Figure 10: Generative Adversarial Network with generator G and discriminator D

2.4 Conditional Generative Adversarial Networks

In 2014, Mirza and Osindero introduced an extension to Goodfellow's Generative Adversarial Networks, namely the Conditional Generative Adversarial Networks (CGAN) [17]. The goal of this modification is the generation of samples depending on a certain condition y, e.g. a class label or incomplete data that will be completed during the generation process. Since the discriminator must verify if the input depends on the condition, both the generator and the discriminator will use y as an extra input during training. Thus the minimax game (Eq 11) described in section 2.3 will be modified such that it includes the condition input y.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x \mid y)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z \mid y)))] \quad (12)$$

The way how y is actually processed by the networks is not predetermined. In [17] Mirza and Osindero just concatenated the input and the condition to a single input vector. Therefore, we can use the learning algorithm as described by Goodfellow et al. [10] without further changes.

2.5 Conditional Generative Adversarial Recurrent Neural Networks

In this thesis, we aim to investigate the CGAN framework's ability to generate sequential data. Since the generator and the discriminator can be chosen arbitrarily as long as the generator is a generative and the discriminator is a classifying model, we will use LSTMs for both. As mentioned in section 2.2, LSTMs are a common model for processing sequential data and were often used with huge success. We refer to this model as Conditional Generative Adversarial Recurrent Neural Networks (CGARNNs). We expect CGARNNs to combine the abilities of generating context-sensitive high quality samples while learning and utilizing the distribution of the sequential data. Other than LSTMs, CGARNNs might give us the possibility to generate data described by context vectors that were not available during training because the LSTM's loss always depends on a specific sample sequence per training step while the CGARNN's loss only depends on the discriminator's feedback. We expect the model to interpolate the behavior for unknown context vectors since a context vector contains multiple random variables and therefore parts of the unknown vectors were processed during training before. This is particularly beneficial if the number of possible context vectors is very large and it is impossible to provide training data for all of them. In the following section we will explore the potential of the CGARNN model empirically in multiple experiments.

3 Experiments

In this section we describe the Conditional Generative Adversarial Recurrent Neural Network (CGARNN) experiments which are divided into two sections.

The first section is a proof-of-concept to show that the model is able to generate data depending on a given context. Therefore we use the MNIST data set [14] and compare our results to those of three other models (Tab. 1): a Generative Adversarial Network (GAN), a Conditional GAN (CGAN) and a Generative Adversarial Recurrent Neural Network (GARNN). We chose those three rather similar models in order to determine the advantages of the single properties of the CGARNN model, namely the use of Recurrent Neural Networks (RNNs) to achieve some kind of memory unit and the dependency on a context vector. The experiments should show that both the use of RNNs and the additional context input improve the quality of the generated data.

	no memory	memory
no context	GAN	GARNN
context	CGAN	CGARNN

Table 1: MNIST - Comparison of model properties

The second section describes the experiments on the Football Events data set (REF) with which we want to discover the advantages of the CGARNN model for the text generation task. For comparison reasons we train a Long Short-Term Memory (LSTM) model that also depends on a context vector in addition. So the main difference between these two models is the way the feedback is given to them. While the CGARNN's generator gets it's feedback from the discriminator, the LSTM model needs an original text from the data set in order to compare it to the generated one. We further discuss the impact of the way the feedback is given to the different models in 3.2.

We implemented all experiments in Python using Google's TensorFlow[1] library for machine learning, published the code under an open source license[2] and wrote a code documentation (see appendix).

[1] `https://www.tensorflow.org/`

[2] `https://bitbucket.org/ROYALBEFF/conditional_generative_adversarial_rnns` (also listed in the appendix)

3.1 MNIST

The first data set we used for our experiments is the well-known MNIST data set [14] consisting of 70000 28×28 images showing handwritten digits. The experiments on the MNIST data set are a proof-of-concept, whereas the experiments in 3.2 show the model's capabilities. As mentioned above, we trained three models in addition to the CGARNN on the MNIST data in order to compare their results. The three models are: a GAN that is independent of the context and does not make use of RNNs, a CGAN that depends on the context and does not make use of RNNs either, and a GARNN that is independent of the context but uses RNNs instead of common Artificial Neural Networks (ANNs) (Tab 1).

	Generator	Discriminator
N (per layer)	noise size, 392, 784	784, 392, 1
loss	Sigmoid cross entropy with logits	
optimizer	ADAM	
k	-	1
batch size	64	
noise size	100	-
α	0.001	
initializer	Glorot normal distribution	
epochs	50000	

Table 2: GANs network settings. N describes the number of neurons per layer (input, hidden, output) for generator and discriminator. The second and third line are the loss and optimizer functions. k is the number of the discriminator's training steps per each training step of the generator. Batch size is the number of training examples per epoch. Noise size describes the size of the generator's input vector that is then formed to a 28×28 image. α is the learning rate. The initializer describes the way the variables are initialized. Epochs is the number of training epochs.

First we trained a GAN with parameters as seen in Tab. 2. In this model both the generator and the discriminator are artificial neural networks sharing most of their network parameters.

The generator consists of three layers: input layer (noise size neurons), hidden layer (392 neurons) and output layer (784 neurons). The number of neurons per layer depends on the noise size which describes the size of the input vector, and on the size of the MNIST data. The input layer contains one neuron per entry in the input vector. The hidden layer contains exactly half of the neurons the output layer contains. This way the size of the output vectors across the layers increases regularly, such that the network's output does not depend on a specific part of the network too much. The output layer then contains exactly 784 (28×28) neurons in order to produce an output vector similar to those in the MNIST data set.

The discriminator also consists of three layers: input layer (784 neurons), hidden layer (392 neurons) and output layer (1 neuron). The number of neurons per layer is also decreasing regularly, such that the discriminator profits from this the same way the generator does. The discriminator's job is to distinguish real MNIST images from those that are produced by the generator. The single scalar output describes the probability that the input vector came from the MNIST data set rather than having been generated. k describes the number of training steps of the discriminator per training step of the generator. We set $k = 1$, which is a very common choice.

As mentioned in chapter 2.3, the loss function of the generator and the discriminator can be described as a minimax game. During our experiments we often observed the discriminator's loss being NaN (not a number). A quick look at the discriminator's loss function (Eq. 13) reveals the reason for this behavior. The problem occurs when the discriminator is so good at classifying the input vectors that its loss value is really close to 0. Rounding errors then lead to a loss value of 0. At this point we try to calculate the logarithm of 0, which is not defined.

$$L_D = -(\log D(x) + \log(1 - D(G(z)))) \tag{13}$$

The first idea that came to mind was adding a small value $\epsilon = 0.001$ to the network's output in order to prevent the loss value from becoming too small (Eq. 14). Though even while this fixed the problem and results in better loss progression and generated samples, it did not feel satisfactory.

$$L_D = -(logD(x + \epsilon) + \log(((1 - D(G(z)) + \epsilon))) \tag{14}$$

Another idea was to try a different and established loss function that does not have the problem described above: the concept of cross entropy. We noticed some similarities between these two loss functions and while having a closer look at them we realized that they are exactly the same. This means we can still use the minimax game described in the paper by just rearranging the formula and without adding the aforementioned value ϵ. In the following we explain cross entropy and then prove that it is the same as the minimax game.

Cross entropy (Eq. 16) can be used as a measure of similarity between two probability distributions. Before we can understand what cross entropy is, we must have an understanding of how entropy works in general. Entropy measures the average information content of a random variable X over a discrete probability distribution p and is described by Eq. 15.

$$H = \sum_{i=1}^{n} p_i \log \frac{1}{p_i} \tag{15}$$

The number of possible values X can obtain is n. p_i is the probability for $X = i$ and $log\frac{1}{p_i}$ is the information content of X. A very demonstrative way of understanding entropy is the task of assigning each value a bit sequence. The goal is to assign the bit sequence in a way that the expected length of the bit sequence of a random sequence of values is minimized. To achieve this goal we assign values with high probabilities shorter bit sequences than values with smaller probabilities. We can use this property to compare the likeliness of two probability distributions that are defined over the same set of values. Let's say we want to approximate a given probability distribution p over a certain set. Furthermore, we assume that H is the entropy defined over p such that the number of expected bits per sequence is minimal. Now we replace p with our approximated distribution $\hat{p}$ for the assignment of the bit sequences. This results in another entropy with another expected number of bits per sequence. The difference between these two expected number of bits can be used to determine the likeliness of two probability distributions over the same set. This method is called cross entropy (Eq. 16).

The two distributions we compare are the real distribution p that describes our data set whereas the second distribution $\hat{p}$ is the distribution representing the trained model. The goal is to change the network's weights in a way that the represented distribution converges to the real distribution or minimizes the loss, respectively.

$$H = \sum_{i}^{n} p_i \log \frac{1}{\hat{p}_i} \tag{16}$$

A closer look at Eq. 16 will help us understand how it works. First of all we can rearrange Eq. 16, because we know that there are just two classes in total ($n = 2$) and this way obtain Eq. 17. Either the input vector will be classified as real or as fake. We can describe these two outcomes with the probabilities y (probability that the input vector is real) and it's complementary probability $1 - y$ (probability that the input vector is fake) for the real probability distribution and $\hat{y}$ and $1 - \hat{y}$ for the learned distribution respectively with $y \in \{0, 1\}$ and $\hat{y} \in [0, 1]$.

$$\begin{aligned} H &= y \log \frac{1}{\hat{y}} + (1 - y) \log \frac{1}{1 - \hat{y}} \\ &= -(y \log \hat{y} + (1 - y) \log(1 - \hat{y})) \end{aligned} \tag{17}$$

$y \log \hat{y}$ describes the error that occurs in case the input vector came from the real data distribution. $(1 - y) \log(1 - \hat{y})$ describes the error for generated input vectors. When calculating the error for an input vector, only the corresponding part of the equation is used. The other part will be 0 due to the actual label y. If y is 0 then the first part of the equation results in 0, otherwise the second part of then equation results in 0. The error then describes the difference between the optimal entropy given by the real data distribution and the entropy given by the learned distribution. Obviously, when the learned and the real distribution are the same, the entropy of the different outcomes are the same and therefore the cross entropy is 0.

Now we prove that the minimax game and the cross-entropy are the same. The main idea behind this proof is that we want to make the cross entropy independent of the actual output of the network. Therefore we rearrange the cross entropy in a way that it does not take the activated output of the network, but its logits. We can describe the activated output of the discriminator $D(x)$ as $\sigma(\hat{x})$, where $\hat{x}$ is the discriminator's last layer's output, before we apply the activation function to it. This rearrangement works, because the activation function of the output layer is the sigmoid function σ. As we can see in equation 18, we can make use of this property and rearrange the cross entropy such that it uses the logits instead of the activated output and therefore will be defined totally. We call this rearranged variant H_σ sigmoid cross entropy with logits.

$$\begin{aligned}
H &= -(y \log \hat{y} + (1-y)\log(1-\hat{y})) \\
&= y * (-\log \hat{y}) + (1-y) * (-\log(1-\hat{y})) \\
&= y * (-\log \sigma(\hat{x})) + (1-y) * (-\log(1-\sigma(\hat{x}))) \\
&= y * (-\log(\frac{1}{1+\exp(\hat{x})})) + (1-y) * (-\log(1 - \frac{1}{1+\exp(\hat{x})})) \\
&= y * (\log(1) - log(1+\exp(\hat{x}))) + (1-y) * (-\log(\frac{exp(\hat{x})}{1+\exp(\hat{x})})) \\
&= y * (-log(1+\exp(\hat{x}))) + (1-y) * (\log(exp(\hat{x})) - \log(1+\exp(\hat{x}))) \\
&= y * (-log(1+\exp(\hat{x}))) + (1-y) * (\hat{x} - \log(1+\exp(\hat{x}))) \\
&= y * (-log(1+\exp(\hat{x}))) + (1-y) * \hat{x} + (1-y) * (-\log(1+\exp(\hat{x}))) \\
&= (1-y) * \hat{x} + (-\log(1+\exp(\hat{x}))) \\
&= \hat{x} - y * \hat{x} - \log(1+\exp(\hat{x})) \\
&= H_\sigma(y, \hat{x})
\end{aligned} \tag{18}$$

Next, we show that we can express the minimax game as the sigmoid cross entropy with logits. As seen in equation 19, this procedure is very straight forward.

$$\begin{aligned}
L_D &= -(logD(x) + \log(1 - D(G(z))) \\
&= -log(D(x)) + (-\log(1 - D(G(z)))) \\
&= -log(\sigma(\hat{x})) + (-\log(1-\sigma(\hat{z}))) \\
&= 1 * (-log(\sigma(\hat{x}))) + 0 * (-log(1-\sigma(\hat{x}))) + 0 * (-\log(\sigma(\hat{z}))) \\
&\qquad + 1 * (-\log(1-\sigma(\hat{z}))) \\
&= 1 * (-log(\sigma(\hat{x}))) + (1-1) * (-log(1-\sigma(\hat{x}))) + 0 * (-\log(\sigma(\hat{z}))) \\
&\qquad + (1-0) * (-\log(1-\sigma(\hat{z}))) \\
&= y_{\hat{x}} * (-log(\sigma(\hat{x}))) + (1-y_{\hat{x}}) * (-log(1-\sigma(\hat{x})) + y_{\hat{z}} * (-\log(\sigma(\hat{z}))) \\
&\qquad + (1-y_{\hat{z}}) * (-\log(1-\sigma(\hat{z}))), \text{ with } y_{\hat{x}} = 1, y_{\hat{z}} = 0 \\
&= H_\sigma(y_{\hat{x}}, \hat{x}) + H_\sigma(y_{\hat{z}}, \hat{z})
\end{aligned} \tag{19}$$

Now that we have shown that the minimax game describes the same optimization problem as the sigmoid cross entropy with logits, we can use it without running into the problem that the loss value can be NaN.

To minimize the loss value during training we use Adaptive Moment Estimation, an optimization method presented by Kingma and Ba in 2015 [13]. Adaptive Moment Estimation (ADAM) is a stochastic gradient-based optimization method, which means that it optimizes stochastic functions, such as the sigmoid cross entropy with logits, by using partial derivatives of that function. The idea of using gradient descent (ascent) to optimize a loss function is standard practice and used in all established optimization methods in the domain of machine learning. What is special about ADAM is the fact that the learning rate is adaptive in each iteration. The advantage of using an adaptive learning rate rather than a constant learning rate is that we can adjust it to the current learning progression. E.g. when the gradients of the last few iterations indicate a step size that is too large, the learning rate will be decreased for the next iterations in order to converge to the global minimum rather than surpassing it. To determine the learning rate in each iteration we use the first (Eq. 20a) and second moment (Eq. 20b) of the gradients, which are the mean value and the variance, where t is the current iteration, g_t the gradients in iteration t, g_t^2 the element-wise multiplication and β_1 and β_2 are hyper-parameters describing the exponentially decreasing influence of the previous gradients. Kingma and Ba provide default values for the hyper-parameters, which are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t \tag{20a}$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2 \tag{20b}$$

m_0 and v_0 are initialized with 0's which leads to biased moments in later iterations. Fortunately, one can simply correct these biased values by dividing by $(1 - \beta_1^t)$ or $(1 - \beta_2^t)$ respectively, which leads to the moments as seen in (Eq. 21). We reformulated the corrected moments to make the influence of the single gradients easier to see and to make it easier to understand why $(1-\beta_1^t)$ or $(1-\beta_2^t)$ respectively is used to correct the biased moments.

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t} = \frac{\sum_{i=1}^t (\beta_1^{t-i} - \beta_1^{t-i+1}) * g_i}{1 - \beta_1^t} = \frac{(1 - \beta_1) * \sum_{i=1}^t \beta_1^{t-i} * g_i}{1 - \beta_1^t} \tag{21a}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t} = \frac{\sum_{i=1}^t (\beta_2^{t-i} - \beta_2^{t-i+1}) * g_i^2}{1 - \beta_2^t} = \frac{(1 - \beta_2) * \sum_{i=1}^t \beta_2^{t-i} * g_i^2}{1 - \beta_2^t} \tag{21b}$$

The actual update step is then described in Eq.(22), where θ_t are the parameters at the t-th iteration, α is the upper bound of the learning rate (default: $\alpha = 0.001$), $\hat{m_t}$ is the mean value of the last t gradients, $\sqrt{\hat{v_t}}$ is the standard deviation of the last t gradients and ϵ is a small value (default: 10^{-8}) that is added to $\sqrt{\hat{v_t}}$ to avoid division by 0.

$$\theta_t = \theta_{t-1} - \alpha * \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \tag{22}$$

The term $\hat{m_t}/\sqrt{\hat{v_t}}$ is also called signal-to-noise ratio (SNR). One can imagine the learning rate adjusting as follows: When the standard deviation is rather large, then the fraction will lead to a small value and so does the adjusted learning rate. A large standard deviation means that the direction the gradients have to move in, in order to reach the global minimum, is vague. In this case the SNR is rather large and the decreased learning rate prevents the gradient steps from being too large and smaller steps are taken instead. A small standard deviation means that the last few steps reveal a clear direction for the gradients, such that the gradient steps can be increased in order to speed up the optimization process. Nonetheless, the step size is always bounded by α.

The next interesting part is the variable initialization which is typically based on a probability distribution where the initialization values are drawn from. For this we used a Xavier initialization, described by Xavier Glorot and Yoshua Bengio in 2010 [3]. The idea behind Xavier initialization is to solve the problem of too large or too small variances of the weights and therefore of the values that are propagated through a network. In the following we assume a uniform probability distribution with mean 0. When the variance of the weights is too small, then all weights are near the mean value. Calculating a layer's output then results in values that are also near 0. Looking at the sigmoid function for instance, this will give us mostly gradients of about 1, which leads to an almost linear behavior (Fig. 11, blue area). But when the variance of the weights is too large, the output values of a layer are far away from the mean value, too. This results in gradients near 0, which in turn means that the weights will stay as they have been in the previous iterations (Fig. 11, red area). Thus, the goal is to find a variance that we can use to initialize the weights in a way such that the variances of the inputs and outputs of all layers are the same. Looking at a single layer, we can describe the variance of its output y as:

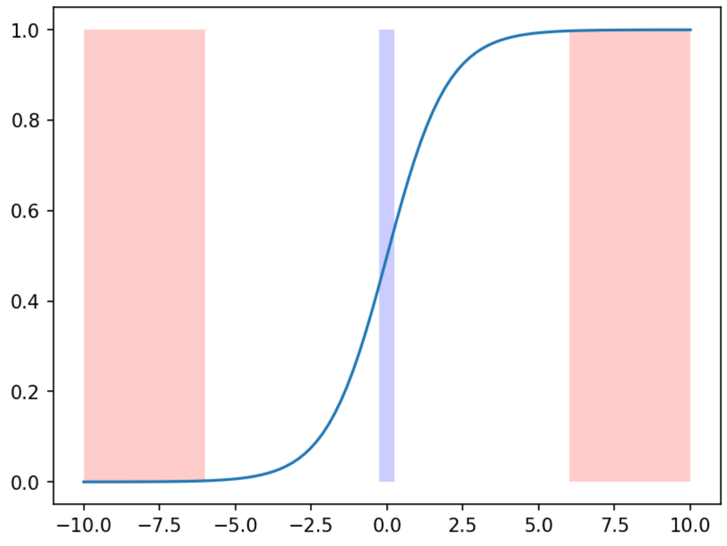


Figure 11: Problem of too small or too large variance of the weights, illustrated by the sigmoid function. A too large variance results in output values that are far away from the mean value and leads to gradients near 0 (red areas). A too small variance results in output values that are very close the mean value and therefore leads to linear behavior (blue area).

$$Var(y) = Var(xW + b) = Var((\sum_{i=1}^{n} w_i x_i) + b) \tag{23}$$

Since b can be seen as another weight that is always multiplied with the input 1, we can drop it to simplify things. The summands are variances of products of independent variables and can therefore be described as:

$$\begin{aligned} Var(w_i x_i) &= \mathbb{E}(w_i)^2 Var(x_i) + \mathbb{E}(x_i)^2 Var(w_i) + Var(w_i)Var(x_i) \\ &= Var(w_i)Var(x_i) \end{aligned} \tag{24}$$

The expected values are 0, such that the variance of w_i and x_i is just the product of their variances. Assuming that the variances of the weights and the input in a single layer are all the same, equation 23 can be expressed as

$$Var(y) = n * Var(w)Var(x) \tag{25}$$

, where n is the number of neurons in the corresponding layer. Now we want the variances of the output y and the input x to be the same. Therefore we want to know the variance of the weights.

$$\begin{aligned} & Var(x) &&= Var(y) \\ \Leftrightarrow\ & Var(x) &&= n * Var(w)Var(x) \\ \Leftrightarrow\ & 1 &&= n * Var(w) \\ \Leftrightarrow\ & \frac{1}{n} &&= Var(w) \end{aligned} \tag{26}$$

In case the size of y is not equal to the size of x, we have to average the input and the output size:

$$Var(w) = \frac{1}{(n_i + n_{i+1})/2} = \frac{2}{n_i + n_{i+1}} \tag{27}$$

To adjust the variance of the uniform distribution to the variance described in equation 27, we needed to choose the boundaries of the interval the random values are drawn from, since the variance of a uniform distribution is given by:

$$Var(x) = \frac{1}{12}(b-a)^2 \tag{28}$$

Where a is the lower bound and b the upper bound of the interval. We assume that $a = -b$. Now we just have to solve the following equation in order to achieve interval boundaries that lead to a uniform distribution with the desired properties.

$$\begin{aligned}
& \frac{2}{n_i + n_{i+1}} && = \frac{1}{12}(a-b)^2 \\
\Leftrightarrow\quad & \frac{2}{n_i + n_{i+1}} && = \frac{(2b)^2}{12} \\
\Leftrightarrow\quad & \frac{2}{n_i + n_{i+1}} && = \frac{4b^2}{12} \\
\Leftrightarrow\quad & \frac{24}{n_i + n_{i+1}} && = 4b^2 \\
\Leftrightarrow\quad & \frac{6}{n_i + n_{i+1}} && = b^2 \\
\Leftrightarrow\quad & \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} && = b \qquad (29)
\end{aligned}$$

In summary this means that we can use a uniform distribution with mean 0 to initialize our weights without running into the problem of too large or too small variances when we set the interval boundaries as seen in equation 29 to adjust the variance of our distribution to $2/(n_i + n_{i+1})$ for each layer, where n_i is the input size and n_{i+1} the output size of layer i.

Now we will have a look at the experimental results. We trained the GAN model 8 times for 50000 epochs with a batch size of 64 and a learning rate α of 0.001 and calculated the means of the loss values (Fig. 12).

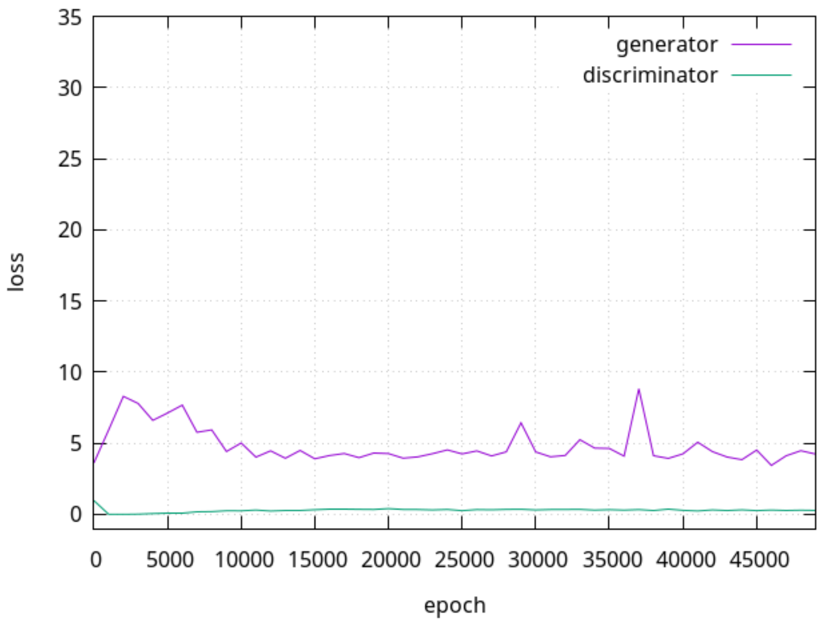


Figure 12: Progress of GAN models loss values over 50000 epochs with a batch size of 64 and a learning rate of 0.001. The pink line shows the generator's loss, the green line shows the discriminator's loss.

Analyzing the plot, we noticed that at the very beginning the discriminator's

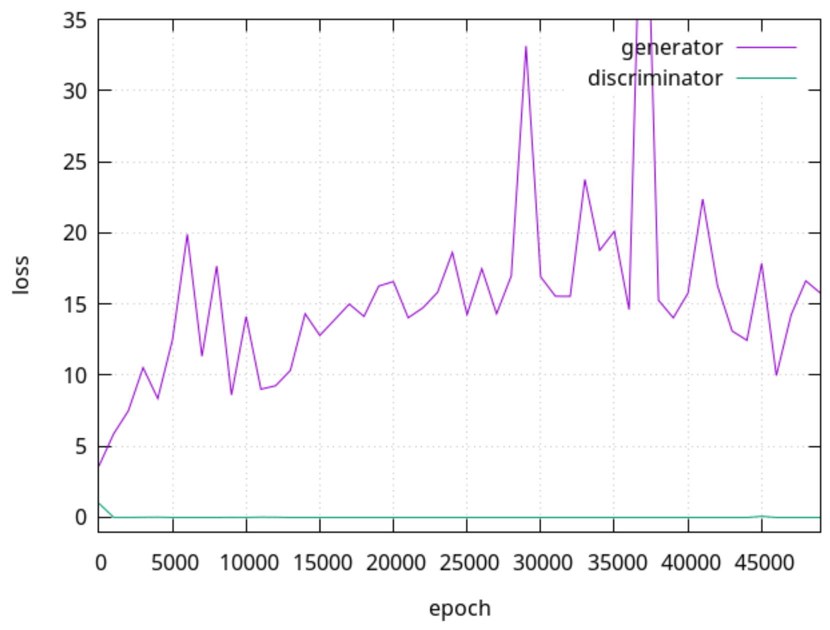


Figure 13: Failed GAN experiment.

loss value goes to 0, while the generator's loss reaches a maximum of about 8. The reason for this is the strong discriminator in the first epochs. Here, most of the generator's outputs are rejected whereupon it slowly progresses and minimizes its loss. At the same time the discriminator gets better, too, which makes training difficult for the generator. The discriminator's loss stays about constant near 0, while the generator's loss decreases at first and then varies around a certain value unstably. All in all both the generator and the discriminator converged to a certain value. While the discriminator converged to 0, the generator converged to a value around 4. The fact that the generator's loss converges to a value greater than 0 is a rather normal behavior of the GAN model. If both networks had a loss near 0, this would mean that the discriminator distinguishes real data from generated data with high accuracy and at the same time the generator generates data that is always classified as real data. These two cases exclude each other. The somehow unstable loss progression of the generator arises from the one GAN experiment that failed(Fig. 13).

Looking at the actual generated data (Fig. 14), we noticed an overfitting of the generator on one certain digit, which in all experiments was 1. A possible reason for this is that the only goal of the generator is to generate data that is classified as real data by the discriminator. The easiest way to achieve this goal is to specialize on a single digit. We expect this issue to resolve itself for the models that depend on a context vector.

The second model we examined is a CGAN. The difference between the GAN and the CGAN model is the extra input that contains some information the generated data depends on. As described in section 2.4, we combine the noise vector and the context vector by simply concatenating them. For generator and discriminator we again use ANNs that share most of their network parameters (Tab. 3). Both

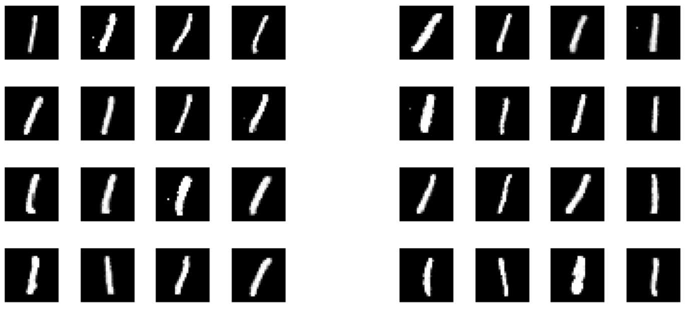


Figure 14: Generated samples of the GAN model showing the overfitting problem.

networks consist of three layers, just as in the GAN model, but here the number of neurons in the input layer is increased by the context size. We do this in order to fit the size of the input vector which for the CGAN model is the noise vector concatenated with the context vector. The context vector has a size of 10, because it is a one-hot-vector representing one of the ten possible labels. We have to do this for both networks, because they both have the context vector as an additional input. The generator uses the context to generate data that depends on this context while the discriminator uses the context to know the label of the input data in order to classify it as real or generated data. All other parameters stay the same as they are for the GAN model.

	Generator	Discriminator
N (per layer)	noise size + context size, 392, 784	784 + context size, 392, 1
loss	Sigmoid cross entropy with logits	
optimizer	ADAM	
k	-	1
batch size	64 / 128	
noise size	100	-
context size	10	
α	0.001	
initializer	Glorot normal distribution	
epochs	50000	

Table 3: CGANs network settings. N describes the number of neurons per layer (input, hidden, output) for generator and discriminator. The second and third line are the loss and optimizer functions. k is the number of the discriminator's training steps per each training step of the generator. Batch size is the number of training examples per epoch. Noise size describes the size of the generator's input vector that is then formed to a 28×28 image. Context size describes the size of the context vector the generated data depends on. α is the learning rate. The initializer describes the way the variables are initialized. Epochs is the number of training epochs.

We also ran the CGAN experiments 8 times and expected the loss progression to behave similar to the GAN model, because of the similarity of these two models. Unfortunately, its behavior was completely different. Instead of decreasing, the loss value of the generator was increasing constantly, while the discriminator acted the same as in the GAN model (Fig. 15). Thus the generated samples do not show any digits at all (Fig. 16). By adding the label as an additional condition to the generator's input, the task of generating MNIST-like data becomes even more difficult because now the generator is not able to specialize on a specific label. If the generator still specialized on a single digit no matter what the given context vector was, the discriminator would instantly classify the input vector as generated, because of the wrong label. The discriminator's task is still the same, except for the fact that it has additional information about the label of the input vector that will be classified. We solved this problem by increasing the number of training samples to 128 samples per epoch as seen in Fig. 17. Thus the raised difficulty of the generation task is balanced by the increased number of training samples. But the deviations of the loss values still adumbrate some instability of the training process.

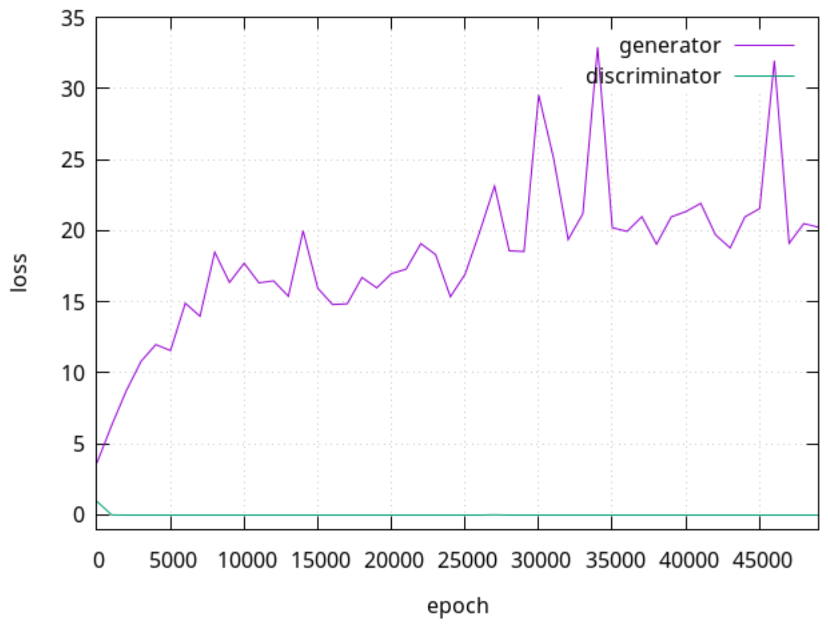


Figure 15: Progress of CGAN models loss values over 50000 epochs with a batch size of 64 and a learning rate of 0.001. The pink line shows the generator's loss, the green line shows the discriminator's loss.

Having a look at the generated data (Fig. 19), we examined that concatenating the label as a context vector to the input noise vector solves the overfitting problem that occurred in the GAN model. The generated samples clearly show the digits that were given by the context vector and only a few of them are hard to recognize. It is no surprise that some of the generated digits are hard to recognize, because the original digits sometimes are too (18). Unexpectedly, in one of our 8 experiments the network was not able to learn properly (Fig. 20).

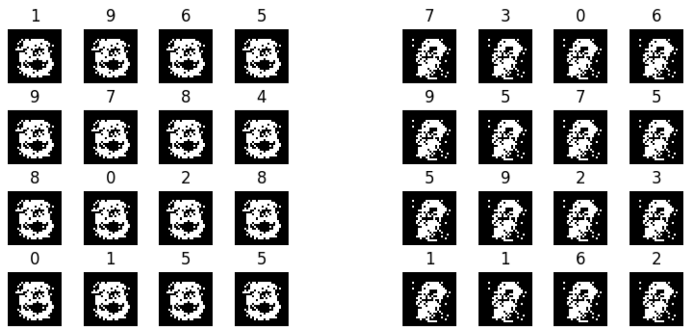


Figure 16: Generated samples of the CGAN model with a batch size of 64. The network was not able to generated any digits.

The third model we trained is a GARNN. This model is similar to the GAN model insofar as it uses exactly the same network parameters (Tab. 2), but uses RNNs instead of ANNs for both generator and discriminator. RNNs add the ability of processing sequential data while storing and regarding information about the preceding inputs. In order to process MNIST data sequentially, we divided each image into four pieces as seen in figure 21. For the generator we do the same with the noise vector. The training then works like this: The generator's input vector will be divided into four parts. Next we pass the sequence part by part to the multi layered RNN. By doing so the RNN will store information about the already processed data. After processing the whole sequence we obtain an output vector of size 392, which will then be multiplied with a weights matrix in the final output layer and finally results in an 28×28 MNIST-like image. For the discriminator we split the input image into four parts and also pass the sequence element-wise to the multi layered RNN, which then results in a single scalar. We have not expected the model to behave different to the GAN, because the only change is the use of RNNs instead of ANNs and at a first view at the MNIST data, we thought that processing the images sequentially would not bring us any advantages. However, the results of the GARNN were slightly better than those of the GAN, because the model was not overfitting as much as the GAN was, but still the diversity of the samples is low (Fig. 22). Examining the loss values of the model (Fig. 23), we again observe that the GARNN behaves very similar to the GAN, except for the increased stability of loss progression due to the use of RNNs. Altogether the results show that the use of RNNs within a GAN can improve the model even with a data set that is not usually processed sequentially.

Last but not least, we will have a look at the CGARNN, that processes data sequentially using RNNs and generating samples depending on a certain context. Taking the results of the previous three experiments into account, we expect the CGARNN to combine the advantages of both properties, namely a more stable learning process and generated samples of high diversity due to the context. For the network settings we use the same settings as the CGAN model (Tab. 3, 64 training samples per batch) and divide the input vectors into four parts the same way we

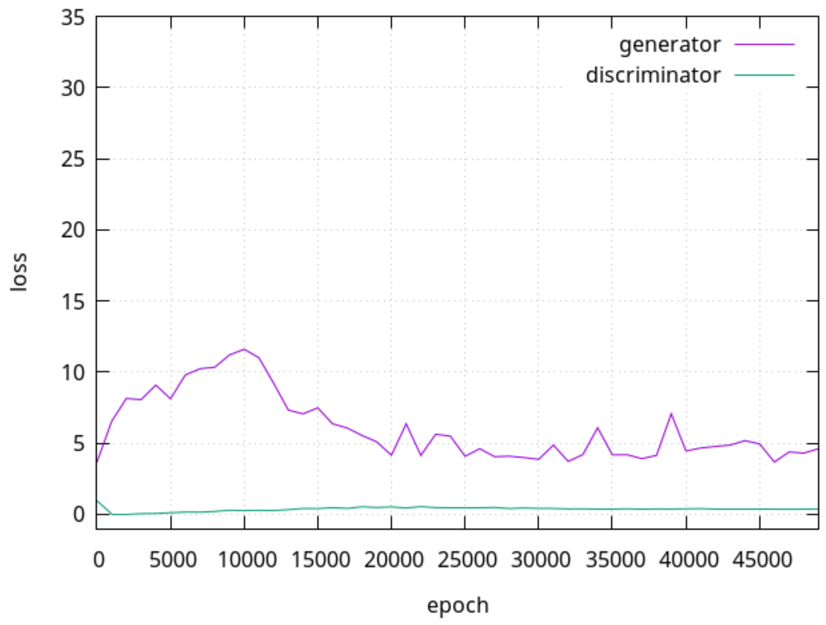


Figure 17: Progress of CGAN models loss values over 50000 epochs with a batch size of 128 and a learning rate of 0.001. The pink line shows the generator's loss, the green line shows the discriminator's loss.

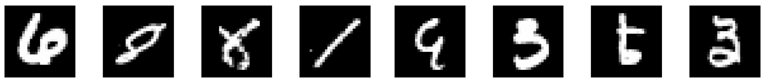


Figure 18: Samples from the MNIST data set showing that there are samples in the original data that are hard to recognize, too.

did for the GARNN. The results confirm our assumptions. Figure 24 shows that the use of RNNs results in a much more stable training process than the use of ANNs (Fig. 17). The generated samples (Fig. 23) show that the use of the context vector results in a higher diversity and prevents the model to overfit, compared to a model without an additional context input (Fig. 22).

Concluding, the experiments showed, that both the sequential processing of the data and the additional context vector improved the learning process and the quality of the generated samples immensely. In order to compare the results of all four models, we combined the loss progressions of the models to two plots, one for the generator (Fig. 26) and the other for the discriminator (Fig. 27). In addition we opposed samples from each model to each other in Figure 28.

3.2 Football Events Data

In this section we analyze the experiments on the Football Events data set[3] and their results. Prior to this, we will have a look at the data set itself. Then we will examine the performance of our model in four experiments in which we will

[3]`https://www.kaggle.com/secareanualin/football-events` (also listed in the appendix)

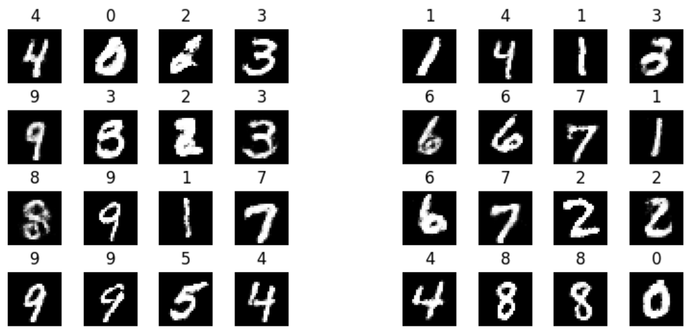


Figure 19: Generated samples of the CGAN model. The number above the corresponding sample is the label that was fed to the network as a context vector.

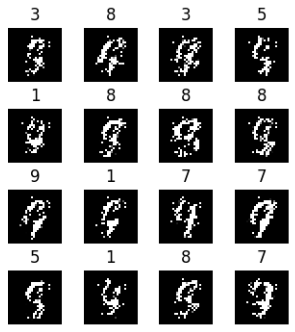


Figure 20: Failed experiment of the CGAN model with a batch size of 128.

use different input and context representations and discuss their advantages and disadvantages. In the last experiment, we train a basic LSTM network on the same task and compare its results to those of the CGARNN model.

3.2.1 Data set

The Football Event data set is a csv file that contains information about 941008 game situations of 9074 different matches. The information is separated in 22 classes: The game specific ID *id_odsp*, the event specific ID *id_event*, the *sort_number* that describes the order in which the events occurred in a game, the *time* that describes the minute in which the event occurred, the *text* that is a description of the event itself, the event types *event_type* (primary event) and *event_type2* (secondary event) that describe the kind of event, the *side* that describes whether the main player participating in this event belongs to the event team, the *event_team*, the *opponent*, the *player* participating in the primary event, the player *player2* participating in the secondary event, the *player_in* that describes the player entering the field in case of a substitution, the *player_out* that describes the player leaving the field in case of a substitution, the *shot_place* that describes the placement of the shot, the *shot_outcome* that describes if the shot was placed were it was intended to be, the

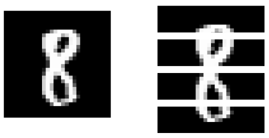


Figure 21: Splitting of MNIST samples in order to use them sequentially.

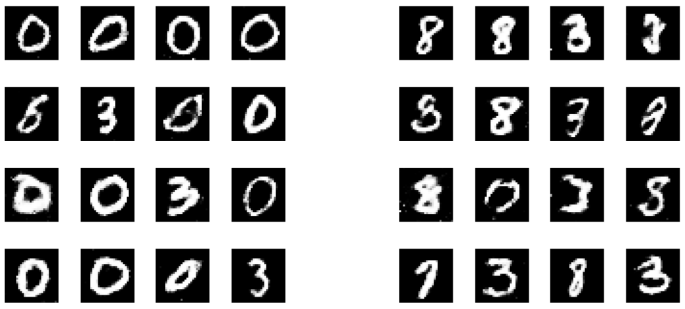


Figure 22: Generated samples of the GARNN model

is_goal that describes if the shot resulted in a goal, the *location* on the field where the event took place, the *bodypart* that was used, the *assist_method* in case of an assisted shot, the *situation* in which the event took place and the *fast_break* that describes if the event was followed by a time-out. Except for the text, the players and the teams, each class in the data set is represented by IDs or *NA* (not available). In order to process *NA*, we replaced it with -1 for each class. The classes *is_goal* and *fast_break* are naturally binary, but there is one exception for the *is_goal* class. In case of an own goal *is_goal* is set to -1. In order to be able to process the players and clubs, we simply enumerated them and thereby gave them IDs as well. The remaining, more complex classes are described in table 4, 5, 6 and 7.

Not all of these 22 classes are actually relevant for our experiments. The classes we do not need are the game specific ID *id_odsp*, the event specific ID *id_event*, the *sort_number*, the *time* and the *side*. Since the goal of the experiments is the achievement of a model that is able to generate context sensitive event descriptions and the six above-mentioned classes do not influence this description, we will ignore them in our experiments.

3.2.2 Preprocessing

First, we need to preprocess parts of the data set in a way that we can use it for the experiments. In order to use the texts from the data set as input sequences for our model, we have to know the different elements the texts are made of. These are club names, player names, other words, punctuation and special characters. Since an event description can contain several sentences, a full stop does not indicate that the end of the text was reached. Therefore, to mark the end of a text, we added $ as an additional symbol. Next, we analyzed all event descriptions in the data set for frequencies of their elements. With this we determined the IDs for each

ID	*event_type*	ID	*event_type2*
0	Announcement	12	Key Pass
1	Attempt	13	Failed through ball
2	Corner	14	Sending off
3	Foul	15	Own goal
4	Yellow card	-1	NA
5	Second yellow card		
6	Red card		
7	Substitution		
8	Free kick won		
9	Offside		
10	Hand ball		
11	Penalty		

Table 4: Possible values for the classes *event_type* and *event_type2*.

ID	*shot_place*	ID	*shot_outcome*
1	Bit too high	1	On target
2	Blocked	2	Off target
3	Bottom left corner	3	Blocked
4	Bottom right corner	4	Hit the bar
5	Center of the goal	-1	NA
6	High and wide		
7	Hits the bar		
8	Misses to the left		
9	Misses to the right		
10	Too high		
11	Top center of the goal		
12	Top left corner		
13	Top right corner		
-1	NA		

Table 5: Possible values for the classes *shot_place* and *shot_outcome*.

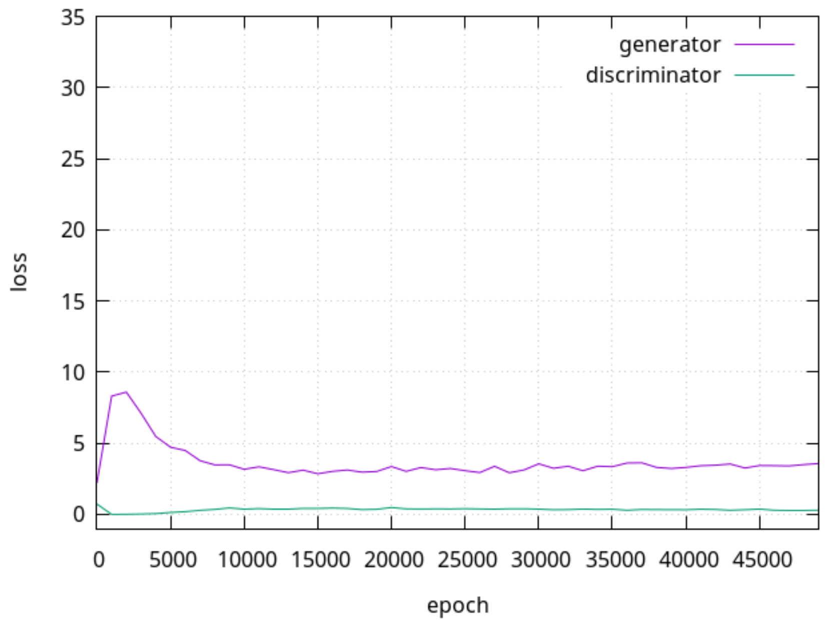


Figure 23: Progress of GARNN model's loss values over 50000 epochs, with a batch size of 64 and a learning rate of 0.001. The pink line shows the generator's loss, the green line shows the discriminator's loss.

element in the description, starting at 0 for the most frequent element and then further enumerating the remaining elements. The context information must be preprocessed, too. Fortunately, most of the event information is already represented by IDs. The only information that is not represented by IDs are the classes *event_team*, *opponent*, *player*, *player2*, *player_in* and *player_out*. We therefore need mappings for clubs and players. These were obtained simply by building a list of all players and clubs in the order they occurred in the data set and then enumerated them starting at 0. This way, all context information for the event descriptions could be represented as numeric values, too. All experiments use the event descriptions and their context vectors as an input, so we want to preprocess them as far as possible, such that while running the experiments only mapping of text elements to indices must be done. That means that we provide a file containing the event descriptions as strings, where all words and special characters are already separated by blanks, and their corresponding context vectors, representing all information by their IDs. The above-mentioned preprocessing must be done for all experiments. For some of the experiments there are more necessary preprocessing steps that will be explained in the corresponding section.

3.2.3 Experiment I

For the CGARNN model experiments on the Football Events data set, we decided to use word embeddings to represent words and other sequence elements such as punctuation characters or the end of line symbol $. Word embeddings are a popular representation of data when it comes to tasks such as text generation because they

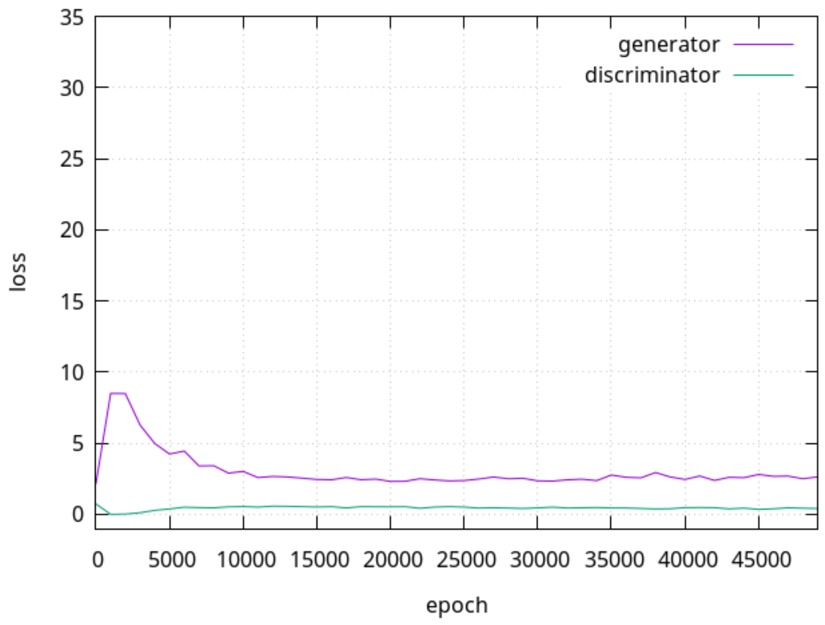


Figure 24: Progress of CGARNN model's loss values over 50000 epochs, with a batch size of 64 and a learning rate of 0.001. The pink line shows the generator's loss, the green line shows the discriminator's loss.

encode similarity of words and linguistic structures and improve the quality of the generated texts. A simple alternative to word embeddings is an index representation where each possible sequence element is assigned an integer value. This way, the word representation is rather arbitrary and does not contain any information about the word itself or about the relation to other words. There are many approaches that benefit from the usage of word embeddings instead of simple index representation (e.g. [1, 2]). To obtain the word embeddings for our dictionary, we followed the approach that was introduced by Mikolov et al. in 2013 [15, 16], namely the Skip-gram model. In order to use this model to obtain the word embeddings, we have to do some more preprocessing. The Skip-gram model uses subsequences of the original texts where each of the subsequences contains a target word and a certain number of history and future words that are directly nearby. We therefore prepared a tsv file containing one target word and up to four future and history words per line. If the target word is the first word in the sequence there are no history words and for the last word in a sequence there are no future words, respectively. The goal of the Skip-gram model is then to maximize the average probability (Eq. 30) that describes how well our model predicts the nearby words given the target words, where T is the length of the observed sequence, c is the number of history and future words and w_t is the current subsequence's target word.

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq i\leq c, i\neq 0}\log p(w_{t+i} \mid w_t) \tag{30}$$

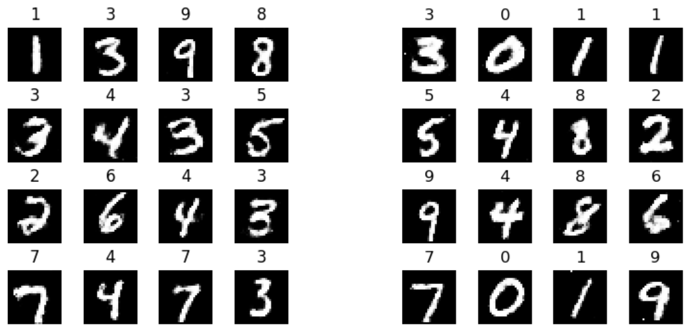


Figure 25: Generated samples of the CGARNN model

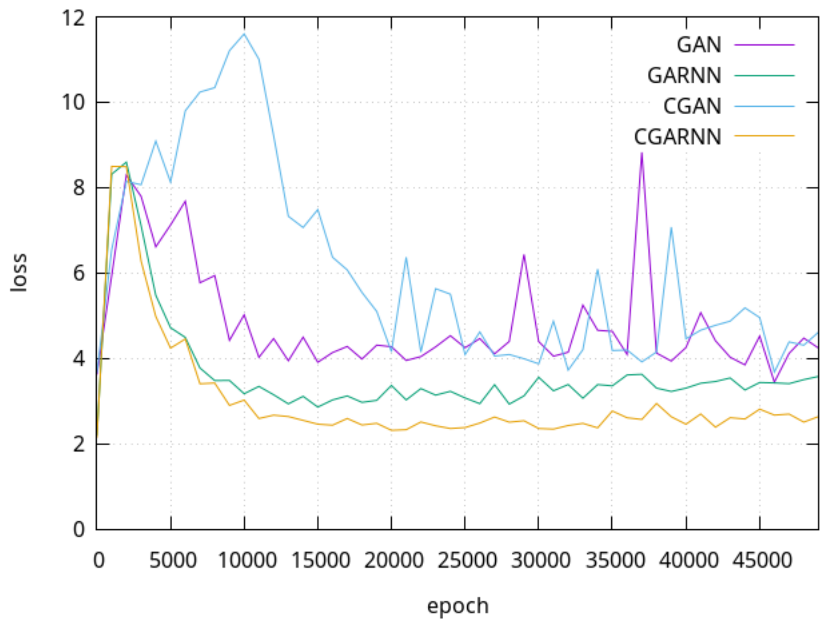


Figure 26: Comparison of loss progression for the generators of all models.

In [16], Mikolov et al. discuss various ways of expressing the probability function p in equation 30. The first way is to describe p using the softmax function

$$p(w_{t+i} \mid w_t) = \frac{\exp(v'_{w_{t+i}})^{\top} v_{w_t}}{\sum_{w=1}^{W} \exp(v'^{\top}_{w} v_{w_t})} \tag{31}$$

that can be used to describe the probability of an event with a certain number of outcomes where all the probabilities sum up to 1. Here the number of possible outcomes is the size of the vocabulary W. v and v' describe vector representations for input and output data. As described by Goldberg and Levy [9], the different vector representations are used to avoid assigning a high probability in case of the target and the nearby word being the same, because in texts, a word is usually not

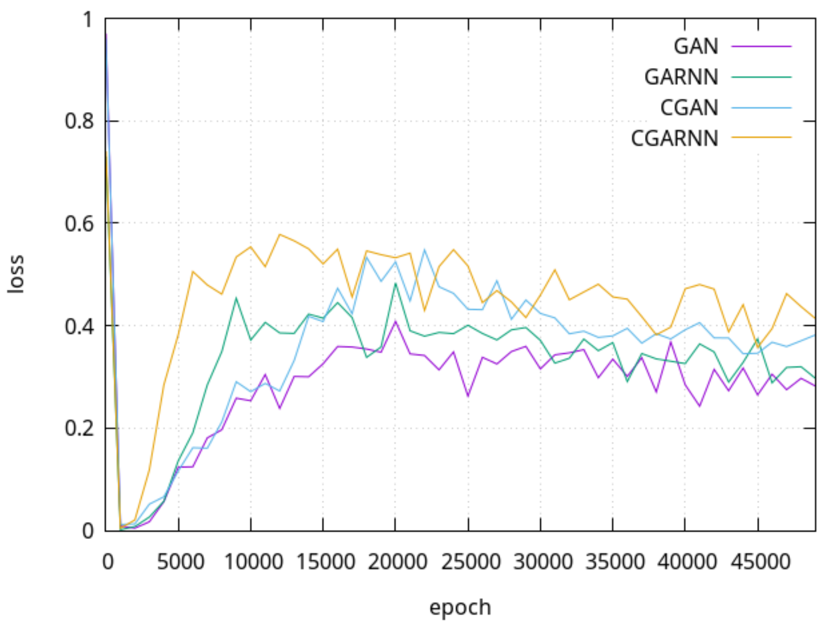
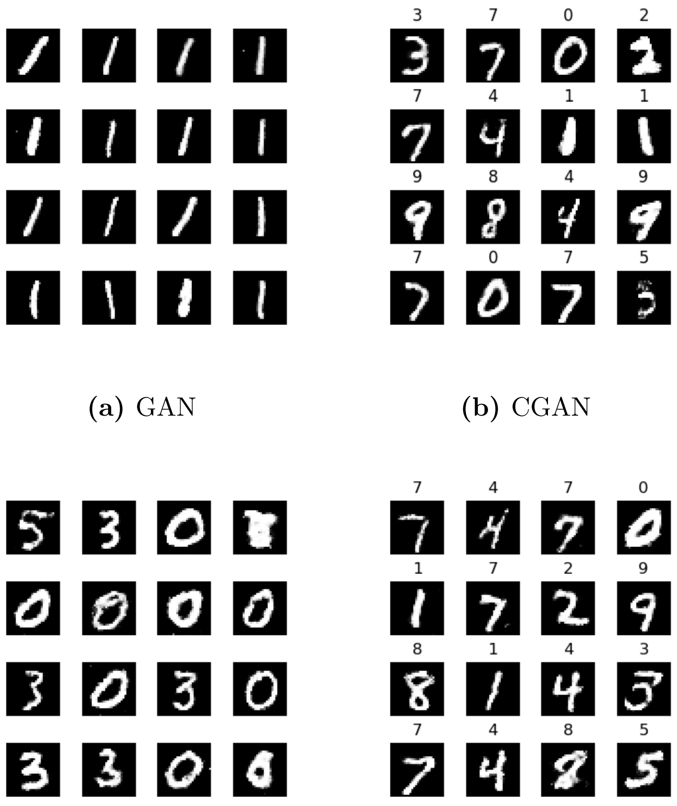


Figure 27: Comparison of loss progression for the discriminators of all models.



(a) GAN **(b)** CGAN

(c) GARNN **(d)** CGARNN

Figure 28: Comparison of generated samples.

ID	*location*	ID	*location*
1	Attacking half	11	Right side of the box
2	Defensive half	12	Right side of the 6yd box
3	Center of the box	13	Very close range
4	Left wing	14	Penalty spot
5	Right wing	15	Outside the box
6	Difficult angle and long range	16	Long range
7	Difficult angle on the left	17	More than 35yds
8	Difficult angle on the right	18	More than 40yds
9	Left side of the box	19	Not recorded
10	Left side of the 6yd box	-1	NA

Table 6: Possible values for *location* class.

ID	*body_part*	ID	*assist_method*	ID	*situation*
0	Right Foot	0	None	1	Open play
1	Left Foot	1	Pass	2	Set piece
2	Head	2	Cross	3	Corner
-1	NA	3	Headed pass	4	Free kick
		4	Through ball	-1	NA

Table 7: Possible values for the classes *bodyparts*, *assist_methods* and *situation.*

followed by itself. Using the softmax function leads to the objective function

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq i\leq c, i\neq 0}\log\frac{\exp(v'_{w_{t+i}})^{\top}v_{w_t}}{\sum_{w=1}^{W}\exp(v'^{\top}_{w}v_{w_t})} \tag{32}$$

The problem of the softmax function is the computational complexity due to W. For each tuple of target word and nearby word $\exp(v'^{\top}_{w}v_{w_t})$ must be calculated for all the words in the vocabulary. Since the vocabulary in text generation tasks is rather large, we aim to use a probability function that is independent of the vocabulary size. To this end, Mikolov et al. introduced the concept of negative sampling [16]. Negative sampling is a modified version of the Noise-Contrastive Estimation as described by Gutmann and Hyvärinen in 2010 [11]. Noise-Contrastive Estimation defines the objective function

$$\frac{1}{2T}\sum_{t=1}^{T}\log\sigma(\log p(x_t)-\log p_n(x_t))+\log(1-\sigma(\log p(y_t)-\log p_n(y_t))) \tag{33}$$

where p_n is the probability distribution the noise samples y_t are drawn from and x_t describes the samples drawn from the real data. This means that the objective is to approximate the probability distribution describing the real data by distinguishing real samples and noise samples. Modifying Noise-Contrastive Estimation as in [16] leads to

$$\log\sigma(v'^{\top}_{w_{t+i}}v_{w_t})+\sum_{j=1}^{k}\mathbb{E}_{w_j\sim p_n(w)}(\log\sigma(-v'^{\top}_{w_j}v_{w_t})) \tag{34}$$

as a representation for $\log p$, where $\log \sigma(v'^{\top}_{w_{t+i}} v_{w_t})$ is the probability of the word w_{t+i} being a nearby word of the target word w_t and $\log \sigma(-v'^{\top}_{w_j} v_{w_t})$ is the probability of the noise vector being a nearby word of w_t. Therefore the Skip-gram objective function is:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq i\leq c, i\neq 0}(\log \sigma(v'^{\top}_{w_{t+i}} v_{w_t}) + \sum_{j=1}^{k}\mathbb{E}_{w_j\sim p_n(w)}(\log \sigma(-v'^{\top}_{w_j} v_{w_t}))) \tag{35}$$

This means that we take k negative samples per pair of target word and nearby word to approximate the data's probability distribution and thereby reduce the computational complexity, which is now independent of the vocabulary size. In order to speed up the learning process even more and to improve the vector representations of words that do not occur often, we use the concept of subsampling during training. Subsampling rejects certain words per training iteration depending on their frequencies. All pairs of target and nearby words that contain any of the rejected words are not considered in the corresponding iteration. Mikolov et al. provided a heuristically chosen function that can be used to reject more frequent words:

$$P(w) = 1 - \sqrt{\frac{\theta}{f(w)}} \tag{36}$$

where w is a word, $f(w)$ the word's frequency and θ a threshold for the word frequency. If the frequency of a word exceeds θ it is assigned a high probability of being rejected for the current iteration. In [16] they chose $\theta = 10^{-5}$ heuristically. Since our data set contains much less words, we increased θ to 10^{-3}. In order to obtain the word embeddings, we trained a neural network consisting of an input layer, a projection layer and an output layer. The projection layer is a matrix of size *vocabulary_size* $\times$ *embedding_size* that maps each index to a vector representation of the corresponding word. Using the Skip-gram objective function (Eq. 35) as the network's loss function, we trained the network for 1000000 iterations with an embedding size of 32 and minimized the loss with gradient descent. After training, the matrix that was used for the projection layer contains the learned word embeddings. We saved those embeddings to a file in order to use it for further computations.

