MIT Libraries | DSpace@MIT

MIT Open Access Articles

DeepKnit: Learning-based generation of machine knitting code

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Scheidt, F, Ou, J, Ishii, H and Meisen, T. 2020. "DeepKnit: Learning-based generation of machine knitting code." Procedia Manufacturing, 51.

As Published: 10.1016/j.promfg.2020.10.068

Publisher: Elsevier BV

Persistent URL: https://hdl.handle.net/1721.1/137098

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of use: Creative Commons Attribution-NonCommercial-NoDerivs License

Massachusetts Institute of Technology

Available online at www.sciencedirect.com

ScienceDirect

Procedia Manufacturing 51 (2020) 485–492

Procedia MANUFACTURING

www.elsevier.com/locate/procedia

30th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2021)
15-18 June 2021, Athens, Greece.

deepKnit: Learning-based Generation of Machine Knitting Code

Fabian Scheidt[a,b,*], Jifei Ou[c], Hiroshi Ishii[c], Tobias Meisen[a]

[a] *Institute of Technologies and Management of the Digital Transformation, University of Wuppertal, Wuppertal, Germany*
[b] *HotSprings GmbH, Aachen, Germany*
[c] *Media Lab, Massachusetts Institute of Technology, Cambridge, MA, USA*

Abstract

Modern knitting machines allow the manufacturing of various textile products with complex surface structures and patterns. However, programming these machines requires expert knowledge due to constraints of the process and the programming language. We present a long short-term memory (LSTM) based deep learning model that generates low-level code of novel knitting patterns based on high-level style specifications. To be processable by our model, we describe knitting instructions as one-dimensional sequences of tokens, which diverts from image-based approaches reported in previous research. We integrate our model into a design tool, that allows to assemble the atomic patterns to bigger swatches or garments. To evaluate our approach quantitatively, we formalize the requirements for patterns to be syntactically correct and valid to manufacture. Although our generated patterns look more random and seem to resemble less to human patterns, our evaluation shows that their knittability is orders of magnitudes better than randomly generated patterns.

© 2020 The Authors. Published by Elsevier Ltd.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of the FAIM 2021.

Keywords: Machine Knitting; Code Generation; Deep Learning; Textile Manufacturing; Artificial Intelligence

1. Introduction

Knitting is a mature textile fabrication technique especially used in fashionable and functional clothing. Modern machines allow fully automatic manufacturing of whole garments with complex textures without the need to assemble cut pieces manually. However, there is no abstraction or parametric design tool comparable to e.g. CAD/CAM systems in CNC manufacturing. Instead, knit programmers need to use stitch-level instructions to create patterns and simple shape primitives, which are then combined to form the machine code of whole garments. The design process oftentimes requires multiple iterations of knitting and tweaking of the machine code, because of the manual way of programming in combination with the hard to predict material behavior. Patterns are characteristic for knitted fabric, as they allow the creation of aesthetic textures while also giving versatile properties to the material. Yet, the design of novel patterns is challenging and laborious. Companies and suppliers try to overcome this difficulty by providing large libraries of patterns with code and preview images. The real objective, however, is to enable designers to create their own patterns, whose code is automatically generated and checked for feasibility.

Recent work presents parametric design tools for the assembly of shape primitives [1, 2] and automatic pipelines to convert 3D models to shaped knits [3, 4, 5]. Patterns are generated based on predefined pipelines, instruction-level procedures or from pictures of existing knit patterns [6, 7, 2, 8]. None of these approaches, however, allows users to generate patterns that are new and affirmed to be feasible to produce without prior knowledge on knit programming.

In recent years, deep learning has gained increasing interests and success in the domain of natural and formal language generation [9]. We make use of established text generation techniques and transfer them to the knitting domain to address the problems discussed above. In specific, we contribute a deep learning model that generates novel knitting patterns based on high-level style specifications, a procedure to validate, if a pattern can be knitted on a machine and a design tool that allows users to generate patterns and assemble them to a knittable swatch. This approach allows untrained users to create patterns that are feasible to produce and aesthetically pleasing.

* The work was conducted when the author studied at MIT.
E-mail address: fabian.scheidt@hotsprings.io (Fabian Scheidt).

2351-9789 © 2020 The Authors. Published by Elsevier Ltd.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of the FAIM 2021.
10.1016/j.promfg.2020.10.068

2. Background and Related Work

Knitting is a technique for producing two- or even three-dimensional fabric. It uses a yarn or thread to form a textile, which differentiates it from nonwovens, that are directly made from fibers without the intermediate step of making a yarn. Unlike weaving, where multiple perpendicular yarns are interlaced, knitting can create the fabric from a single yarn, by forming loops that interlock with each other [10].

