TU WIEN Informatics

Neural Network Arena: Comparing Machine Learning Models using Long-Term Dependency and Physical System Time Series Benchmarks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Hannes Brantner, BSc
Matrikelnummer 01614466

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr.rer.nat. Radu Grosu, BSc
Mitwirkung: Dipl.-Ing. Dr. Ramin Hasani, BSc

Wien, 31. April 2021

Hannes Brantner	Radu Grosu

TU WIEN Informatics

Neural Network Arena: Comparing Machine Learning Models using Long-Term Dependency and Physical System Time Series Benchmarks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Hannes Brantner, BSc
Registration Number 01614466

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.rer.nat. Radu Grosu, BSc
Assistance: Dipl.-Ing. Dr. Ramin Hasani, BSc

Vienna, 31st April, 2021

Hannes Brantner | Radu Grosu

Declaration of Authorship

Hannes Brantner, BSc

I hereby declare that I have written this work independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 31st April, 2021

Hannes Brantner

Acknowledgements

d First, I have to thank Ramin for providing me excellent support throughout my work on the thesis. He cared about me and always pointed me to state-of-the-art literature, as he wanted to push me forward. I also have to thank Prof. Grosu for participating in numerous online meetings and sharing his in-depth knowledge in the machine learning domain. Furthermore, I want to thank Mathias Lechner for giving me first-class support on questions I had regarding various machine learning models. I have to expressly point out that he was always willing to help me and provided his responses incredibly fast. Finally, I have to thank my parents for providing me with mental and financial support throughout my whole study journey.

Kurzfassung

Die Vielfalt der Modelle für maschinelles Lernen hat in den letzten Jahren mit dem Aufblühen der Forschung in diesem Bereich rapide zugenommen. Diese Arbeit versucht einen Überblick über Modelle zu geben, die in der Lage sind mit regelmäßig abgetasteten Zeitreihendaten umzugehen, ohne eine vorgegebene Historienlänge zu spezifizieren, die vom Modell berücksichtigt werden soll. Daher sind alle in dieser Arbeit vorgestellten Modelle entweder Abkömmlinge des rekurrenten neuronalen Netzes oder der Transformer-Architektur [VSP+17]. Darüber hinaus wurden neue Modelle eingeführt, um die gegebene Architektur des Transformers [VSP+17] und des unitären rekurrenten neuronalen Netzes [JSD+17] zu verbessern. Nach der Einführung aller Modelle werden sie anhand fünf Benchmarks verglichen. Diese Benchmarks versuchen die Fähigkeit der Modelle zur Erfassung langfristiger Abhängigkeiten und die Fähigkeit der Modelle zur Modellierung physikalischer Systeme zu testen. Darüber hinaus wird eine zeitkontinuierliche Speicherzelle eingeführt, die in der Lage ist, ein Datenbit über eine große Anzahl von Zeitschritten zu speichern, ohne die gespeicherte Information zu verlieren. Diese Speicherzelle wird unter Verwendung der LTC-Network-Architektur [HLA+20] aufgebaut. Der gesamte für diese Arbeit verwendete Code ist unter `https://github.com/Oidlichtnwoada/NeuralNetworkArena` verfügbar.

Abstract

The diversity of machine learning models has rapidly increased in recent years as research in the machine learning domain flourishes. This thesis tries to give an overview of machine learning models that are capable of dealing with regularly sampled time series data without specifying a given history length that should be taken into account by the model. Therefore, all models presented in this thesis are either derivatives of the recurrent neural network or the Transformer [VSP+17] architecture. Furthermore, new machine learning models have been introduced to improve the given Transformer [VSP+17] and unitary recurrent neural network [JSD+17] architecture. After the introduction of all models, they are all benchmarked against five benchmarks and compared thoroughly. These benchmarks try to determine the model's capabilities to capture long-term dependencies and the ability to model physical systems. Moreover, the time-continuous Memory Cell architecture is introduced that is capable of storing a data bit over a large number of time steps without losing the stored information. This architecture is built using the LTC network [HLA+20] architecture. All code used for this thesis is available under `https://github.com/Oidlichtnwoada/NeuralNetworkArena`.

Contents

CHAPTER 1

Introduction

1.1 Machine Learning Terms

A machine learning model is a mathematical parametrized function that gets input and produces an output. For example, the machine learning model GPT-3 proposed in [BMR+20] has 175 billion scalar parameters. This thesis will use imitation learning to set the parameters of machine learning models optimally. Imitation learning means an associative expected output provided for each input that the model should return by applying its function to the input. Of course, when the model's function is applied to the input with the model's parameters' initial state, the returned model output will differ from the desired output in almost all cases. The measure responsible for quantifying this error between model output and the expected output is called a loss function and has a scalar return value. A sample loss function can be constructed quickly by computing the mean of all squared errors between the model output and the expected output. The model output is also often denoted as the prediction of the model. For each input sample, the loss function describes the error the model makes by applying its function, and this error is only dependent on the model's parameters. In practice, the loss function is applied to a batch of inputs separately, and the arithmetic mean of all scalar loss function return values of the individual input samples is used as a loss function to differentiate. The size of this input batch is called batch size. A computer scientist wants to find the global minimum of that function concerning all machine learning model parameters in the general case. A visualized loss surface where the loss function return value is plotted in the z-axis and all possible model parameter combinations are given as points on the plane is given as follows:

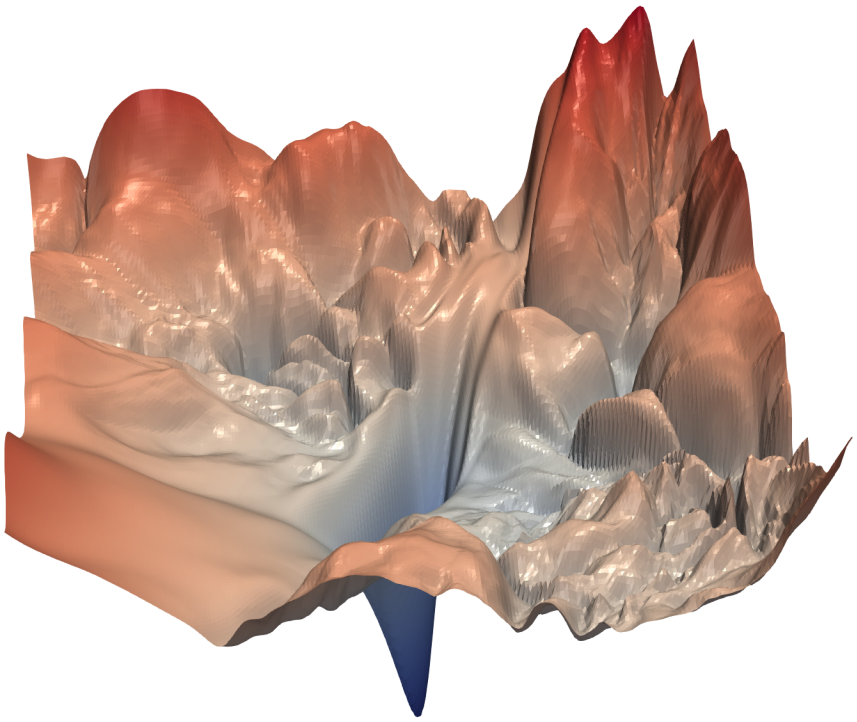

Figure 1.1: visualized loss surface [LXT+18, p. 1]

As this is a problem that cannot be solved analytically in most cases, it is approximated by using gradient descent [RHW86, p. 6-12]. This method incrementally changes each parameter depending on the gradient of each parameter's loss function in a lockstep fashion. By denoting the loss function with L, the learning rate with α, the whole old parameter set with p, the old single scalar parameter with p_i and the new single scalar parameter with p'_i, the formula to update the individual parameters p_i in a single gradient descent step can be given as follows [RHW86, p. 6-12]:

$$\forall p_i : \; p'_i = p_i - \alpha * \frac{\partial L}{\partial p_i}(p) \tag{1.1}$$

It is essential to note that the model and the loss measure must be deterministic functions for the gradient to exist. This update rule ensures that if the loss function increases with increasing p_i, a decrease of the parameter will happen, leading to a decreasing loss function result. The opposite case holds as well, which is why there is a minus sign in Equation (1.1). The learning rate α determines how significant in magnitude the update to the parameters should be at each gradient descent step. A too-small learning rate will lead to slow convergence, and a too-large learning rate will lead to divergence. Therefore, a too-large learning rate is far more dangerous than a too-small one. Convergence means that the parameter updates have led to a local minimum of the loss function. There are

no guarantees that this is the global minimum. Divergence means that the loss function diverges towards infinity. A local minimum or convergence can be reached by applying the gradient descent update rule to as many inputs as needed to set the loss function derivative to nearly zero. The trajectory of the parameter set on the loss surface when repeatedly applying the gradient descent update rule was visualized in [CPGK19, p. 2] with the initial starting parameter set denoted as a black triangle as follows:

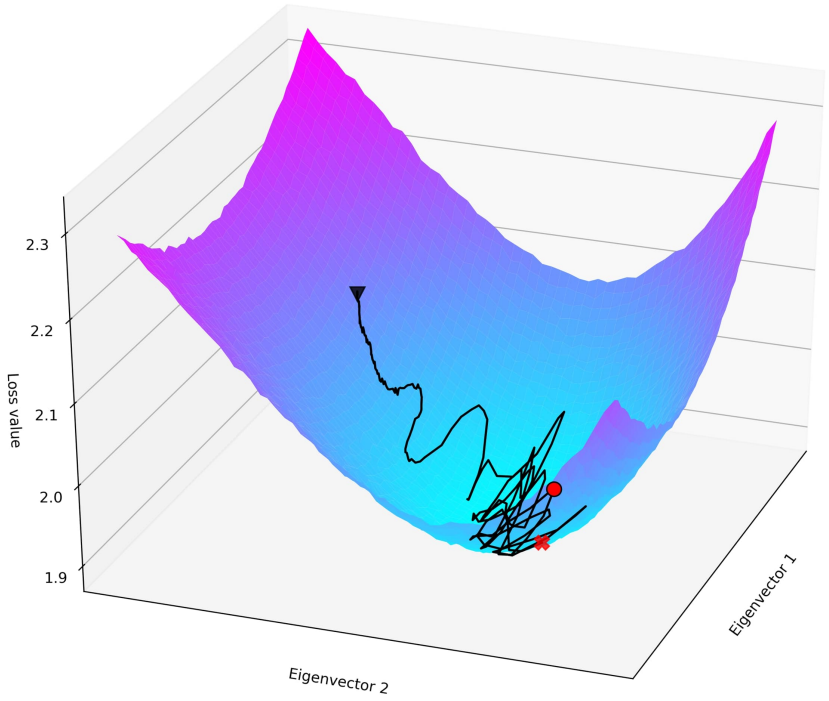

Figure 1.2: visualization of gradient descent

The differentiation of the loss function, which can be represented as a computational graph with lots of nested functions, involves many chain rule applications for the individual model parameter derivatives. The machine learning term for repeatedly applying the chain rule is backpropagation. If these nested functions correspond to applying the same machine learning model function across multiple input time steps as done in recurrent neural networks, then this backpropagation procedure can also be called backpropagation through time as introduced in [RHW86, p. 6-12]. The chain rule for differentiating $z(y(x_0))$ with respect to x for $x = x_0$ where z and y are both functions in a single variable is given by:

$$\frac{dz}{dx}\Big|_{x=x_0} = \frac{dz}{dy}\Big|_{y=y(x_0)} * \frac{dy}{dx}\Big|_{x=x_0} \tag{1.2}$$

The above equation reveals that a machine learning framework has to compute all partial derivatives of all functions present in the above-mentioned computational graph.

Furthermore, it must keep track of the so-called activations, which are denoted by $y(x_0)$ in the above Equation (1.2), as otherwise the gradient of the loss function with respect to the individual parameters cannot be computed. As this can use lots of memory, reversible layers were introduced by [GRUG17] where intermediate activations can be computed from the layer's output vector, which makes storing intermediate activations obsolete.

1.2 Problem Statement

As the sheer amount of different machine learning models can be overwhelming, the task was to fix a distinct application domain and compare the most influential machine learning models in this domain with suitable benchmarks. Benchmarks are just large input data sets with associative expected outputs. Additionally, ideas for possible improvements in existing architectures should be implemented and benchmarked against existing ones. All benchmarked models should be implemented in the same machine learning framework, and the benchmark suite should be extensible and reusable for other machine learning research projects. The whole implementation work done for this thesis should be accessible for everyone by open-sourcing all the code. As mentioned in the abstract, all the models covered in this thesis are either derivatives of the recurrent neural network or the Transformer [VSP+17] architecture. The benchmarks used in this thesis either test the models for their capabilities to capture long-term dependencies or their ability to model physical systems.

1.3 How to better model Physical Systems

Differential equations guide physical systems. The relation between system state x, system input u and system output y is given by the state derivative function f and the output function h, both of which depend on the absolute time t, as follows:

$$\dot{x}(t) = f(x(t), u(t), t) \tag{1.3}$$

$$y(t) = h(x(t), u(t), t) \tag{1.4}$$

This form of system description applies to all continuous physical systems in our daily surroundings. Most of these systems are even time-invariant. This means the functions f and h do not depend on the absolute time t. For example, a mechanical pendulum will now approximately behave the same as in one year, as its dynamics do not depend on the absolute time t. The system description presented in Equation (1.3) and Equation (1.4) proposes that machine learning models built similarly and whose state is also determined by a differential equation should be pretty capable of modeling the input-output relation of physical systems. When the benchmarked models are introduced in more detail, it can be seen that all continuous-time machine learning models use a comparable structure in terms of parameterizing the state derivative and the output function. The key takeaway point is that continuous physical systems map an input function $x(t)$ to an output function $y(t)$ as visualized in [Smi97, p. 102]:

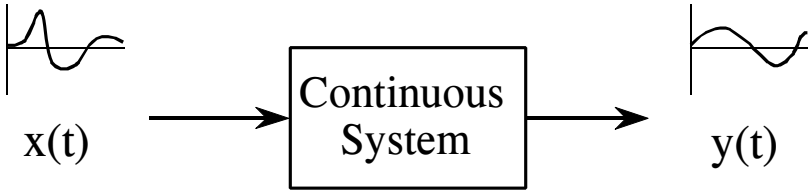

Figure 1.3: visualization of input-output relation of a continuous system

1.4 Sampled Physical Systems

As the current state's evaluation x at time point t', with initial state x_0 given the dynamics from Section 1.3, can be computationally very expensive or even infeasible, sampling was introduced to avoid solving a complex differential equation. Therefore, the whole system is only observed at equidistant successive time instants, values belonging to this time instant are denoted with a subscript index $k \in \mathbb{Z}$, and the system is now called discrete. Difference equations guide discrete systems. The relation between system state x, system input u and system output y is given by the next state function f and the output function h, both of which depend on the time instant k, as follows:

$$x_{k+1} = f(x_k, u_k, k) \tag{1.5}$$

$$y_k = h(x_k, u_k, k) \tag{1.6}$$

It must be noted that x and y are time series in discrete systems and no more functions like in continuous-time physical systems. This slightly off-topic explanation is necessary, as vanilla recurrent neural networks are built using the same principle. The system equations, Equation (1.5) and Equation (1.6), require a regularly (equidistantly) sampled input x. A similar argument as before in Section 1.3 proposes now that a machine learning model with a similar structure, which gets a regularly sampled input of a physical system, should also be pretty capable of modeling the input-output relation of this sampled physical system. The corresponding machine learning models are then called discrete-time machine learning models. The key takeaway point is that discrete physical systems map an input sequence $x[n]$ to an output sequence $y[t]$ as visualized in [Smi97, p. 102]:

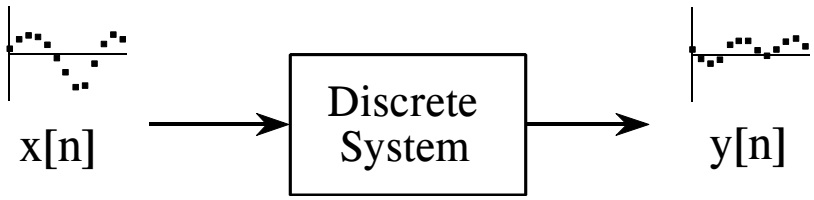

Figure 1.4: visualization of input-output relation of a discrete system

1.5 Why capturing Long-Term Dependencies is difficult

The difficulty will be outlined solely on the example of vanilla recurrent neural networks (RNNs). How Transformer-based and advanced RNN architectures tackle the problem will be discussed later. