Tampere University

Eetu Tunturi

USER ACTION PREDICTION WITH NEURAL NETWORKS

Bachelor's Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Khazar Khorrami
May 2023

ABSTRACT

Eetu Tunturi: User action prediction with neural networks
Bachelor's Thesis
Tampere University
Bachelor's Degree Programme in Information Technology
May 2023

Artificial intelligence and machine learning have been used to solve a wide variety of complex problems in different fields. In this study, machine learning will be used to predict user actions in software applications. The ability to predict user actions would enable the creation of dynamic user interfaces. Dynamic user interfaces can be used to improve the usability of software applications. Usability means that the user can achieve their goals more effectively and efficiently. Additionally, good usability helps new users to learn to use the application faster.

This thesis aimed to study if neural networks can learn patterns in software application usage. Software application usage can be presented as a sequence of individual actions performed by the user. Neural networks were used to predict the user's actions based on their previous actions. Additionally, the performance of different neural network architectures was compared. First, a small review of previous research into user action prediction is provided. In the experimental part of this thesis, neural networks were trained to predict the next element in the sequence of actions. The neural networks were trained using data from Vertex G4, a computer-aided design (CAD) application. The experiments use data from a desktop application with a graphical user interface, but the results also generalize to other kinds of user interfaces, like mobile applications and command-line applications. Three different neural network architectures were experimented with: long short-term memory (LSTM), gated recurrent unit (GRU), and transformer.

The results of this study show that neural networks can be used to predict user actions in certain kinds of software applications. All of the studied architectures achieve similar performance metrics, which shows that they are all valid solutions for predicting user actions. However, the models trained in this study leave a lot of room for improvement. As such, to conclude this study, some ideas for improving the models are presented.

Keywords: neural networks, usability, user action prediction

TIIVISTELMÄ

Eetu Tunturi: Käyttäjän toimintojen ennustaminen neuroverkoilla
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Toukokuu 2023

Tekoälyä ja koneoppimista käyttäen on ratkaistu monenlaisia ongelmia erilaisilla aloilla. Tässä tutkimuksessa koneoppimista käytetään käyttäjän toimintojen ennustamiseen työpöytäsovelluksessa. Kyky ennustaa käyttäjän toimintoja mahdollistaa dynaamisten käyttöliittymien kehittämisen. Dynaamisilla käyttöliittymillä voidaan parantaa sovellusten käytettävyyttä. Käytettävyys tarkoittaa, että käyttäjä pystyy saavuttamaan tavoitteensa täydellisemmin ja tehokkaammin. Lisäksi hyvä käytettävyys auttaa uutta käyttäjää oppimaan sovelluksen käytön nopeammin.

Tämän työn tarkoituksena on tutkia voivatko neuroverkot oppia sovellusten käytön rakennetta. Sovelluksen käyttö voidaan esittää sarjana, joka koostuu yksittäisistä käyttäjän suorittamista toiminnoista. Neuroverkkoja käytettiin ennustamaan käyttäjän suorittamia toimintoja hänen aiempien toimintojen perusteella. Lisäksi eri neuroverkkoarkkitehtuurien soveltuvuutta käyttäjän toimintojen ennustamiseen vertailtiin. Ensin tehtiin lyhyt katsaus aiempiin tutkimuksiin käyttäjän toimintojen ennustamisesta. Tämän tutkimuksen kokeellisessa osiossa koulutettiin neuroverkko-malleja ennustamaan seuraava toiminto sarjassa toimintoja. Mallit koulutettiin käyttäen dataa Vertex Systemsin tuottamasta Vertex G4 -sovelluksesta. Tutkimus kohdistuu graafisen käyttöliittymän omaavaan työpöytäsovellukseen, mutta tulokset yleistyvät myös muunkaltaisiin käyttöliittymiin, kuten mobiili- tai komentorivisovelluksiin. Tässä työssä tutkittiin kolmea eri neuroverkkoarkkitehtuuria: pitkäkestoinen lyhytkestomuisi (engl. long short-term memory, lyh. LSTM), gated recurrent unit (GRU) ja transformer.

Tämän tutkimuksen tulokset osoittavat, että neuroverkot voivat ennustaa käyttäjien toimintoja tietynlaisissa sovelluksissa. Kaikki kolme tutkittua neuroverkkoarkkitehtuuria soveltuvat käyttäjän toimintojen ennustamiseen. Työn lopuksi ehdotetaan menetelmiä, joilla tässä työssä käytettyjä malleja voitaisiin jatkossa kehittää parempien tulosten saavuttamiseksi.

Avainsanat: neuroverkot, käytettävyys, käyttäjän toimintojen ennustaminen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial intelligence
CAD	Computer-aided design
CEC	Constant Error Carousel
GRU	Gated recurrent unit
LSTM	Long short-term memory
ML	Machine learning
MLP	Multi-layer perceptron
RNN	Recurrent neural network

1. INTRODUCTION

Bad usability is a common problem in software applications. Bad usability means that users either cannot achieve their goals in using an application or achieving their goals may be inefficient. The negative effects of bad usability grow especially as the complexity of software grows. Complex software is software that has a large number of unique operations which may be used in many different use cases. The most common way to display these masses of operations to a user is a complicated graphical user interface. However, it is slow to search for different operations in a graphical menu, especially for a new user. Often keyboard shortcuts are provided, so that actions may be performed quickly. However, most users only use a small number of shortcuts. Keyboard shortcut usage depends on a multitude of factors, such as the amount of experience the user has working with computers, and the shortcut usage of other people in their working environment. [1] One solution to the problems of complicated menus and keyboard shortcuts is to create a small graphical menu that dynamically recommends the user commands based on their previous actions.

This study aims to test user action prediction methods that enable the development of a dynamic action recommendation system. Different neural network models are trained on historical user data. Each model's input will be a sequence of user actions and the goal of the models will be to predict the next user action based on the observed input sequence.

This thesis is structured as follows. Chapter 2 will provide some background based on existing research. Chapter 3 consists of theory about neural networks both in general and of the specific types used in this study. Chapter 4 describes the data, the experiment, and the results. Chapter 5 will conclude this study with a summary of the results and discussion about possible further research.

2. BACKGROUND

Machine learning (ML) is a subset of artificial intelligence (AI) that aims to build systems that imitate human learning. In a supervised ML system, the system is trained to perform a task using a set of training data, and it learns a solution that generalizes to other similar data. ML has been one of the biggest trends in technology over the past decade because the recent availability of large datasets and computing power has enabled ML to improve fast.

Modern software applications collect lots of data from users. The user data can be leveraged to improve existing user interfaces and to create entirely new ones using ML. Voice assistants, chatbots, and content recommender systems are some of the most prominent applications of ML to user interfaces. Moreover, lots of ML algorithms have been proposed for intelligent user interfaces. Most prior research in predicting user actions is focused on Markov models. Markov models assume that only the last action affects the next one. Davison and Hirsh predict sequences of UNIX commands using a Markov model that predicts based on only the last known command [2].

However, modern software applications are far too complex to be predictable using first-order Markov models. Using more than just one action improves prediction accuracy significantly. Therefore, higher-order Markov models were proposed. Hartmann and Schreiber present the FxL algorithm, which uses weighted Markov models of different orders. [3] Higher-order Markov models, such as FxL, use more than just one action. However, they are still limited to a small fixed-size input.