2.1. Knit Programming

The manufacturing of knitted fabric on an industrial scale requires the use of purpose-made machines to produce large quantities. Unlike hand knitting, where all the unsecured loops are held on a single needle, in machine knitting each loop is held by its own hook-shaped needle. A large number of these needles are arranged in a row, called needle bed. A carrier positions the yarn in a location where the needles can reach it. Flat bed knitting machines – sometimes referred to as V-bed machines – have two opposing needle beds, which perform five fundamental operations to knit garments and patterns:

- **Knit.** A needle reaches to the yarn and pulls a new loop through all loops it held before. It then drops all the previous loops, since they are now secured by the new one.
- **Tuck.** Just like the knit operation, a needle reaches to the yarn and grabs it but does not release the loops that were held before. Tucking therefore adds an additional loop to a needle.
- **Transfer.** All loops held by a needle are transferred to the needle on the opposing bed. Transferring can be used to continue knitting on the opposing bed. Since the relative position of both beds can be offset, the combination of multiple transfer operations can be used to move loops sideways. The offset of the relative bed position is called *racking*.
- **Split.** The split operation combines the knit and transfer operation. Instead of dropping the loops attached to a needle after forming a new loop, those loops are transferred to the opposing bed.
- **Miss.** The needle performs no operation and keeps holding the loops.

For a comprehensive and illustrated description of these instructions, we refer to McCann et al. [1].

Combinations of these operations in conjunction with racking yield to higher level operations that make programming of shapes and patterns feasible. Each combined operation is identified by a number and visualized using a color as shown in figure 1. The operations outlined in this paper are specific to the *SDS-ONE APEX3* design system by Shima Seiki [11]. However, the outlined principle applies to machines and software of other manufacturers as well [12]. The available colors include the fundamental operations performed on one of the two beds. Move operations knit a loop and then move it sideways, thus forming a hole in the pattern. This is achieved by transferring the loop to the opposing bed, racking and transferring it again. Crossing loops can be created with adjacent move operations. Since they are commonly used and sometimes require to move loops further than one needle sideways, cross colors act as a shortcut for these instructions. Figure 2 shows examples for patterns knitted using fundamental, move and cross operations. The correlation between the position of a color in the code and its location on the fabric can be seen. A detailed review of operations and their usage for patterns and shaping can be found in the thesis of Underwood [13].

2.2. Knit Pattern Generation

Color patterns have successfully been created using jaquard knitting in previous research [14], so we consider patterns as surface structures knitted with only a single yarn. Karmon et al. apply the structure of photos onto knitted fabric using a mapping pipeline [6]. The generation of knitting instructions based on a high-level input is similar to our approach, but the output of the pipeline is limited to a small set of existing and recombined micro-patterns. Kaspar, Oh et al. build a code generator that uses photos of knitting as an input to a machine learning model [7]. Hofmann et al. show a method to convert instructions for hand knitting to machine code [8]. These works make existing patterns available to machine knitting but do not allow the creation of novel patterns. Further research of Kaspar et al. suggests the creation of novel pattern code using a domain specific language [2]. While this approach results in new and oftentimes non-repetitive patterns, it is not accessible to unskilled users, as it requires knowledge about knit programming, visual appeal and knittability.

No.	Operation	No.	Operation
1	Front knit	6	Front knit + move 1P left
2	Back knit	7	Front knit + move 1P right
11	Front tuck	40	Front knit + transfer
12	Back tuck	8	Back knit + move 1P left
101	Front split knit	9	Back knit + move 1P right
102	Back split knit	50	Back knit + transfer
0	Miss	4	Front stitch cross (1)
16	No needle selection	5	Front(-Back) stitch cross (1)
116	Front miss	10	Front-Back stitch cross (1)
117	Back miss	14	Front stitch cross (2)
		15	Front(-Back) stitch cross (2)
		100	Front-Back stitch cross (2)

Fig. 1: Selection of combined needle operations. Numbers and colors correspond to Shima Seiki software in default configuration [11].

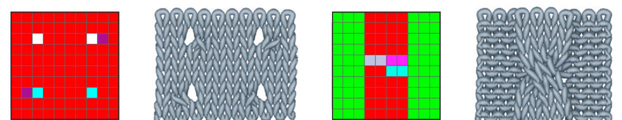


Fig. 2: Examples of knit code and renderings of the resulting patterns. Move operations are shown on the left, cross operations on the right.

2.3. Shaping of Knitted Products

Research on shaping of knitted products introduces the concept of parametric shape primitives that are assembled to create a desired shape [1]. Kaspar et al. take this approach further by creating an environment that allows to apply patterns to these primitives [2]. The pattern generator presented in this work could be integrated into this tool. The application of patterns to shapes generated from 3D models [3, 4, 5] is an open field of research.

2.4. Sequence Generation

Recent research on sequence generation is mainly on natural language generation. Tasks like summarization or image captioning [15, 16] are related to this work, as code is a formal description or summary of an outcome. Works on formal language generation, however, widely focus on the generation of code based on an informal representation like a description [17] or sketch [18]. The generation of novel sequences has been approached for text [9, 19] and music [20] since these do not have strong syntactical constraints. This becomes clear when looking at the work of Shane, who presents a generator for hand knitting instructions, that generates novel instructions but lacks syntactical correctness [21]. Since the outlined sequence generation tasks commonly make use of deep learning models using the long short-term memory (LSTM) cell proposed by Hochreiter and Schmidhuber [22], this work does so as well. We define a metric for syntactical correctness and show that an LSTM-based sequence generation model is able to generate valid knitting patterns.