Since there is no standard method for evaluating word embeddings [7] and due to its complexity, we decided to evaluate the word embeddings by representing them graphically and checking the resulting plots for noticeable patterns and regularities. In order to represent high dimensional vectors graphically, we have to map them to a two dimensional vector space first. Therefore, we use t-distributed stochastic neighbor embedding (t-SNE) as described by van der Maaten and Hinton in 2015 [19]. We consider two sets of data points X and Y, where X is the set of the original high dimensional data points and Y is the set of the corresponding data points in the two dimensional space. Then we assign a probability that x_i and x_j are nearby points by

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n} \tag{37}$$

to each pair of data points where $p_{j|i}$ is the probability that x_j is a neighbor of x_i and vice versa for $p_{i|j}$, using the softmax function

$$p_{j|i} = \frac{\exp(- \parallel x_i - x_j \parallel^2)/2\sigma_i}{\sum_{k \neq l} \exp(- \parallel y_k - y_l \parallel^2)} \tag{38}$$

$\parallel \cdot \parallel$ denotes the Euclidean distance. For the set Y, we also describe a probability distribution

$$q_{ij} = \frac{\exp(- \parallel y_i - y_j \parallel^2)}{\sum_{k \neq l} \exp(- \parallel y_k - y_l \parallel^2)} \tag{39}$$

that describes the probability that y_i and y_j are neighbors. The goal is then to minimize the difference between the probability distributions p and q because we want nearby data points in the high dimensional space to be nearby in the two dimensional space too. To this end, the Kullback-Leibner divergence KL (Eq. 40) is used, which is the sum of the cross-entropy of p_i and q_i and the entropy of p_i for a fixed data point i.

$$\begin{aligned} KL(p_i \parallel q_i) &= -\sum_j p_{ij} \log q_{ij} + \sum_j p_{ij} \log p_{ij} \\ &= \sum_j p_{ij} \log p_{ij} - \sum_j p_{ij} \log q_{ij} \\ &= \sum_j p_{ij} \log p_{ij} - p_{ij} \log q_{ij} \\ &= \sum_j p_{ij} (\log p_{ij} - \log q_{ij}) \\ &= \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \end{aligned} \tag{40}$$

The loss function describing the overall difference of p and q for all pairs of data points is then described by

$$KL(p \parallel q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{41}$$

In order to minimize the loss function, gradient descent is used. In our experiments we used the t-SNE implementation of the scikit-learn[4] Python library for machine learning. Figure 29 shows an extract of the learned vector representations.

Having a vector representation for all words and characters unfortunately leads to the problem that we cannot restrict the generated vectors to the subset of the vector space that actually represents valid elements. In comparison, the problem would not occur when using an index representation instead, because the output would then be a bag-of-words - a vector containing as many values as the vocabulary does, where each of these values would represent the probability of the element with the corresponding index. The idea is to find the nearest vector to the generated

[4] `http://scikit-learn.org/stable/index.html`

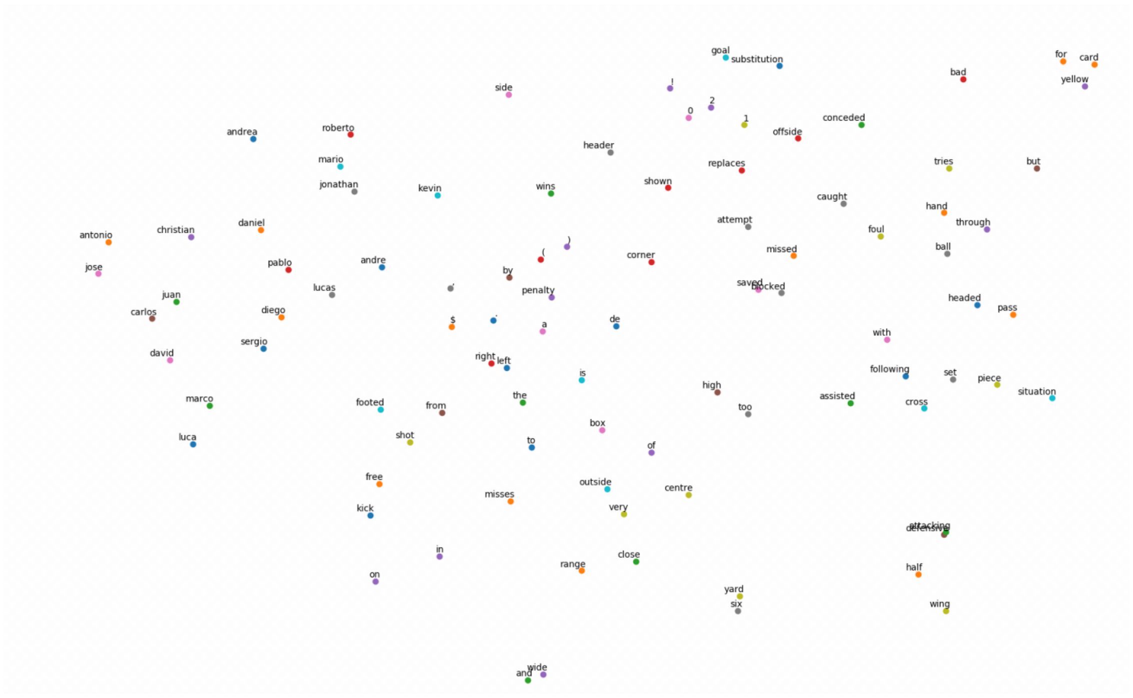


Figure 29: Extract of the learned word embeddings with four history and future words in the Skip-gram data and an embedding size of 32. The plot does not contain all words in the vocabulary and is reduced to the 200 most frequent words for the purpose of plot clarity. We see that many words that are either used together or in a similar context are nearby, e.g. the numbers 0,1,2, left and right, the left and right parenthesis, $ and full stop and so on. The complete plot is listed in the appendix.

one. What makes the nearest neighbor search difficult is the high dimensionality of the vector space. The most naive way to determine the nearest vector would be the calculation of the Euclidean distance of the generated vector to all those that are actually representing sequence elements. The time complexity per batch would be $\mathcal{O}(batch\_size \times dictionary\_size \times embedding\_size)$, where $dictionary\_size$ is the number of possible sequence elements.

To avoid the calculation of all the Euclidean distances, we will use an optimized form of a multidimensional binary search tree as described by Friedman, Bentley and Finkel in 1977 [8]. Multidimensional binary search trees are also known as k-d-trees. Before we describe the optimized version of the k-d-tree, we will have a look at the standard k-d-tree as described by Bentley in 1975 [4]. A k-d-tree is a binary search tree with k-dimensional node elements. To build a k-d-tree, we start with an initial, empty tree and insert all given vectors gradually. Therefore, we start with the first element V_0 and make it the tree's root node in layer 0. For all the remaining vectors we will continue in the same manner we would for regular binary search trees. Each new node we want to insert into the tree must find its proper place such that the order persists. The main difference is the way how we compare the current node with the one we want to insert. Depending on the current layer, we have to compare a different dimension of the vectors. Each layer i is assigned a vector dimension

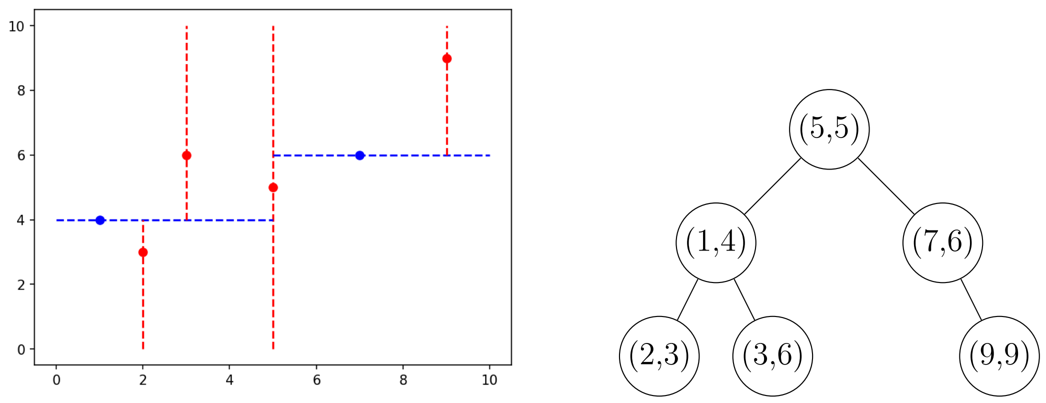


Figure 30: Example for a k-d-tree with k=2. The points $(5,5),(1,4),(2,3),(3,6),(7,6),(9,9)$ were added to the tree in this order. Left: Graphical representation of a plane that is divided into subspaces by points in tree. Right: k-d-tree after inserting all the points.

$d = i \mod k$. If the value of the inserted node in dimension d is smaller than that of the current node, the insertion will be continued in the left subtree, otherwise it will be continued in the right subtree. In the next step, dimension $d+1$ will be used for comparison. In case $d+1 \geq k$, we use the first dimension ($d = 0$) again. This results in a binary tree in which each node in the i-th layer can be understood as a point in a k-dimensional space that divides the space in the d-th dimension into two subspaces (Fig. 30).

When inserting the vectors gradually in this manner, there is no guarantee that the resulting tree is balanced. This possibly makes a nearest neighbor search rather inefficient. Therefore, we use an optimized k-d-tree [8] instead. For building an optimized tree, we have to know all the vectors in advance. Instead of choosing dimension d to divide the space into two subspaces with regard to the depth of the corresponding node, d is determined in the following way: For each space division we look at all values for each vector in the current subspace. Since we want to divide the space in a way that we separate those vectors that are rather different to each other, we use the dimension with the highest variance of the values to do so. Thus the space is divided into two largest possible subspaces where the "left" subspace contains all the vectors having a smaller value than the mean in dimension d and the "right" subspace contains all the vectors with larger values accordingly. This procedure always results in a balanced tree and guarantees that the tree will be of logarithmic depth. For the nearest neighbor search, the concept of buckets is added to the optimized k-d-tree. Instead of dividing the space at every vector in our data, the tree is only divided at a certain number of vectors such that each of the resulting subspaces contains a predefined number of vectors, the bucket size. These subspaces are called buckets and are represented by the leave nodes of the tree. Thanks to the modified division of space, we can assume that all buckets contain almost the same number of vectors. After building the optimized k-d-tree it can be used for a nearest neighbor search in vector space [8]. When searching for the nearest neighbor of a given query vector Q, we first start a binary search for the bucket where the nearest neighbor can most likely be found. Keep in mind that the dimension at each node in the tree is still the same as the dimension we used while building the tree. Having found the desired bucket, we determine the nearest neighbor among all the vectors in the bucket. The distance to the nearest neighbor then decides if the search is finished. Regarding the hypersphere that is formed when using the determined distance as a radius, we check if it exceeds the boundaries of the current subspace. In that case, it is possible that there is a nearer vector to Q in one of the intersecting subspaces. So we have to calculate the distances for those vectors, too. Even if a large amount of intersecting subspaces could worsen the time complexity, Friedman et al. showed that the expected time complexity is logarithmic and therefore is advantageous for our experiment.

Now that we have described all the conditions for the experiment, we will have a look at the model itself. The network specifications for this experiment were the following: For weight initialization we used the Xavier initialization again. The generator was a 5 layer LSTM model, the discriminator a 2 layer LSTM model. We chose a higher number of layers for the generator because the task of generating word vectors is more complex than the task of distinguishing between real and generated sequences. The generator's input then is a batch of random noise vectors drawn from the same vector space as the word embeddings. Each noise vector will be concatenated with a random context vector that contains 16 context variables, namely *event_type*, *event_type2*, *event_team*, *opponent*, *player*, *player2*, *player_in*, *player_out*, *shot_place*, *shot_outcome*, *is_goal*, *location*, *body_part*, *assist_method*, *situation* and *fast_break* as described in section 3.2.1. The drawn context vectors are limited in a way that combinations that cannot occur because they contain conflicting information e.g. *is_goal = 1* and *event_type2* = 15 (own goal) will not be generated. Also, there is certain information that must be available in case of some events e.g. for *event_type* = 7 (substitution) the variables *player_in* and *player_out* must contain valid player IDs. The concatenated vectors are used to produce the batch of the first elements of the sequences. Afterwards ,the last batch of generated elements will be concatenated with the same labels again and used as the next input for the generator and so on. This leads to the problem that we have to decide when to stop the sequence generation. The most obvious way to do so is to stop the generation when the end of sequence symbol $ was generated. Unfortunately, this approach leads to even more problems. Early in learning, the $ most likely will not be generated at all such that the generator will generate an infinite sequence, which cannot be handled in practice. The second problem is that even if the $ will be generated at some point, the generated sequences are not of the same length. We therefore must pad them to the same length in order to use them for further computations, but we do not know the maximal length in advance. This way we cannot define the dimensions of the generator's output. To avoid these two problems, we looked for the longest sequence in the Football Events data set, which was 54, such that we always ask the generator to generate sequences of exactly that size. Therefore the generator's output dimensionality is $54 \times batch\_size \times embedding\_size$. After mapping the generator's output vectors to their nearest neighbors and concatenating them with the context vectors, we will pass them to the discriminator. In conclusion, the discriminator's

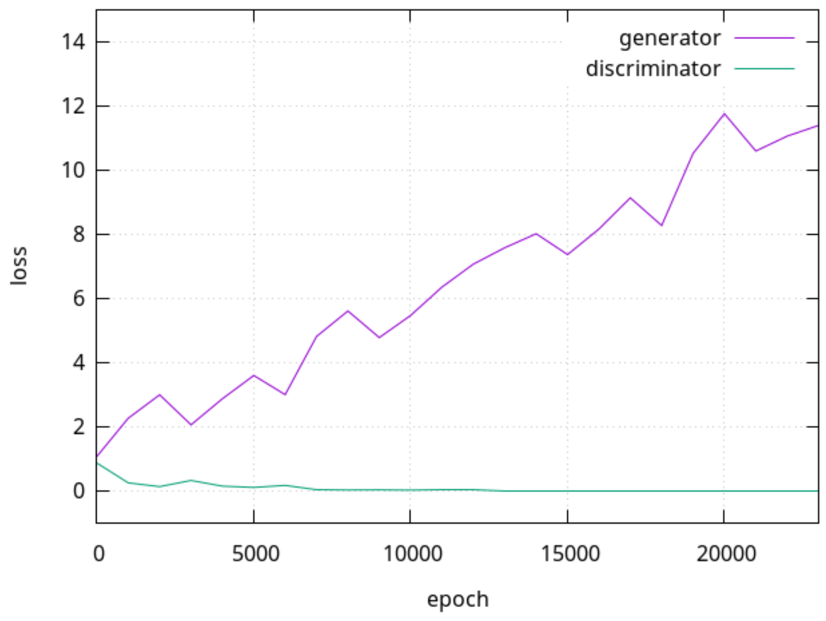


Figure 31: Loss progression of the CGARNN model on the Football Events data set with word embeddings.

input dimensionality is $54 \times batch\_size \times (embedding\_size + 16)$. Since the generator only generates sequences of length 54 and therefore the discriminator only takes input of the same length, we have to pad all samples drawn from the data set to the length of 54 by appending \$'s when using real data as the discriminator's input. The output of the discriminator is then a scalar between 0 and 1, representing an estimation of the input sequence to be generated (0) or real data (1) with regard to the concatenated context vector.

We trained the model for 24000 iterations (~12 hours) using the above-mentioned network parameters. The progression of the loss values (Fig. 31) unfortunately shows that the generator was not able to converge whereas the discriminator's loss reaches the minimum of 0 very early in training. Observing the generated samples (Fig. 32) we see that the network is clearly not learning to generate texts, because it just repeats a few words over and over again. Moreover, the generated words are names only. Having a closer look at the learned word embeddings by plotting all words in the data set instead of only the 200 most frequent (Fig. 33), we see that the Skip-gram model was not able to learn proper vector representations for names. We expected the names to be nearby in the vector space because they occur in similar contexts. Even if the names total frequency is large (97.7%) because a name occurs in almost every sentence of the data set, a single name occurs rather infrequent which makes the learning of a proper vector representation difficult. The names are evenly spread over the whole vector space and due to the large number of names, any nearest neighbor search will most likely return a name.

cole cole cole cole cole cole cole cole cole cole cole cole cole cole cole cole cole ryan ryan ryan ryan ryan ryan forster

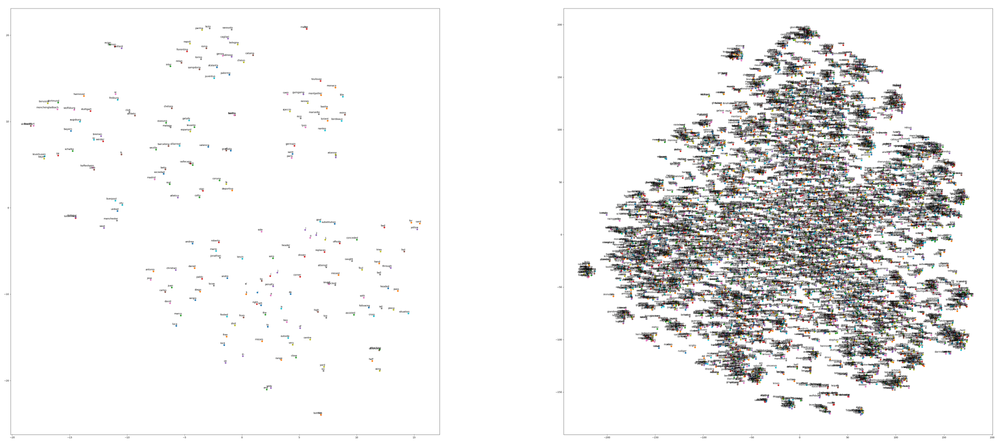
joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe joe sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo sabo

cole cole cole cole cole cole cole cole cole cole cole cole cole ham ham ham enner enner enner enner enner enner enner enner enner enner enner tokelo

Figure 32: Generated samples of the CGARNN model on the Football Events data set with word embeddings. The samples are made of names only. A file containing all generated samples is listed in the appendix.

3.2.4 Experiment II

Due to the difficulties that result from the weak word embeddings representing names, we decided to rerun the experiment with new word embeddings where all player names are represented by the *PLAYER* placeholder and all clubs by the *CLUB* placeholder. Therefore the total number of name occurrences does not change but we only have to learn two vector representations for all of them. Since the player names are encoded in the context vector, the placeholders can easily be replaced after the text was generated. We therefore expect the nearest neighbor search to be more successful and improve the text generation. We again trained the model for 24000 iterations ($\sim$ 7 hours) using the same network and parameters as described in section 3.2.3 and word embeddings with placeholders instead of actual names. The loss value's progression slightly improved (Fig. 34). Where the generator that was trained on word embeddings with names reached a loss value of about 12, the replacement of the names with placeholders reduced the generator's loss by roughly 50%, leading to a value of about 6. The discriminator still converges to a loss value of 0 but takes more iterations to do so compared to the first experiment. Still, the loss progression shows that the generator is not able to assert itself against the discriminator. Examining the generated samples in figure 35, we see that the weak word embeddings for names in the first experiment were actually a problem. If a sentence actually contained a high number of names, the model should now generate sequences containing the placeholders with a high ratio. However, the problem of the missing variances within the sequences is still unsolved.



(a) (b)

Figure 33: Learned word embeddings with four history and future words in the Skip-gram data and an embedding size of 32. **(a)** 200 most frequent words. Almost no names. **(b)** All words. The plot clearly shows that the name's word vector were not learned properly and are evenly spread. A high resolution versions of the images are listed in the appendix.

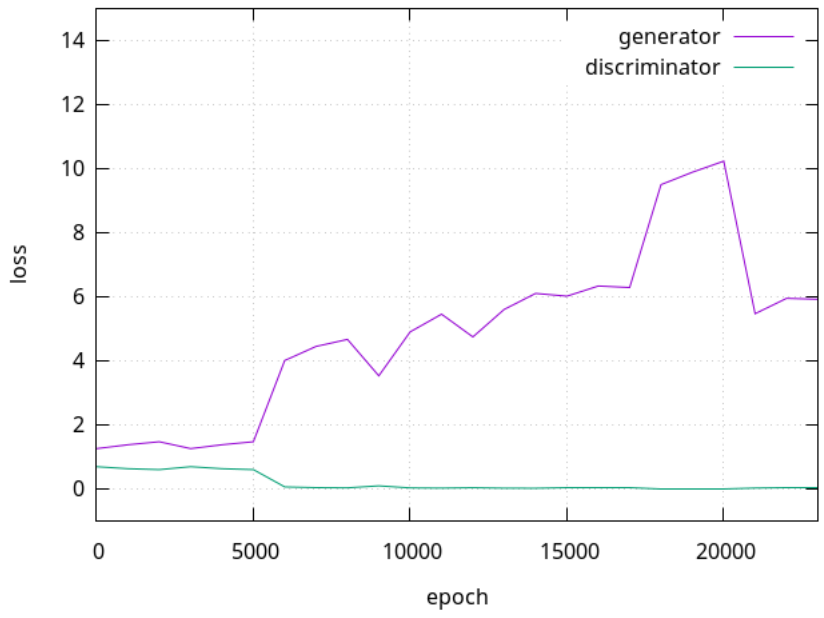


Figure 34: Loss progression of the CGARNN model on the Football Events data set with word embeddings and placeholders for names.

fighting fighting fighting fighting fighting fighting fighting fighting fighting fighting fighting fighting violent vio-lent violent violent violent violent violent violent violent violent violent violent violent

62' 62' 62' 62' 62' 62' 62' 62' 62' 62' fighting fighting fighting fighting fight-ing fighting

six six six six six six six six six half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half half

Figure 35: Generated samples of the CGARNN model on the Football Events data set with word embeddings and placeholders for names. The samples are made of a few words that are repeating steadily. A file containing all generated samples is listed in the appendix.

3.2.5 Experiment III

Facing the problem that the model still is not able to generate proper sequences, we want to investigate its behavior when changing the generator's output from a single sequence element to a bag-of-words that describes the probability for all words in the dictionary. The generator's output size will therefore increase to the size of the vocabulary. We want to keep this size small in order to reduce the duration per training iteration and because of the problems mentioned in section 3.2.3. This is why we use name placeholders for this experiment again. Since we use the bag-of-words approach for this experiment, the word embeddings and therefore the k-d-tree becomes unnecessary. For the generator's input, we decided to use a simple index representation of words. The problem that occurs in this case is that we have to take the word with the largest value out of the bag-of-words after every element that was generated in order to use it as the generator's next input. Unfortunately, we noticed that the *argmax* function is not differentiable and therefore, we were not able to obtain any gradients at all. In order to solve this problem, we decided to use a bag-of-words as the generator's input, too. This way the generator's output can easily be used as its input again. The use of the bag-of-words also makes generating a noise vector for the generator much easier since we can simply generate a vector containing a probability for each word in the vocabulary. For generating noise samples while using indices, we could randomly choose an index. The problem with this approach is that the generator could be biased by the noise sample. For example, the noise input could be the index representing a full stop and the generator would only generate the end of line symbol $ since this is the most probable element that is following the full stop.

In order to make the results of the different experiments as comparable as possible, we again trained the model for 24000 iterations ($\sim$ 17 hours). Figure 36 shows a significant improvement in the loss progression. The generator's loss at the end of the training reaches a value of 4 while the discriminator never clearly reaches 0, other than in the last experiment where the discriminator reaches a value near 0 very early in the training. This indicates the difficulty of generating a single vector representing a word compared to the determination of a probability vector. While examining the generated sequences, we noticed that they have not profited by the improved loss progression and are still made of just a few words, repeating continuously.

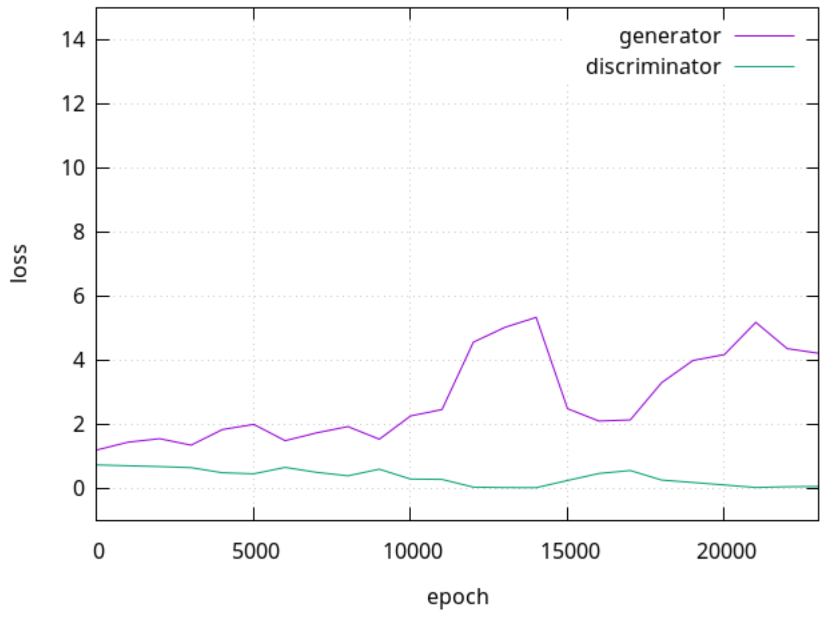


Figure 36: Loss progression of the CGARNN model on the Football Events data set with index representations for words and placeholders.