Adam et al. use LSTMs to predict user actions in medical applications. LSTMs are a type of neural network that is capable of learning long temporal dependencies. Adam et al. find that the ability to learn longer temporal dependencies outperforms the limited input of higher-order Markov models. They make predictions using the 100 previous actions. However, their study is the only work that uses LSTMs or similar models to predict user actions. [4] It remains unclear how important long temporal dependencies are in user action prediction in general.

Besides just sequences of user actions, different kinds of information could also be used to predict the next action [5]. For example, content recommender systems often group similar users together and recommend them content based on what similar users are

doing instead of making recommendations based on all users. The similarity of users can be based on their usage history, location, age, or other relevant information. [6]

3. METHODOLOGY

Three different kinds of neural networks were used in this study: long short-term memory (LSTM), gated recurrent unit (GRU), and transformer networks. These three models were chosen because they work well with sequential data. Additionally to these three models, also the multi-layer perceptron (MLP) will be introduced because it forms the basic structure of a neural network that the three used models are based on.

3.1 Multi-layer perceptron

MLPs are simple neural networks that form the basic structure that many other artificial neural networks follow. MLPs consist of an input layer, one or more hidden layers, and an output layer. In an MLP, all neurons of each layer are connected to all neurons of the next layer, which is why MLPs are also commonly known as feed-forward neural networks or fully connected neural networks. The basic structure of an MLP is illustrated in figure 3.1. The first layer is the input layer. The shape of the input layer depends on the shape of the data. The input layer does not have any activation function. After the input layer, there can be any number of hidden layers, where each layer can have a non-linear activation function. The network in Figure 3.1 has one hidden layer. The last layer is the output layer, where the neurons also have activation functions. Neurons in one layer have the same activation function, but different layers may have different activation functions. [7]

The input of a neuron in a hidden layer or the output layer consists of the outputs of all neurons of the previous layer and a constant bias term. The neuron computes a weighted sum of all of its inputs and then calculates the activation function of the sum. The linear combination is calculated as

$$s_i = \sum_{j=0}^{J} w_j x_j, \tag{3.1}$$

where J is the number of neurons in the previous layer, w_j is the weight for the jth input and x_j is the jth input. The bias term is included in the sum by setting $x_0 = 1$.

The non-linear activation functions are what enable the neural network to learn complicated patterns. For hidden layers, the most commonly used activation function is called

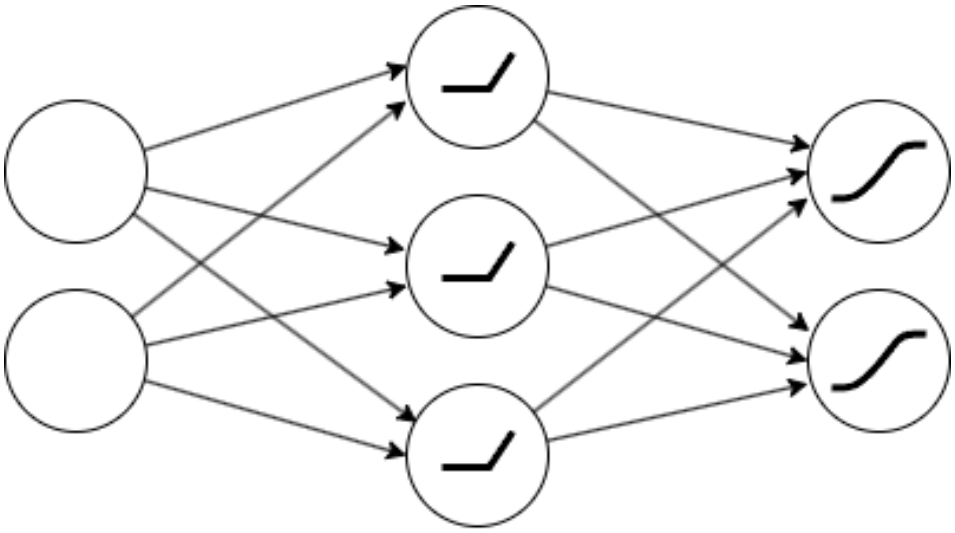

Figure 3.1. *Diagram of a simple multi-layer perceptron.*

rectified linear unit (ReLU). ReLU sets all negative values to zero while keeping the positive values:

$$f_i(s_i) = \max(0, s_i) = \begin{cases} s_i & s_i > 0, \\ 0 & \text{otherwise} \end{cases} . \tag{3.2}$$

Softmax is commonly used as the activation function of the output layer when the target consists of discrete data with C classes because it scales the output into a probability distribution. Softmax is calculated as [7]

$$f_i(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{C} e^{s_j}}. \tag{3.3}$$

Neural networks are evaluated in two phases during training. The outputs of the network are calculated in the forward pass. The network's weights are adjusted in the backward pass. A loss function is chosen that will indicate the amount of difference between the output of the network and a correct output that is known beforehand (ground truth). A common loss function for classification tasks is cross-entropy.

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i), \tag{3.4}$$

where C is the total number of classes, y_i (0 or 1) is the ground truth value for class i and $\hat{y}_i$ is the network's output for class i. The loss function is differentiated with respect to the weights using the backpropagation algorithm. In order for backpropagation to work, activation functions have to be differentiable. An optimization algorithm optimizes the weights

using the gradients obtained from backpropagation. A simple algorithm for updating the weights would be to use a constant learning rate. The new weights would be:

$$W^{t+1} = W^t - \alpha \nabla L, \tag{3.5}$$

where α is the learning rate and ∇L is the gradient of the loss with respect to W. The learning rate is a parameter that defines how much the weights are adjusted each iteration. Using a constant learning rate can work, but in practice, better results are achieved by gradually lowering the learning rate during training. [7]

There are many alternative optimization algorithms that are commonly used. Adam is an optimization algorithm that uses different learning rates for different parameters and adjusts the learning rates during training [8].

3.2 Recurrent neural networks

Recurrent neural networks (RNN) expand on the feed-forward structure of the MLP by adding cyclic connections to the neurons. Cyclic connections mean that a layer can have connections to previous layers or itself. The cyclical structure allows RNNs to learn temporal dependencies, which means they work well with sequential data. RNNs take a sequence of vectors as input and output a sequence of vectors. The length of the input sequence to an RNN can be variable in the temporal dimension, unlike the input of the MLP, which is fixed in size. The variable input size can be thought of as sequentially inputting one vector at a time to the network. The network maintains a hidden state that remembers information from its past inputs, which is what enables RNNs to learn temporal dependencies. RNNs can be unfolded into a computational graph that illustrates the sequential processing, like in figure 3.2.