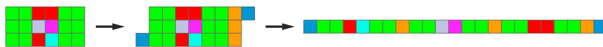


Fig. 3: Preprocessing of a pattern: End-of-line tokens (here shown in orange) are added to the right side of the pattern. Start- and end-of-sequence tokens (here shown in blue) are added to the bottom left and top right. The pattern is then split by row and concatenated.

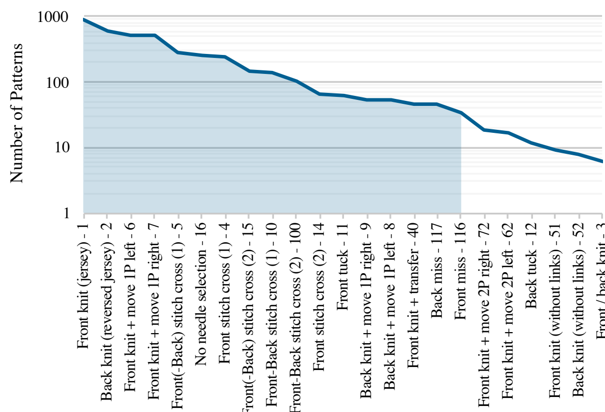


Fig. 4: Histogram showing the number of patterns that make use of each operation. The selection used as a filter is highlighted as an area under the graph.

3. Deep Learning Model

In order to train a deep learning model for code generation, we first discuss how knitting code should be represented. We build a labeled dataset of code samples and construct a model architecture that is used for training. We then show different decoding methods that can be used to generate new patterns.

3.1. Image vs. Sequence

As shown before, needle instructions are placed on a two-dimensional canvas, and can be edited using draw tools similar to those used in image processing. Therefore, it is fair to assume, that the code is a two-dimensional representation and should be treated like an image. However, there are distinct differences, justifying that it rather is a one-dimensional sequence:

- **Discrete colors.** Colors of images are elements of a color space and thus have an ordered relation to each other, that can be used to describe their similarity. In knit programming, color are completely categorical and have no order relation. For example there is no meaningful color between a front and a back knit.
- **No scale.** Scaling the code completely changes its meaning. Scaling a set of instructions does not make them taller or wider, but instead repeats them in adjacent courses.
- **No rotation.** Rotation not only changes the meaning of the code, but can also introduce syntactical problems. The colors for cables for example need to be used in pairs, which breaks when applying rotation.

Image generation networks make use of ordered colors, scale and rotation as they are trained towards the human perception. Since none of this applies to knitting code, we propose to treat the code as a sequence of discrete tokens with a fixed alignment and a constant scale, comparable to the handling of text in natural language processing. As shown in figure 3, two-dimensional knitting code can be processed as a sequence by adding a special end-of-line token at the end of each course. Additional tokens mark the start and end of the pattern. Concatenating the courses then leaves a one-dimensional sequence, that can be converted back to a two-dimensional pattern by splitting it at the position of the end-of-line tokens. This view matches the manufacturing, as the instructions are executed sequentially course by course, bottom to top.

3.2. Selection of Data

We extract 1978 human-made patterns from the library of the *APEX3* software [11] and a proprietary library provided by a manufacturer. The patterns are labeled with one or more of eight style categories, most of which are named closely to the main instruction used in the pattern. The **links** style uses only front and back knit colors to create wrinkled surface textures. **Cable** patterns twist loops sideways, by using the cross instructions. **Stitch move** patterns typically use move instructions

to create an arrangement of holes in the fabric. **Tuck** instructions can be used to make certain regions of the fabric thicker and therefore form characteristic patterns. **Miss** patterns create holes using move instructions and enlarge them by using miss. The miss operation makes the horizontal passes of the thread visible. **Stripe**, **jacquard** and **inlay** require more than one yarn to be knitted and can not be fully represented by the instructions explained before. These categories are therefore not considered here. We filter out patterns that are not rectangular or unusually big by limiting the width and height to 30 instructions each. In addition, we remove patterns that use uncommon colors. Figure 4 shows the number of patterns using different colors. The 17 most frequent instructions widely match those shown in figure 1. This leaves us with a dataset of 1171 different patterns.

3.3. Model Architecture

We convert the dataset into sequences and train a recurrent neural network to recreate the sequences of the training data following the methods of previous work [19] and common sequence generation techniques [23]. Our vocabulary consists of the 17 selected colors, the end-of-line, start and end token. The maximum sequence length is 932 tokens, which is a result of the maximum dimensions of 30 by 30, the maximum number of 30 end-of-line tokens and the single use of the start and end tokens. For the implementation all sequences need to have the same length, so we add a special padding token to the end of shorter sequences.

One-hot encoding is used to convert the discrete tokens into a vector representation. The five style categories are encoded in a five-dimensional vector ranging from zero to one, to represent percentages of style usage. The style vector is repeated and concatenated with the sequence to act as the model input. Since the model output should be a vector whose entries represent each token of the vocabulary, we use a dense layer with 21 cells as our last layer to match the vocabulary size. Because sequence generation problems with discrete tokens are effectively classification problems, we apply the softmax activation function to this output layer and use categorical cross entropy as our loss function. End-of-line and end-of sequence tokens are important for the alignment and termination of the pattern, so we weight them 10 and 100 times in the loss function to penalize their misplacement thoroughly. We measure the accuracy of the model as a fraction of the number of correctly classified tokens and the total number of predictions across all classes. As suggested by Kaspar, Oh et al. [7], we additionally define a foreground accuracy, that ignores the most common colors 1 and 2, since they make up more than 80% of the tokens in the training data.

Table 1: Metrics on different network architectures using LSTM layers

Hidden layers	Train FULL	Train FG	Test FULL	Test FG
3 x 500 cells	98%	98%	76%	67%
2 x 500 cells	98%	98%	78%	71%
2 x 100 cells	88%	80%	78%	69%
2 x 50 cells	76%	65%	74%	54%
2 x 100 cells + weights	81%	75%	79%	72%

We split the training data into a train and test set (90% and 10%) and evaluate different combinations of recurrent layers with dropout in between to reduce overfitting. We aim to find an architecture that has a similarly high accuracy on both sets (table 1), and performs well on the knittability and uniqueness metric outlined in the next chapter. As shown in figure 5, we settle on two LSTM layers with 100 cells each, since it achieves a reasonable accuracy while not overfitting the training set too much. We further increase the foreground accuracy by weighting color 1 with 0.8 and color 2 with 0.9 in the loss function, thus encouraging the model to be correct on all other colors in the training data. The resulting accuracy of 79% (72% foreground) is reasonably good since patterns can always be continued in more than one correct way.

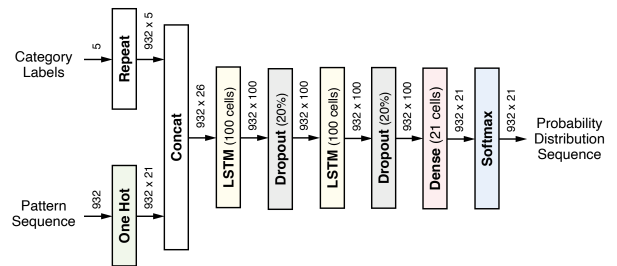


Fig. 5: Architecture of the pattern generation model

3.4. Sequence Decoding

The model is used as a many-to-many sequence prediction model: Patterns are generated by initializing the state of the recurrent layers randomly and providing an initial sequence to the model. The model is then asked to complete that sequence. To generate novel patterns, we provide only the start-of-sequence token as an initial input and generate a single new instruction. The generated instruction is added to the resulting code and fed back into the model, to produce the next instruction. This is repeated until the end-of-sequence token is generated.

The output of the model's final softmax layer is a probability distribution assigning a probability ϕ_i to each token v_i. It is not a single discrete token, so there are different methods to choose one. Instead of greedy decoding, which is taking the argmax over the distribution and thus picks the most likely token, we can sample randomly from the distribution [19]. This gives colors with similar high probabilities an almost equal chance to be sampled and thus allows the model to generate different pattern even if the initial state of the model is the same. Nevertheless, this also means that syntactical problems can be introduced. This behavior can be influenced by scaling the result of the output layer before applying the softmax function, so

$$\phi(\boldsymbol{x})_i = \frac{\exp(x_i/\tau)}{\sum_{j=1}^{n}\exp(x_j/\tau)} = \frac{\exp(1/\tau)^{x_i}}{\sum_{j=1}^{n}\exp(1/\tau)^{x_j}} = \frac{a^{x_i}}{\sum_{j=1}^{n}a^{x_j}} \quad (1)$$

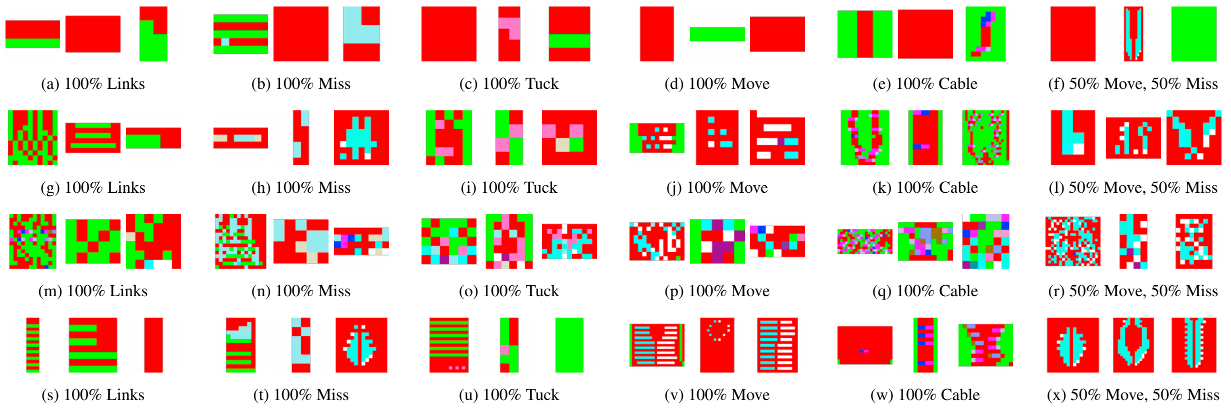


Fig. 6: Generated patterns: a-f greedy, g-l random with $\tau = 0.7$, m-r random with $\tau = 1.5$, s-x beam search with $k = 5$, $\tau = 3.0$, normalized by length