3.2.6 Experiment IV

We expected the model as described in section 3.2.5 to converge and generate good sequences, because it makes use of LSTMs and bags-of-words, which have often been used very effectively. Thus we started to look for the cause of the poor results in different places and found a possible reason in the generation of the random context vectors. In our experiments, the generator's context vectors were always randomly generated and never came from the original data set. As described in section 3.2.1, the context vectors consist of 16 different variables, each with a different number of possible assignments. The total number of possible context vectors exceeds $245 * 10^{24}$. Therefore, under the assumption that every vector is equally likely to be generated, the probability of generating the same context vector twice is nearly 0. This means that the generator will most probably see every context vector only once. In general, a model advances by doing the same task multiple times. In this case the problem *bit bit bit bit bit bit bit bit bit yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard yard*

bit bit bit bit bit bit $

bit bit bit bit bit bit bit bit bit bit bit bit bit bit 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 after

Figure 37: Generated samples of the CGARNN model on the Football Events data set with index representations for words and placeholders. The samples are made of a few words that are repeating steadily. A file containing all generated samples is listed in the appendix.

seems to be that the generator cannot make use of its feedback since it will never be prompted to generate a sequence with the same context vector more than once. In order to reduce the number of possible context vectors drastically, we only wanted to use context vectors that actually occur in the data set. Unfortunately, the total number of different context vectors in the data set is 25379 which leads to about 37 samples per context vector. Comparing this to established data sets like the MNIST data set that consists of 7000 samples per context vector or label, respectively, we realized that even this restriction will most probably not lead to better results. Thus, in order to increase the number of training samples per context vector, we will reduce the context vector's size from 16 to 2 variables. For the 2 remaining context variables we chose *event_type* and *event_type2* since they contain the game situation's most essential information. This further restriction leads to a total number of 11 context vectors and therefore to a number of about 85000 training samples per vector assuming that the context vectors are uniformly distributed. Since there are only 17 possible assignments for the two variables, we decided to represent them as two concatenated one-hot-vectors. Therefore each possible variable assignment has its own weight which makes treating these assignments differently easier than having a single weight for all of them. This was practically impossible in the previous experiments since the context vector would have exceeded a length of 1000 and therefore the duration per training iteration would have been impractically long. For this experiment, we prepared another text-context data file that contains all texts and the two corresponding event types.

We reran the experiment for 24000 iterations ($\sim$ 31 hours). Analyzing the model's loss progression (Fig. 38), we see that, against our expectations, the above-mentioned changes to the context vectors deteriorate the learning process instead of improving it. The discriminator again reached its optimum of 0 early in the training while the generator's loss increased continuously. Therefore the generated samples did not improve either (Fig. 39).

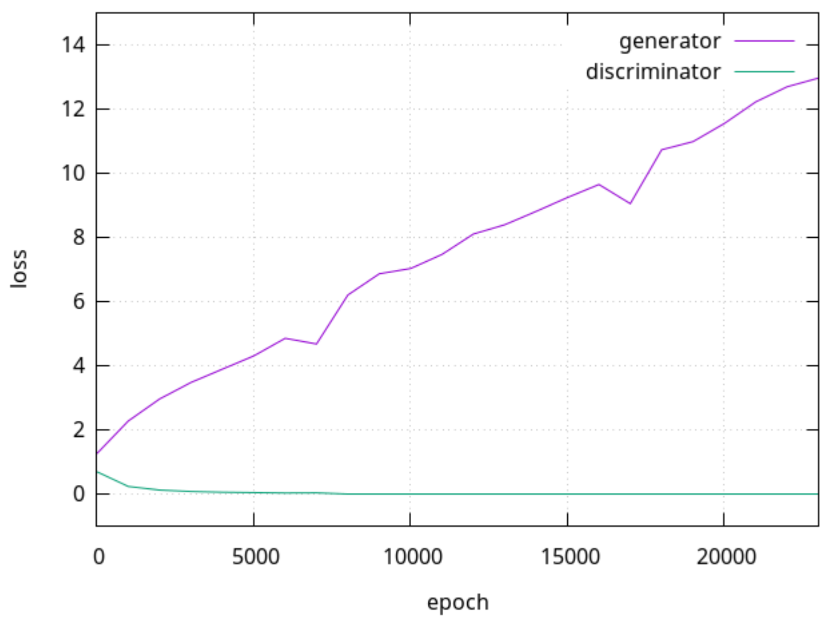


Figure 38: Loss progression of the CGARNN model on the Football Events data set with index representations for words and placeholders. The context vector was reduced in the way that it only consists of the event type variables.

3.2.7 Experiment V

In this last experiment, we used a basic LSTM model on the Football Events data set. The goal again was the generation of event descriptions with regard to certain context vectors. Since the results in section 3.2.5 were the best we could achieve with the CGARNN model, we decided to use the same preprocessed data for this experiment, namely an index representation for words with placeholders for players and clubs along with the complete context vector consisting of all 16 variables. The LSTM network used in this experiment consisted of three layers and 17 LSTM cells per layer. The number of cells per layer is determined by the context vector's size and the size of an input word which was represented by a single integer value in this experiment. The network's weights were once again initialized using the Xavier initialization. In each training iteration, we passed a batch of size 64 to the LSTM network. Each batch consisted of a subsequence of certain length - the step size, which was 3 in this experiment - that was drawn from the original data set and the corresponding sequence's context vector. We therefore used a random sequence from the data set and also randomly determined a start index for the subsequence. Then we took as many words as prescribed by the step size from this sequence, starting at the determined position. If the start position was too close to the sequence's end such that there were not enough sequence elements to build a subsequence of the desired length, we simply padded this subsequence with the end of line symbol $. When we actually passed the sequence elements to the LSTM network as its input, we concatenated the context vector of this subsequence to all its elements. Then the resulting input values were fed to the network successively. The network then *saved saved saved saved saved saved bit bit (((((((defensive defensive defensive defensive defensive side side side side side side side side side side by by by by by by by by by by by by by by by by by by by by by by by by by by*

bit bit bit bit bit bit bit bit bit bit set set set set set set set set set set capitalise capitalise bottom bottom bottom bottom bottom bottom bottom by by by by by by by by by by by by by by by by by by by by by by by by by by by

to to to to to to to to to to to to to to to to to to to post post defensive defensive defensive defensive defensive defensive defensive defensive defensive defensive defensive defensive defensive 75' 75' 75' 75' 75' 75' 75' 75' 75' 75' 75' 75' 75' 75' 75' 75' by to to to

Figure 39: Generated samples of the CGARNN model on the Football Events data set with index representations for words and placeholders. The context vectors only consisted of the event type variables. The samples are made of a few words that are repeating steadily. A file containing all generated samples is listed in the appendix.

predicted the next word following those that were given as the input, represented as a bag-of-words. The loss value was determined by comparing the network's prediction and the actual word that would have been next in the sequence by using the sigmoid cross entropy with logits. In order to minimize the loss function we used the ADAM optimizer.

We trained the LSTM model for 24000 iterations ($\sim$ 7 hours) with a learning rate of $\alpha = 0.001$. The resulting loss progression looked very promising since the model reached a small loss value early in the learning process. The loss progression of the LSTM model is depicted in figure 40. After the 24000 training iterations, we used the model to generate a batch of sequences (Fig. 41). We therefore fed a batch of start sequences and context vectors to the network, each consisting of the first three elements of samples randomly drawn from the data set. The network then completed the sequences to a length of 54 elements. The results indicate that the LSTM was able to learn some regularities of the sequences, e.g. a player is often followed by his club (*by PLAYER CLUB*) and sentences end with a full stop. Even though this is an improvement compared to the results of the CGARNN model, the generated sequences are of a low quality since the model is clearly overfitting on a few learned patterns such as the ones described above. Moreover, the generated sequences are rather generic and do not reveal any context sensitivity at all. We therefore drew the conclusion that the LSTM model learned some simple regularities within the data but is not able to sufficiently consider the specified context vectors. This results in sequences even worse than those that would result if we trained a basic LSTM model without any context vectors at all. This leads to the conclusion that the use of the context vectors clearly deteriorates the generated samples.

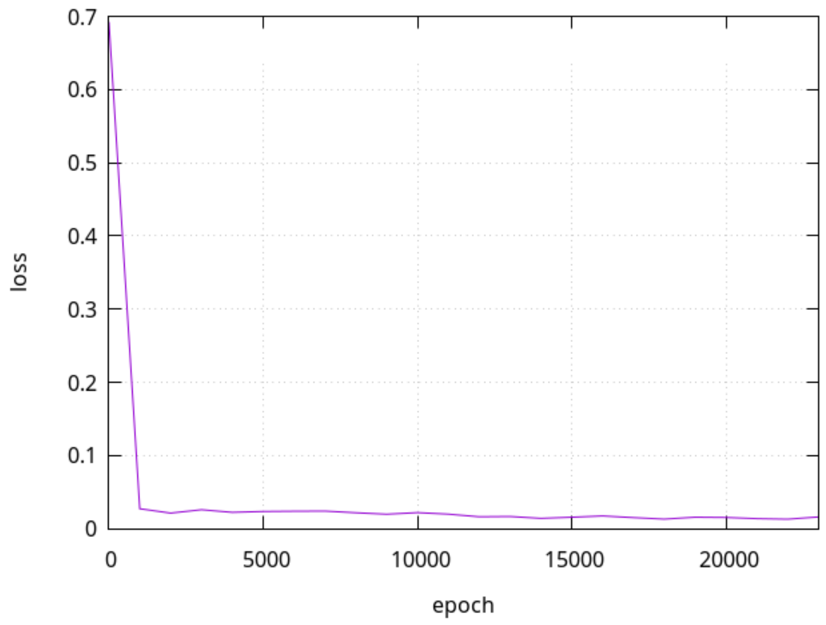


Figure 40: LSTM model's loss progression on the Football Events data set.

4 Conclusion

The experiments on the MNIST data set in section 3.1 clearly showed that the CGARNN model is able to generate high quality samples with respect to a certain context. Beyond that, we showed that it outperforms the basic GAN and CGAN models. Unfortunately, the subsequent experiments on the Football Events data set were unsuccessful since none of the applied network settings and data representations lead to sequences of high quality. The generated sequences could not even be recognized as sentences and therefore, we could not go a step further and verify the context dependency of the generated sequences. This indicates that one of the big problems of the CGARNN model is to find a data set that fits the requirements

attempt blocked CLUB *. by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER CLUB by PLAYER*

PLAYER (CLUB *CLUB) $ . . . .*

foul by PLAYER *CLUB . . . .*

Figure 41: Generated samples of the LSTM model. The words in bold were given to the network as a start sequence. The remaining words were generated.

such as containing sequential data whose labels consist of information that are represented by the sequences itself. A label containing too much information leads to the issue of having to represent it as a context vector that the CGARNN model is able to process properly and at the same time keep its dimensionality rather small. This is essential because a higher dimensionality of the model leads to a longer duration per training iteration. However, if the label contains too little information, the model will not be able to generate appropriate sequences due to the lack of information. As we were looking for a data set for the CGARNN experiments, the Football Events data set was the only (more or less) applicable data set we found.

During the composition of this thesis, Hyland, Esteband and Rätsch developed the same model we introduced in this thesis but referred to it as Recurrent Conditional GANs (RCGANs) [6]. They were able to generate synthetic medical data that could be used in the public domain since it does not contain data of actual patients. This confirms that the CGARNN model is indeed able to generate more complex sequential data with respect to a certain context. Moreover, the CGARNN model offers new possibilities since it seems to be able to provide domain specific data that can be used by publicly since critical and private information will not be included in this data. For this reason, we expect the CGARNN model to gain more relevance in many different domains in the future.

References

[1] Ehsaneddin Asgari and Mohammad RK Mofrad. Continuous distributed representation of biological sequences for deep proteomics and genomics. *PloS one*, 10(11):e0141287, 2015.

[2] Samy Bengio and Georg Heigold. Word embeddings for speech recognition. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[3] Y Bengio and X Glorot. Understanding the difficulty of training deep feed forward neural networks. pages 249–256, 01 2010.

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[5] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[6] C. Esteban, S. L. Hyland, and G. Rätsch. Real-valued (Medical) Time Series Generation with Recurrent Conditional GANs. *ArXiv e-prints*, June 2017.

[7] Manaal Faruqui, Yulia Tsvetkov, Pushpendre Rastogi, and Chris Dyer. Problems with evaluation of word embeddings using word similarity tasks. *CoRR*, abs/1605.02276, 2016.

[8] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[9] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *CoRR*, abs/1402.3722, 2014.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[11] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 297–304, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

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

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

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

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[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] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[18] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[19] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.