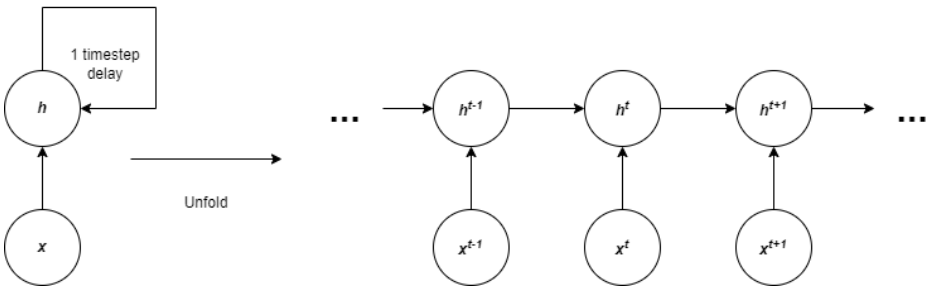

Figure 3.2. *Diagram of an unfolded RNN. x_t denotes the input at timestep t and h_t denotes the hidden state at timestep t. All timesteps use the same weights. Adapted from [7]*

RNNs often suffer from two problems with gradients, known as the exploding gradient problem and the vanishing gradient problem. The cyclic structure means that RNNs are usually very deep, which means that they have lots of layers. Due to how backpropagation

works, the gradients in the first layer are the product of the gradients of all following layers, which makes the gradient unstable. Unstable gradients often approach zero or grow uncontrollably. Small gradients are problematic because they cause learning to be too slow or nonexistent. Large gradients are problematic because the gradients only depict the behavior of the loss function in a small neighborhood. Large gradients may end up in places where the gradient does not describe the behavior of the function anymore. In theory, RNNs are able to learn long-term dependencies, but because of these gradient problems, they can rarely do it in practice. [7] In this section two solutions to the gradient problems will be presented. Long short-term memory (LSTM) was the first successful solution to the gradient problems that worked well with long sequences [9]. The gated recurrent unit (GRU) is a variation of the LSTM that uses a simpler structure [10].

3.2.1 Long short-term memory

LSTMs are a kind of recurrent neural network that was developed to solve the vanishing and exploding gradient problems. LSTMs have achieved good results in many applications, like natural language processing, translation, and speech recognition. [9] [7]

Hochreiter & Schmidhuber proposed a structure known as a memory cell to replace a regular neuron. Each cell has a single recurrent connection to itself with a constant weight of 1.0, known as a constant error carousel (CEC) neuron. Memory cells also contain two gates that aim to discard irrelevant information, the input gate and the output gate. The input gate discards irrelevant information that is being input into the cell. The output gate prevents irrelevant information from ending up in the output of a cell, which could then be taken as input by other cells. [9] The cell structure has seen many variations since the proposal of the original LSTM. Gers et al. introduced the forget gate that replaces the constant weight of the CEC neuron. The forget gate aims to discard irrelevant information from the recurrent connection of the cell to itself. [11]

Figure 3.3 contains a diagram of a single LSTM cell. The gates are represented by the nodes with sigmoids in them. Each sigmoid and tanh node contains a linear combination with learned weights and a non-linear activation function, like a neuron in an MLP. The output of the Input gate is

$$g_t = \sigma(W_g h_{t-1} + U_g x_t + b_g), \tag{3.6}$$

where σ is the logistic sigmoid function, h_{t-1} is the hidden state, x_t is the input of the cell, W_g are the weights of the recurrent connection, U_g are the weights of the input and b_i is the bias of the input gate. The output of the forget gate is

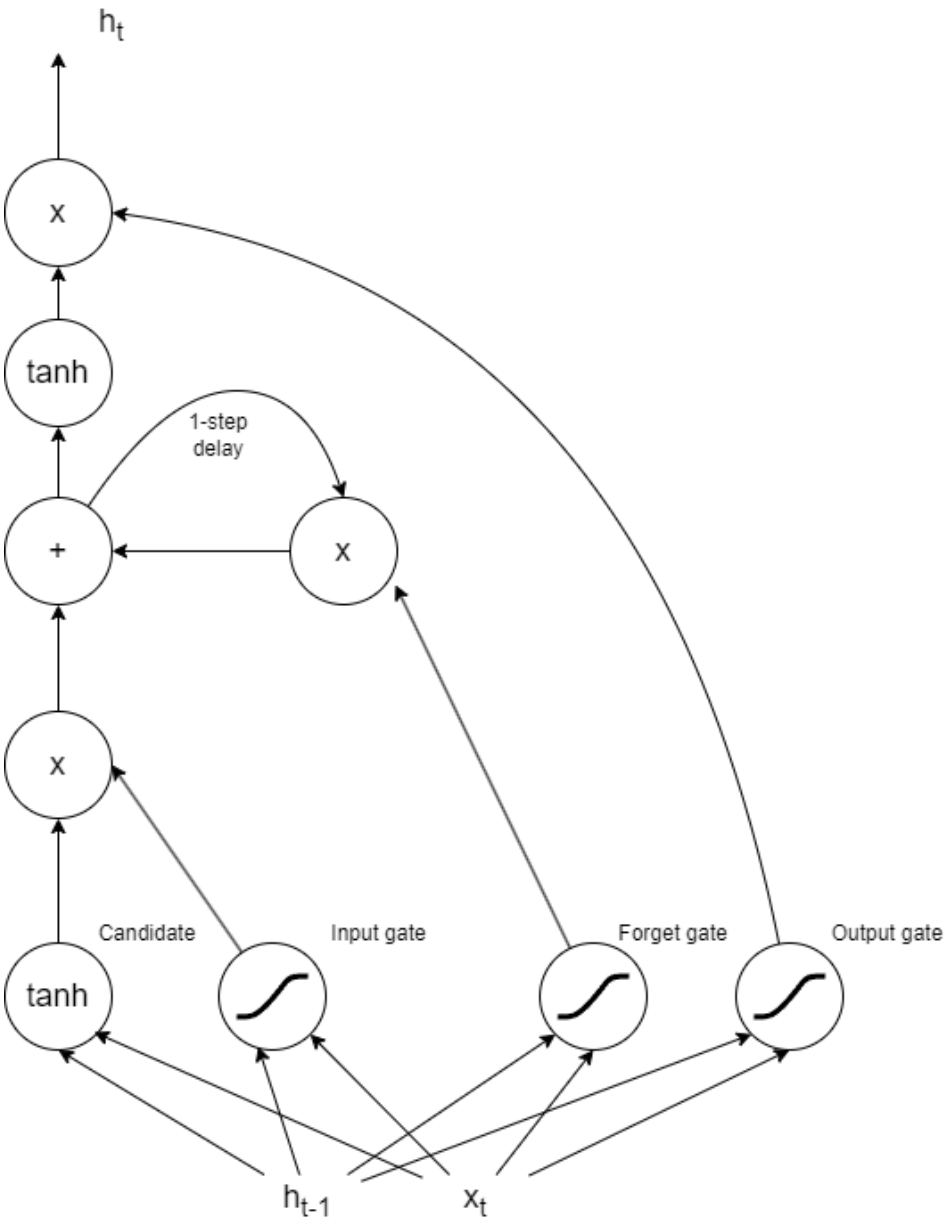

Figure 3.3. *Diagram of an lstm memory cell. Adapted from [7]*

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f), \tag{3.7}$$

where W_f, U_g, and b_f are the recurrent weights, input weights, and the bias of the forget gate. The output of the output gate is

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o), \tag{3.8}$$

where W_o, U_o, and b_o are the recurrent weights, input weights, and bias of the output gate. The input gate decides which values in the cell state will be updated. The tanh layer proposes candidate values that will replace the old ones if the input gate allows it. The

output of the tanh layer is

$$C_t = \tanh\left(W_C h_{t-1} + U_C x_t + b_C\right), \tag{3.9}$$

where W_C, U_C, and b_C are the recurrent weights, input weights, and bias of the candidate layer, and tanh is the hyperbolic tangent function. The cell state is updated as

$$S_t = C_t \cdot g_t + f_t \cdot S_{t-1}, \tag{3.10}$$

where S_{t-1} is the cell state of the previous time step. Finally, the output value h_t is obtained from

$$h_t = o_t \cdot \tanh(S_t). \tag{3.11}$$

3.2.2 Gated Recurrent Unit

GRU is a variant of the LSTM proposed by Cho et al. in 2014. It combines the input and output gates of the LSTM into one gate, the update gate, and merges the cell state and the hidden state. [10] With the reduced amount of separate parts within each cell, GRUs are much simpler than LSTMs. GRUs have been found to be easier and less computationally expensive to train while achieving equal results in small datasets. However, LSTMs tend to outperform GRUs in large datasets. [12] [13] [14]

Figure 3.4 contains a diagram of a single GRU cell. The output of the reset gate is

$$R_t = \sigma(W_R h_{t-1} + U_R x_t + b_R), \tag{3.12}$$

where W_R, U_R, and b_R are the recurrent weights, input weights, and the bias of the reset gate. The output of the update gate is

$$u_t = \sigma(W_U h_{t-1} + U_U x_t + b_U), \tag{3.13}$$

where W_U, U_U, and b_U are the recurrent weights, input weights, and the bias of the update gate. The output of the cell h_t is

$$h_t = \tanh(W_t x_t + R_t U_t h_{t-1} + b_t) \cdot u_t + (1 - u_t) \cdot h_{t-1}, \tag{3.14}$$

where W_t, U_t, and b_t are the recurrent weights, input weights, and the bias of the tanh layer.

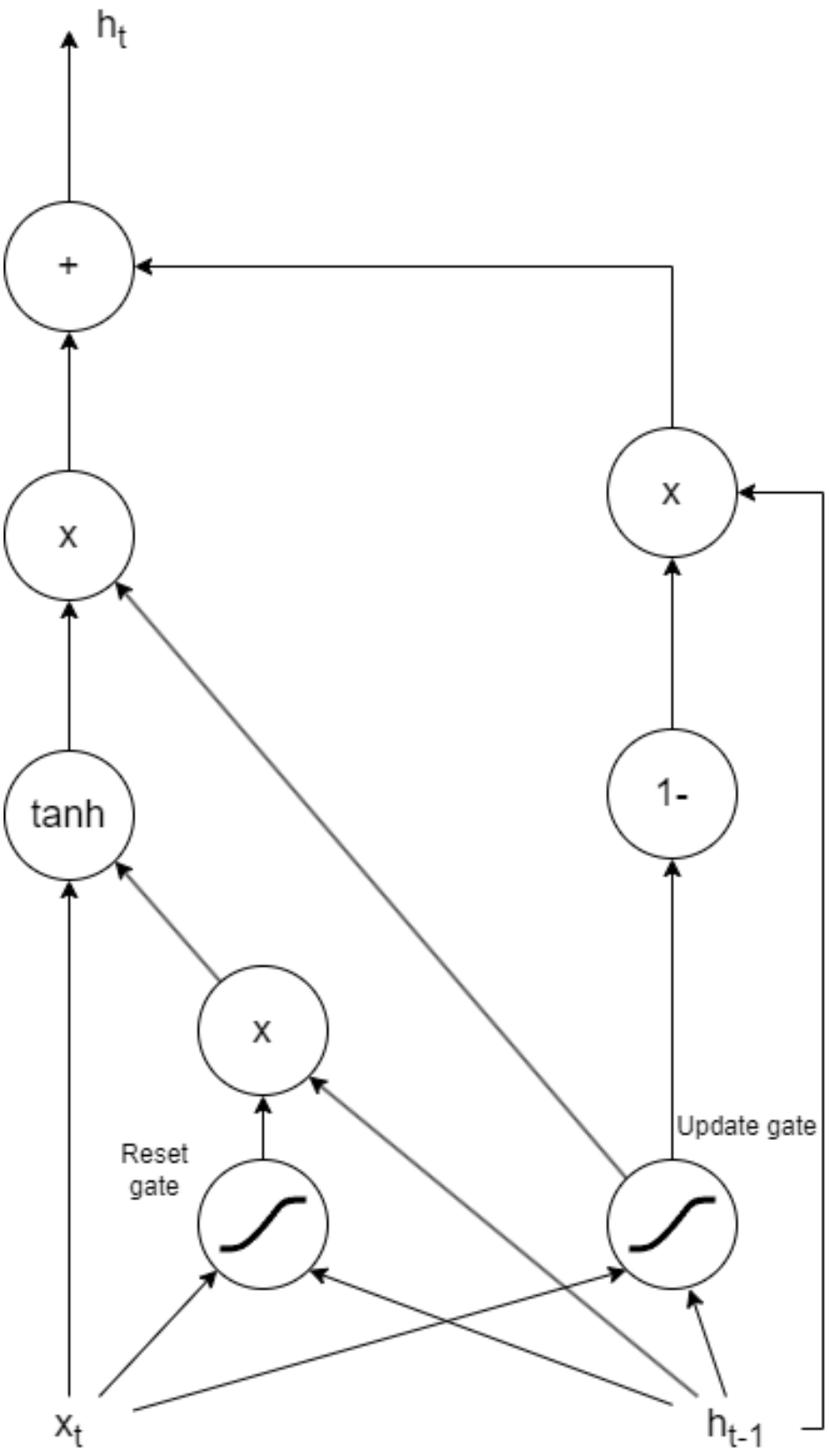

Figure 3.4. *Diagram of a memory cell in a GRU architecture*

3.3 Transformer

Transformers are machine learning models that use an attention mechanism to process sequential data. Transformers were developed by Vaswani et al. for natural language

processing tasks, such as translation. The attention mechanism allows the transformer to process the entire input sequence at once. RNNs have to process all timesteps sequentially, which can be seen for example in Equation 3.9, where the new state requires the previous state to be known. Therefore, it is hard to parallelize the calculation of RNNs. Transformers are easier to parallelize because the attention mechanism can access all timesteps at once. [15] The parallelization advantage is not the main motivation behind the use of transformers, as transformers have also been outperforming RNNs in many applications. [16] [17]