is effectively the activation function of the output layer, where τ is a positive parameter called *temperature*. The name originates from the way it influences the sampling: High temperatures give the generated output more diversity at the cost of it having more mistakes, while low temperatures make the output more conservative. The temperature effectively changes the base of the softmax function, which results in two special cases:

$$\lim_{\tau\to 0}\phi(\boldsymbol{x})_i = \lim_{a\to\infty}\frac{a^{x_i}}{a^{x_{max}}} = \begin{cases}1 & \text{if } x_i = x_{max}\\ 0 & \text{if } x_i \neq x_{max}\end{cases} \tag{2}$$

$$\lim_{\tau\to\infty}\phi(\boldsymbol{x})_i = \lim_{a\to 1}\frac{a^{x_i}}{\sum_{j=1}^{n} a^{x_j}} = \frac{1}{n} \tag{3}$$

For $\tau \to 0$ the distance in probability between the most likely token and the less likely ones increases until the highest probability is one and all others are zero. This makes the random decoding equal to greedy decoding. For $\tau \to \infty$ all probabilities are equal, so the decoding is effectively performed without using the predicted weights at all.

Greedy and random decoding expand a sequence by one token at each step and only keep one hypothesis, thus neglecting that other hypotheses could lead to a higher probability for the full sequence. It is intractable to generate all possible sequences to find the most likely one, so beam search can be used to get an approximation. This is done by expanding a sequence hypothesis by one token and then keeping the k best hypotheses instead of only one. Again, the temperature τ can be used to embrace less likely instructions and thus increase entropy [23].

A range of patterns are generated using the outlined decoding methods (figure 6). They are properly aligned and the dimensions are similar to those of the training set. Greedy decoding generates simple patterns, that mainly consist of the jersey colors 1 and 2. As a result, the instructions that are characteristic for a certain style rarely appear (figures 6a to 6f). Random decoding overcomes this problem: Low values for τ increase the diversity of the generated patterns, while keeping the patterns clean and maintaining the overall alignment. The randomness helps to emphasize the characteristic instructions for each category of style (figures 6g to 6l). Increasing the temperature increases the randomness, thus sacrificing the alignment and also occasionally breaking syntactical rules like the pairing of cross stitches. The influence of the style input decreases and untypical stitch combinations are mixed (figures 6m to 6r). Beam search has the same lack of diversity as greedy sampling: The most common colors are used excessively and the impact of the style input is low. Unlike greedy sampling though, beam search finds the pattern with the highest overall probability, so modifying the probability of each prediction changes the overall result. This is again done by increasing the temperature, thus lowering high probabilities and increasing low ones. In contrast to random sampling this does not make the patterns noisy though, which results in cleaner and more aligned results. However, the variety of patterns is still low (figures 6s to 6x).

4. Knittability

The nature of knitting's loop structure makes it prone to unravel, if one loop fell out of its position. In addition, the manufacturing process has physical limitations, that further restrict the feasibility of patterns. For these reasons, a creative generator creating previously unseen code is worth nothing, if the code can not be executed on a machine. This makes knittability the probably most important metric for a knit code generator.

4.1. Syntactical Correctness

The only syntactical requirements for patterns are caused by combinations of the cross colors (4, 5, 10, 14, 15, 100), since they are the only instructions that do not immediately translate into needle-level operations, as they need to come in pairs. If two adjacent pairs make the code ambiguous, the second pair of cross colors needs to be used. In addition, no more than three of the same color can be adjacent.

4.2. Process Reliability

Knitting machines will execute any code that is syntactically correct. Nonetheless, there are various circumstances that make parts of the knitting unravel, jam the machine during the production, break yarn or even needles. For this reason, additional checks need to be performed to ensure process reliability. While syntactical problems can be spotted on an instruction-level, reliability issues occur on the process-level, so there are various different combinations of instructions that cause the same type of complication. The following situations can be problematic:

- **Too many loops in a needle.** The size of a needle is physically limited to some number of loops. Multiple loops in a needle can be held by using tuck operations or moving adjacent loops to one needle. When a needle is full and can not take any more loops, new loops or those transferred to the needle are unintentionally dropped.
- **Loop held for too many courses.** When miss operations are performed on needles that hold loops, too many miss operations performed by the same needle put stress on the loops and thus also the needle holding them.
- **Transfer of too many loops.** As previously explained, there are different ways to make one needle hold multiple loops. When multiple loops are transferred to the adjacent bed, there is a greater chance that one of the loops is dropped while transferring it.
- **Too much racking.** The machine performs move operations using different amounts of racking at the same time. Loops are transferred to the opposing bed in a first step. It then racks to the left side and performs the necessary transfer operations before doing the same for the right side. This moves certain loops a very long distance causing the yarn to wear out and possibly break.
- **Continuous pickup stitch.** An operation that causes a previously empty needle to pick up the yarn is called pickup stitch. When adjacent needles perform pickup stitches and release the picked up yarn in the same later course, no loop can be formed. Thus, following operations continue to be pickup stitches.
- **Transfer of pickup stitch.** Since pickup stitches are held very unstable, they should not be transferred to the opposing bed, as they might be dropped.

We identify problems by building an environment that simulates all operations in the same order that the machine would do. This allows us to track the origin and position of all knitted loops at any step of the production and in the finished products. We check for the outlined problems after each needle operation and only consider a pattern knittable if no problem occurs. Since thresholds for problematic situations highly depend on the used machine and yarn, we follow the recommendations of knit experts: We only allow a maximum of three loops per needle, holding a loop for not more than seven courses, transferring up to two loops and racking no further than four needles. We further ignore transferring of pickup stitches, as they are unproblematic for most materials. Since patterns are typically repeated when used, we repeat the generated patterns horizontally and vertically and pad them with front knitting (color 1), to check for boundary conditions.

5. Model Performance

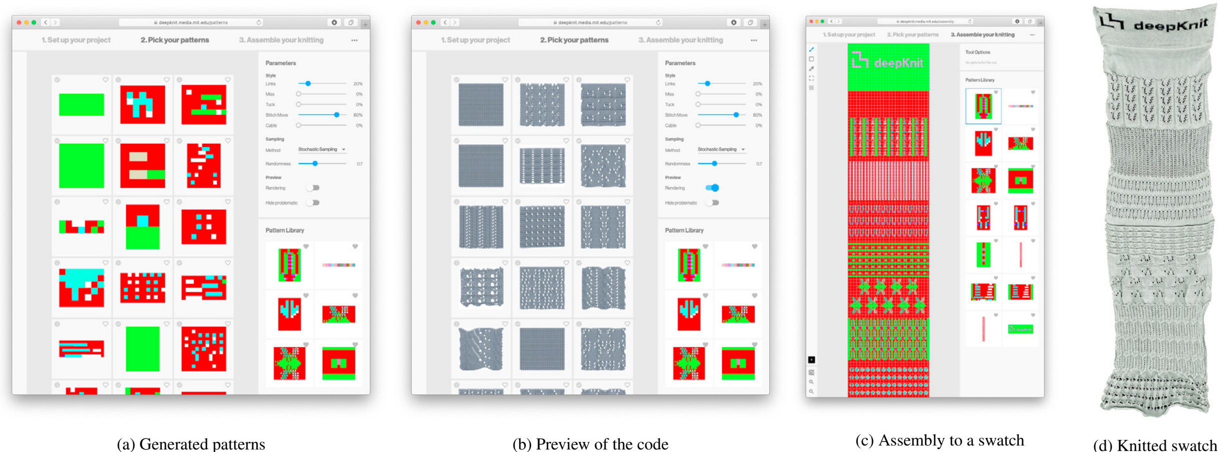
The performance of our model is evaluated by generating new patterns and checking their knittability. 1000 patterns are generated for each of the style combinations found in the training set. This process is repeated for different decoding methods and parametrizations. In addition to that, the uniqueness of the generated patterns is compared. Uniqueness is measured as the percentage of unique patterns. The results are shown in table 2.

Unsurprisingly, increasing the temperature of the random decoding increases the number of unique patterns, while also lowering the amount of knittable patterns. High temperatures increase the probability of less likely tokens to be generated, which is especially notable for end-of-line and end-of-sequence tokens. The generation of these tokens in inappropriate places can be identified by misaligned or short patterns. Infinite temperature means that the sequence is generated without considering the model output at all. For the generation of each instruction, there now is an equally high chance that the end-of-sequence token is decoded, while the model originally learned to use this token only in the end of a pattern. This causes the sampling to stop earlier and results in a decreasing uniqueness. Beam search has a similar behavior, but the overall knittability is higher while the uniqueness is lower. This result is in line with the way that beam search works: Because the probability distribution of the full pattern is considered instead of each individual instruction, the pattern is less noisy compared to random decoding. Single misplaced instruction like unpaired cross stitches are therefore less likely to appear and don't break the correctness of the entire pattern.

The overall knittability is very good, as it is close to 50% even for high temperatures, which means that one out of two patterns can be manufactured. The distribution of the dimensions of generated patterns is comparable to the dimensions of the training set. This is a very good result, when compar-

Table 2: Knittability, uniqueness and average pattern dimensions (courses x wales) for different decoding methods. Percentages are averaged across the style categories since the variation is insignificant.

Decoding Method	Knittability	Uniqueness	Average Size
Greedy	98.57%	17.84%	11 x 8
Random $\tau = 0.1$	98.45%	22.05%	10 x 8
Random $\tau = 0.5$	92.87%	70.18%	7 x 7
Random $\tau = 0.7$	87.89%	84.89%	6 x 7
Random $\tau = 1.0$	77.54%	90.22%	6 x 6
Random $\tau = 1.5$	56.78%	92.55%	5 x 5
Random $\tau \to \infty$	3.95%	91.91%	2 x 13
Beam Search $\tau = 1.0$	98.74%	17.59%	14 x 9
Beam Search $\tau = 3.0$	94.36%	39.55%	18 x 13
Beam Search $\tau = 5.0$	88.13%	50.72%	17 x 18



(a) Generated patterns (b) Preview of the code (c) Assembly to a swatch (d) Knitted swatch

Fig. 7: Making a swatch of generated patterns using our design tool

ing it to fully random generated patterns with a fixed size, that achieve 0.1% for patterns with four courses and four operations per course (4x4) and only 0.002% for 5x5 patterns. Both sizes are still very small. Bigger full random patterns could not be found in a reasonable amount of time.

6. Design Tool

In order to provide easy accessibility of the model for knit designers and process engineers, we integrate the model into a web-based design tool, which allows the generation of patterns and the assembly to bigger swatches. Figure 7 shows the making of a swatch using our design tool. The user interface divides the design process into three steps: Project setup, pattern selection and assembly. The project setup step initially allows to set the size of the targeted swatch or garment. Then, the pattern selection step (figure 7a) is used to generate novel patterns using the previously trained model. Decoding parameters can be set in the panel on the top right of the screen. The style can be varied continuously, by providing the amount of style as a percentage of each category. The decoding method can be selected and corresponding parameters like the temperature can be set. Patterns appear on the left side as they are generated. An image of each pattern can be rendered using the proprietary software of the machine manufacturer in the background (figure 7b). Appealing patterns can be marked as a favorite, which stores them in the pattern library shown on the bottom right of the screen. The knittability check is performed for each of the generated patterns and visualized using an icon. Patterns with problems can optionally be hidden. The assembly section allows to combine the previously selected patterns to a larger swatch or garment. In the beginning, a blank canvas with a size according to the project setup is shown. Different tools can be used to draw and place patterns on the canvas: The texture tool allows to stamp patterns on the canvas, repeat and mirror them or swap front and back colors. The selection tool allows to fill areas with patterns or create new ones from the canvas. The pen and color picker tool allow a manual correction. Our design tool allows an import and export of the proprietary machine format for patterns and the assembly. In addition, an import and export of images allows the integration with other editing software.

Multiple patterns were generated with different styles and decoding configurations. They were modified manually to clean them up or make them more interesting. The originally generated patterns and the edited variation are both shown in the pattern library in figure 7c. The logo on the top was prepared in an image editing software and imported as a pattern. The completed assembly was exported to the proprietary file format and then converted to machine instructions. It took around 30 minutes to knit on the machine.

7. Discussion

Our code generation approach aims to support the very specific problem of pattern creation, which makes it only a small part of the apparel design process. Our model is able to greatly mimic the appearance of the training data, while not exactly replicating it. When using a suitable decoding method, the style input influences the generated patterns properly and therefore allows the specification of a desired look. We showed some knitted patterns to experts and they confirmed that the generated patterns look like typical patterns found in the libraries. While our model allows users to specify the rough style they desire, there is no explicit way to specify the overall arrangement within a pattern. This limitation leaves them to just generate many different ones and hope to find one that matches their intention. Nevertheless, it allows to choose patterns, without needing to know about low-level machine instructions.

The generated code is not guaranteed to be free of errors, but the knit check allows to find and locate problematic patterns automatically. It is oftentimes still possible to generate a preview of problematic patterns. If the appearance matches the intention of a user but the code is problematic, the correction still needs to be done manually.

Our design tool makes the model accessible and provides basic tools to assemble the generated patterns to bigger swatches or garments. The combination of model and design tool makes our work different to previous approaches, that either lack novelty in their output or require knowledge about knit programming in their input. Our tool allows to generate novel patterns and pick them visually, without the need to understand the underlying code. The generated code can be used in other software for further processing. However, the assembly of the generated patterns is still a manual process, which requires at least some knowledge about knit programming.

8. Future Work

Numerous research has been done on other sequence tasks, especially in the context of language modeling. Therefore, a next step is to apply previous research on language models to knitting, to further support the apparel design process: Since knitting patterns can be expressed in different styles, a style transfer model could be trained on a curated dataset, similar to a language translation model [24]. Another direction is to use our knit check in reverse to create an artificial dataset with incorrect patterns and corrections, that could be the input to a grammar correction model [25] that learns to correct any invalid pattern.

Our work introduced metrics for *knittability*, a necessary condition for a pattern to be useful at all, and *uniqueness*, describing the diversity of the output. The parameters of our model result in a tradeoff between those two quantities, which future models could try to decouple. To better reflect our actual goal, metrics for *creativity* and *novelty* are needed. Future work needs to formally describe those quantities in general and especially in the context of knitting. This will allow the training and evaluation of more sophisticated models and methods like generative adversarial networks (GANs), transformer networks and reinforcement learning. In addition to knittability, these could also be used to formulate a more suited loss function for the model training.