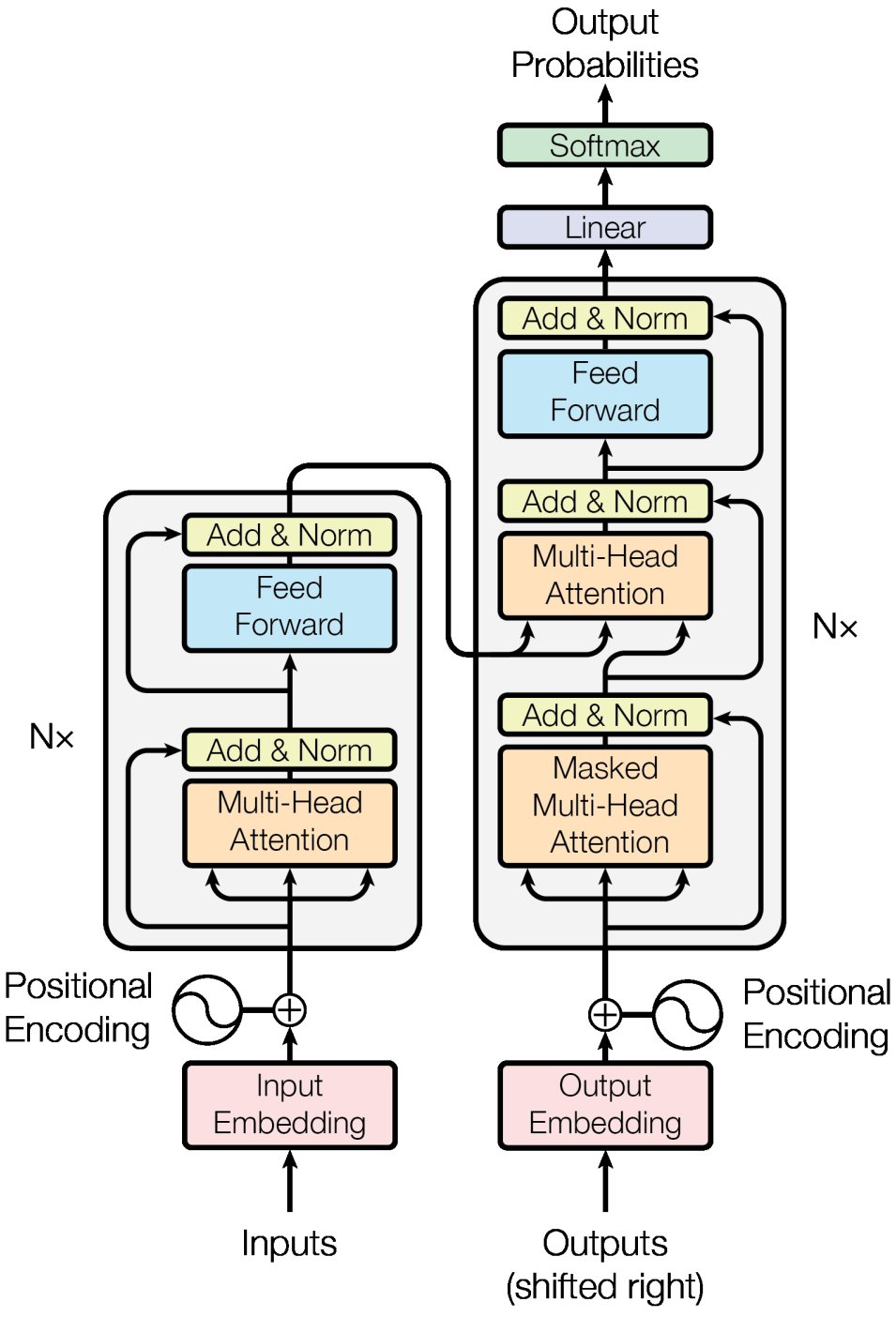

Figure 3.5. *The transformer architecture, reproduced from [15]*

A diagram of the transformer architecture can be seen in Figure 3.5. The full architecture has an encoder-decoder structure. In the figure, the encoder part is the left half and the decoder is the right half. [15] In many transformer models, only the encoder part is used. Inputs to the encoder are preprocessed using learned embeddings. Embeddings transform the classes that are represented as unique tokens into dense fixed-size vectors. The encoder takes the embedded inputs and adds positional encoding to them. Positional encoding adds information about the positions of the input elements in the sequence. The positional encoding is followed by N identical encoder layers. Each layer begins with multi-head self-attention, which is summed to its own input and normalized using layer normalization. The normalization is followed by a simple MLP, the output of which is again summed to its input and normalized. [15]

Attention is a mechanism that emphasizes relevant parts of the input. The input of an attention layer consists of queries and key-value pairs. In self-attention, which is what the encoder part uses, the queries, keys, and values are all a copy of the input vector. The output of an attention layer is a weighted sum of the values. The weight of a value is calculated using the query and the key that correspond to that value. There are different kinds of attention functions. The attention function used in the transformer architecture is known as "Scaled Dot-Product Attention" and is calculated as

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V, \tag{3.15}$$

where Q are queries, K are keys, V are values and d_k is the dimensionality of the keys. [15]

Multi-head attention uses multiple parallel attention layers instead of just one. The inputs to the different heads are created by using learned linear projections on the inputs of the multi-head attention layer.

$$\text{MultiHead}(Q, K, V) = \text{Concatenate}(\text{head}_1, ..., \text{head}_h)W^O, \tag{3.16}$$

where h is the number of attention heads and W^O is a learned linear projection that projects the concatenated outputs of the individual heads into the dimensionality of the output of the multi-head attention. The individual heads are calculated as

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \tag{3.17}$$

where W_i^Q, W_i^K, and W_i^V are learned linear projections of the queries, keys, and values respectively. [15]

4. EXPERIMENTS

In this study, I aim to train neural networks to predict user actions based on the user's previous actions. I train three different neural networks and evaluate their performance on unseen test data. Section 4.1 describes the dataset that was used in this study. Section 4.2 explains what kind of preprocessing I did to the dataset. Section 4.3 outlines the architectures of the models in detail. Section 4.4 explains how the models were trained. Section 4.5 explains the metrics that are used to evaluate the performance of the trained models. In section 4.6 the results of the experiments are presented.

4.1 Data

The dataset of this study has been provided to me by Vertex Systems [18]. The dataset contains information about user behavior in Vertex G4, which is a computer-aided design (CAD) application used for mechanical engineering. In this case, user behavior refers to sequences of actions performed by the users in the graphical user interface of the application.

The dataset consists of roughly 8.5 million actions. Actions are categorical data, where each category is identified by a unique string. In order to use the data to train neural networks, the dataset has to be processed into a format that conforms to the constraints that neural networks impose on their inputs and outputs.

4.2 Data processing

The dataset contains some actions that are not useful for predicting user behavior. For that reason, some types of actions were filtered out. The majority of what a user does in a CAD application consists of creating and modifying a 3D model of their product. Therefore, the majority of the most common actions relate to zooming, rotating, panning, and selecting parts of the graphical model. A small list of such actions was hand-picked and filtered out of the data. The assumption that some common actions are not helpful in predicting user behavior was tested as a part of the experiment. It was found that the filtering did not affect the observed performance metrics, which indicates that the actions that were left out did not contribute to the predictions of the models. Filtering out data

that the models ignore reduces the amount of data that the models require, which makes training the models much faster while having no effect on the measured performance of the models. After the filtering, there are 1397 unique actions in the dataset. Actions are categorical data, where each category is represented by a unique string. However, neural networks do not work with strings, they work with numerical data. The strings that represent the actions are tokenized into integers such that each string corresponds to one unique integer.

Initially, the dataset consists of sessions. A session consists of all actions from the time a user opens the application to the time they close it. Sessions are anywhere between 1 and 20 000 actions long. The sessions have to be processed into a dataset that contains pairs of inputs and labels. The inputs are the data that the model uses to make its predictions. Labels denote the correct prediction given the corresponding input. The purpose of the model is to take in a sequence of actions as input and predict what the next action will be. Recurrent neural networks work with variable input lengths, but the transformer network used in this study does not. Therefore, we must choose a constant length for the input and divide the sessions into sequences with that input length. For each input sequence, the correct label is the next action in the session. The way a session is processed into input sequences and labels is illustrated in Figure 4.1. If the session is so short that it does not contain a sequence that can be divided into an input of the chosen input length and a label, it is ignored. The sessions are divided into inputs and labels using a constant-sized sliding window. An input sequence and a label are chosen as described in the previous paragraph so that the input starts at the beginning of a session, which results in one input-label pair. The next pair is chosen so that the window is shifted forward by one timestep at a time until the end of the session.