9. Conclusion

We have shown, that recurrent neural networks can be used to generate knitting code, that is syntactically correct, knittable and visually appealing. The generated patterns indicate, that knitting code can in fact be represented by sequences of instructions, without sacrificing two-dimensional context and alignment. As a result, the application of known text-generation techniques to knitting code is a promising research direction. The presented design tool makes the trained model accessible and easy to use for designers. The knitting style can be specified by familiar categories and appealing patterns can be selected. The build-in knit check allows to quickly sort out patterns that are difficult or impossible to produce. Import and export options integrate our tool seamlessly into the apparel design process.

References

[1] J. McCann, L. Albaugh, V. Narayanan, A. Grow, W. Matusik, J. Mankoff, J. Hodgins, A compiler for 3D machine knitting, ACM Transactions on Graphics (TOG) 35 (2016) 1–11.
[2] A. Kaspar, L. Makatura, W. Matusik, Knitting skeletons: A computer-aided design tool for shaping and patterning of knitted garments, 2019.
[3] V. Narayanan, L. Albaugh, J. Hodgins, S. Coros, J. McCann, Automatic machine knitting of 3D meshes, ACM Transactions on Graphics (TOG) 37 (2018) 1–15.
[4] K. Wu, X. Gao, Z. Ferguson, D. Panozzo, C. Yuksel, Stitch meshing, ACM Transactions on Graphics (TOG) 37 (2018) 1–14.
[5] K. Wu, H. Swan, C. Yuksel, Knittable stitch meshes, ACM Transactions on Graphics (TOG) 38 (2019) 1–13.
[6] A. Karmon, Y. Sterman, T. Shaked, E. Sheffer, S. Nir, KNITIT: a computational tool for design, simulation, and fabrication of multiple structured knits, in: Proceedings of the 2nd ACM Symposium on Computational Fabrication, 2018, pp. 1–10.
[7] A. Kaspar, T.-H. Oh, L. Makatura, P. Kellnhofer, J. Aslarus, W. Matusik, Neural inverse knitting: From images to manufacturing instructions, arXiv preprint arXiv:1902.02752 (2019).
[8] M. Hofmann, L. Albaugh, T. Sethapakadi, J. Hodgins, S. E. Hudson, J. McCann, J. Mankoff, Knitpicking textures: Programming and modifying complex knitted textures for machine and hand knitting, 2019.
[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, OpenAI Blog 1 (2019) 9.
[10] T. Gries, D. Veit, B. Wulfhorst, V. Schrank, A. Hehl, K. P. Weber, Processes and Machines for Knitwear Production, Carl Hanser Verlag GmbH & Co. KG, 2014, pp. 173–194.
[11] Shima Seiki, SDS-ONE APEX3 Flat knitting, 2019. URL: http://www.shimaseiki.com/product/design/sdsone_apex/flat/, accessed 29 December 2019.
[12] H. Stoll AG & Co. KG, Pattern Software M1Plus®, 2019. URL: https://www.stoll.com/en/software/m1plusr/.
[13] J. Underwood, The design of 3D shape knitted preforms, 2009. PhD thesis.
[14] M. Toyoura, T. Igarashi, X. Mao, Generating jacquard fabric pattern with visual impressions, IEEE Transactions on Industrial Informatics 15 (2018) 4536–4544.
[15] I. Sutskever, J. Martens, G. E. Hinton, Generating text with recurrent neural networks, in: Proceedings of the 28th international conference on machine learning (ICML-11), 2011, pp. 1017–1024.
[16] O. Vinyals, A. Toshev, S. Bengio, D. Erhan, Show and tell: A neural image caption generator, 2015.
[17] P. Yin, G. Neubig, A syntactic neural model for general-purpose code generation, 2017.
[18] T. Beltramelli, pix2code: Generating code from a graphical user interface screenshot, 2018.
[19] A. Karpathy, The Unreasonable Effectiveness of Recurrent Neural Networks, 2015. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/, accessed 20 March 2019.
[20] K. Choi, G. Fazekas, M. Sandler, Text-based LSTM networks for automatic music composition, arXiv preprint arXiv:1604.05358 (2016).
[21] J. Shane, SkyKnit: When knitters teamed up with a neural network (2018). URL: https://aiweirdness.com/post/173096796277/skyknit-when-knitters-teamed-up-with-a-neural.
[22] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (1997) 1735–1780.
[23] Z. Xie, Neural text generation: A practical guide, arXiv preprint arXiv:1711.09534 (2017).
[24] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, arXiv preprint arXiv:1409.0473 (2014).
[25] O. Vinyals, Ł. Kaiser, T. Koo, S. Petrov, I. Sutskever, G. Hinton, Grammar as a foreign language, in: Advances in neural information processing systems, 2015, pp. 2773–2781.