Depending on the input length, there are between 2.05 million and 2.15 million pairs of inputs and labels. The number of samples varies because the input length changes the number of sequences that can be extracted from each session. The data was divided into training and validation data such that 80% of the data was used for training and the remaining 20% was used for validation, resulting in 1.64 to 1.72 million training samples and 0.41 to 0.43 million validation samples.

4.3 Models

Three neural network models with different architectures were experimented with. The architectures used were GRU, LSTM, and transformer. The output of each model is a probability distribution of all actions. To turn the probability distribution into a single action, which is the prediction we want, we choose the action with the highest predicted probability.

The LSTM and GRU models have the same structure, which is illustrated in Figure 4.2.

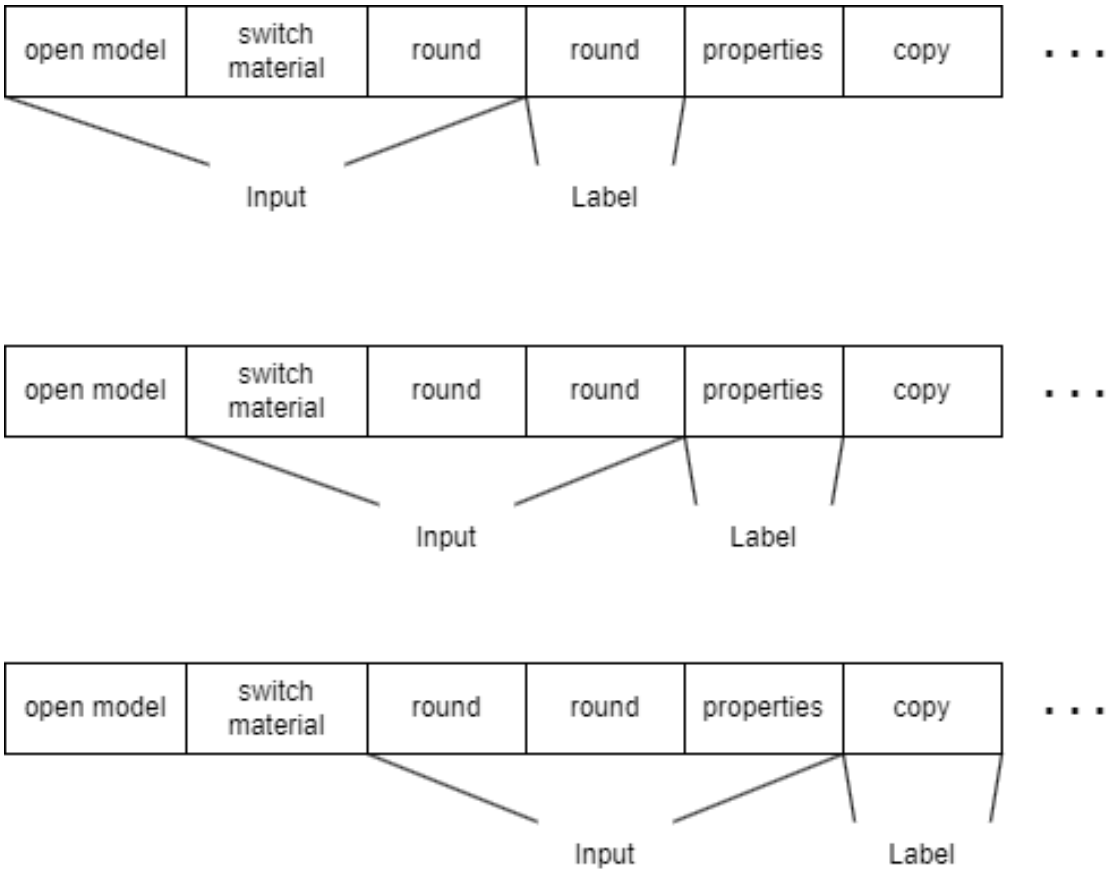

Figure 4.1. *Illustration of a sliding window splitting an example session into constant-sized sequences. In this example, the window size is 3 and the window is shifted by 1 step each iteration.*

The input of the model is a sequence of integers, that identify the actions. The model has an embedding layer that transforms the integers into dense fixed-size vectors. During the experiments I found that an embedding size of 512 worked the best with all three models. The embedding is learned by the model during training such that actions that are semantically similar are also similar in the embedding space. The embedding is followed by one recurrent layer, two feedforward layers with ReLU activation, and finally, the output layer, which uses softmax as its activation function.

The transformer model structure is illustrated in Figure 4.3. The transformer model has an embedding layer like the recurrent models. The embedding output is summed with the positional encoding, which is followed by a transformer encoder. The output of the transformer encoder has a two-dimensional shape (input length, embedding dimension). The output we want from the model is a one-dimensional probability distribution. The output of the encoder is made one-dimensional using an average pooling layer. Average pooling calculates the average of the encoder output along the temporal dimension, which creates a one-dimensional vector with the length of the embedding dimension.

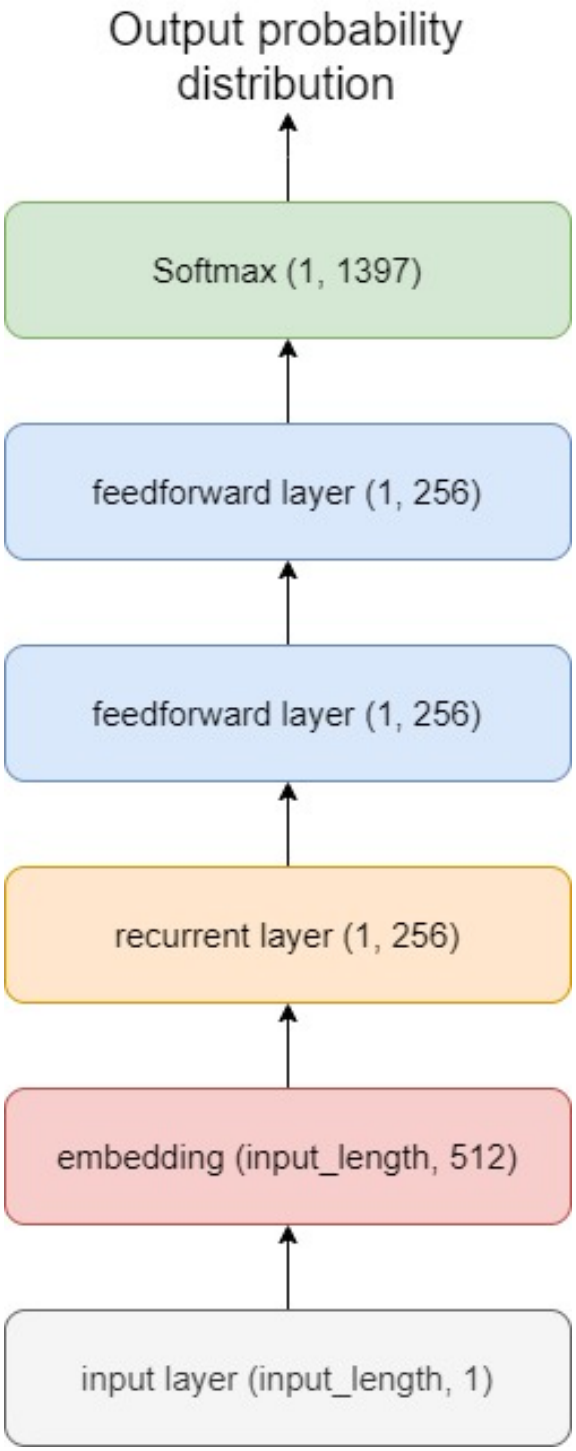

Figure 4.2. *The structure of the recurrent neural network models. The shape of the output of each layer is in parentheses.*

4.4 Training

All models were trained using the Keras API [19]. Keras is a high-level API for programming neural networks that uses TensorFlow [20] as its backend. Hyperparameter optimization was done manually without using any algorithms. All models were trained using cross-entropy loss and the Adam optimizer with a learning rate of 0.0001. To prevent

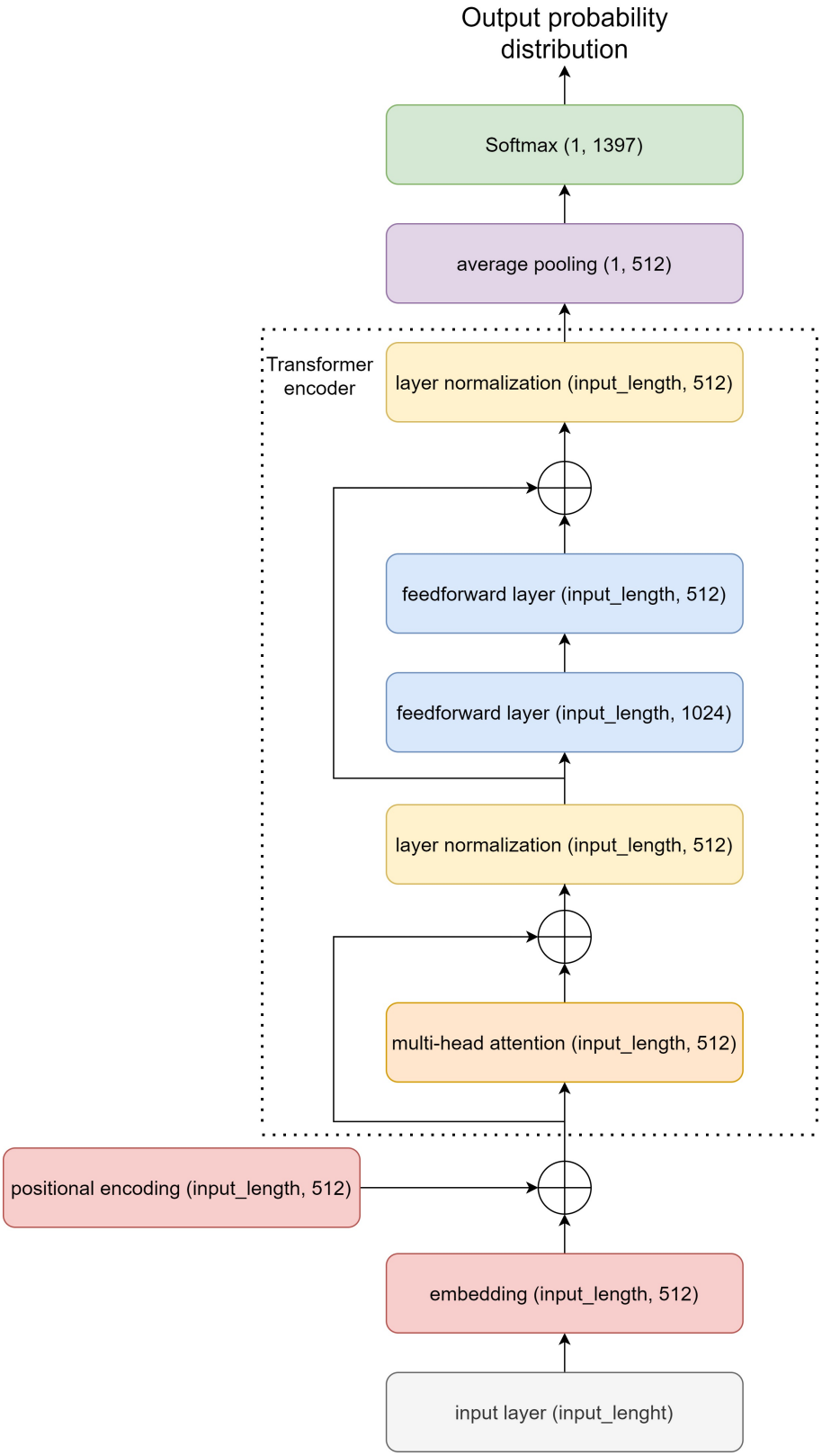

Figure 4.3. *The structure of the transformer model. The shape of the output of each layer is in parentheses.*

overfitting, the training of the models was stopped when the loss in the validation dataset did not decrease for 4 epochs.

In all three models, dropout was used to prevent overfitting. Dropout temporarily disables some randomly chosen neurons and their connections during training [21]. In the recurrent neural networks, each feedforward layer uses dropout with a rate of 0.35. The recurrent connections of the recurrent layers have a dropout rate of 0.20. In the transformer model, the two feedforward layers both have a dropout rate of 0.25.

4.5 Evaluation

Two metrics of performance will be measured, accuracy and top-5 accuracy. The output of the model is a probability distribution over all actions. Accuracy measures the number of predictions where the action with the highest predicted probability is correct. Top-5 accuracy measures the number of predictions where one of the 5 actions with the highest predicted probabilities is correct.

Top-5 accuracy was chosen because, in the application where the dataset was gathered from, the order of actions may vary a little and still produce the same result. The slight variation in the order of actions means that there can be multiple predictions that are sensible. Sometimes there may be many predictions that are good, but the accuracy metric only measures predictions that are exactly the same as the labels. Top-5 accuracy can measure some of the predictions where there are many good alternatives, which the simple accuracy metric cannot measure.

To gain some perspective on what can be considered a good accuracy, the results should be compared to a baseline. A simple baseline would be to always predict the most frequent action, or when measuring top-5 accuracy, the 5 most frequent actions. This kind of baseline approach would result in a 3.3% accuracy and a 12.4% top-5 accuracy.

4.6 Results

The results show in Figure 4.4. The results contain the accuracy and top-5 accuracy (Section 4.5), which were calculated using the validation data. Additionally to the three different architectures, the effects of the model's input length were also experimented with.

Compared to the baseline (see Table 4.1), the achieved results of 65% accuracy and 84% top-5 accuracy are good. The effect of input length on model performance is visualized in Figure 4.4. The performance of the model with an input length of 1 is significantly worse than the models with a longer input, but its significantly better than the simple baseline. An input length of 1 is not something that would ever actually be used in practice, but it serves as a point of reference for the models with a longer input sequence. The visualization clearly illustrates how increasing the input length yields diminishing increases in accuracy. The LSTM model performs worse with an input length of 200 than with an

architecture	input length	accuracy	top-5 accuracy
Baseline	-	0.0334	0.1247
GRU	1	0.5704	0.7414
GRU	5	0.6389	0.8032
GRU	10	0.6438	0.8120
GRU	25	0.6483	0.8209
GRU	100	0.6521	0.8268
GRU	200	0.6541	0.8277
LSTM	1	0.5684	0.7400
LSTM	5	0.6397	0.8049
LSTM	10	0.6435	0.8121
LSTM	25	0.6498	0.8232
LSTM	100	0.6524	0.8268
LSTM	200	0.6512	0.8280
Transformer	1	0.5662	0.7399
Transformer	5	0.6369	0.8090
Transformer	10	0.6408	0.8174
Transformer	25	0.6440	0.8297
Transformer	100	0.6459	0.8378
Transformer	200	0.6488	0.8416

Table 4.1. *Results of the experiments.*

input length of 100, probably due to overfitting. Models that use longer inputs require more processing power, which means that it is not feasible to increase the input length for small performance improvements. The training of models with an input length of 200 took roughly twice as long as the models with an input length of 25, which means that the computational cost of training doubled as well.

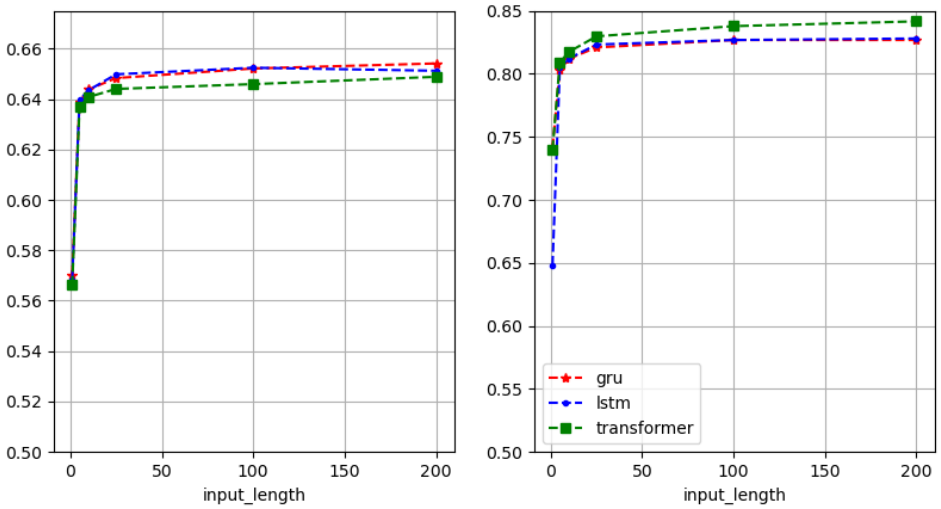

Figure 4.4. *Plot of the effect of input length on the evaluation metrics.*

5. CONCLUSION

This study aimed to predict user actions in software applications in a way that could be used to create a recommendation system that recommends relevant actions to users. I applied recurrent neural networks and transformers to the problem of user action prediction. The proposed methods are capable of correctly predicting a significant portion of the test data, but they leave a lot of room for improvement. It is also hard to estimate how good the performance metrics really are because it is unclear what kind of performance would be required to create something that provides significant user value. The effects of the length of the sequence of actions input to the models are small. Increasing the length of the input improves performance slightly, but longer inputs take more computational power to process, which should be taken into consideration when choosing the input length.

The studied models only use commands to make their predictions. Additional information about the state of the application could greatly improve performance. However, the availability of additional information is highly application specific. For example, a modal application could take into account the active mode in the predictions. Another possible way to improve performance would be to personalize the models for individual users. Personalizing could be done, for example, by first training a base model on data from all users and then further training the model with user feedback during regular use of the application.

REFERENCES

[1] Peres, S. C., Tamborello, F. P., Fleetwood, M. D., Chung, P. and Paige-Smith, D. L. Keyboard Shortcut Usage: The Roles of Social Factors and Computer Experience. eng. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 48.5 (2004), pp. 803–807. ISSN: 1071-1813.

[2] Davison, B. and Hirsh, H. Probabilistic Online Action Prediction. *AAAI* (1998), pp. 5–12.

[3] Hartmann, M. and Schreiber, D. Prediction Algorithms for User Actions. *15th Workshop on Adaptivity and User Modeling in Interactive Systems*. Ed. by I. Brunkhorst, D. Krause and W. Sitou. 2007.

[4] Adam, C., Aliotti, A., Malliaros, F. D. and Cournède, P.-H. Dynamic monitoring of software use with recurrent neural networks. *Data & Knowledge Engineering* 125 (2020), p. 101781. ISSN: 0169-023X. DOI: https://doi.org/10.1016/j.datak.2019.101781.

[5] Xu, Y., Lin, M., Lu, H., Cardone, G., Lane, N., Chen, Z., Campbell, A. and Choudhury, T. Preference, Context and Communities: A Multi-Faceted Approach to Predicting Smartphone App Usage Patterns. *Proceedings of the 2013 International Symposium on Wearable Computers*. ISWC '13. Zurich, Switzerland: Association for Computing Machinery, 2013, pp. 69–76. ISBN: 9781450321273. DOI: 10.1145/2493988.2494333.

[6] Narayanan, M. and Cherukuri, A. K. A study and analysis of recommendation systems for location-based social network (LBSN) with big data. eng. *IIMB management review* 28.1 (2016), pp. 25–30. ISSN: 0970-3896.

[7] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[8] Kingma, D. P. and Ba, J. *Adam: A Method for Stochastic Optimization*. English. 2017. DOI: https://doi.org/10.48550/arXiv.1412.6980.

[9] Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation* 9.8 (1997), p. 1735. ISSN: 08997667.

[10] Cho, K., Merrienboer, B. van, Gülçehre, Ç., Bougares, F., Schwenk, H. and Bengio, Y. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078.

[11] Gers, F. A., Schmidhuber, J. and Cummins, F. Learning to Forget: Continual Prediction with LSTM. *Neural Computation* 12.10 (2000), pp. 2451–2471. ISSN: 08997667.

[12] Yang, S., Yu, X. and Zhou, Y. LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example. *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*. 2020, pp. 98–101. DOI: 10.1109/IWECAI50956.2020.00027.

[13] Cahuantzi, R., Chen, X. and Güttel, S. *A comparison of LSTM and GRU networks for learning symbolic sequences*. 2023. arXiv: 2107.02248 [cs.LG].

[14] Mirzaei, S., Kang, J.-L. and Chu, K.-Y. A comparative study on long short-term memory and gated recurrent unit neural networks in fault diagnosis for chemical processes using visualization. eng. *Journal of the Taiwan Institute of Chemical Engineers* 130 (2022), pp. 104028–. ISSN: 1876-1070.

[15] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762.

[16] Karita, S., Chen, N., Hayashi, T., Hori, T., Inaguma, H., Jiang, Z., Someki, M., Soplin, N. E. Y., Yamamoto, R., Wang, X., Watanabe, S., Yoshimura, T. and Zhang, W. A Comparative Study on Transformer vs RNN in Speech Applications. *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, 2019. DOI: 10.1109/asru46091.2019.9003750.

[17] Lakew, S. M., Cettolo, M. and Federico, M. *A Comparison of Transformer and Recurrent Neural Networks on Multilingual Neural Machine Translation*. 2018. arXiv: 1806.06957 [cs.CL].

[18] *Vertex Systems Oy*. 2023. URL: https://vertexcad.com/.

[19] Chollet, F. et al. *Keras*. https://keras.io. 2015.

[20] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[21] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958.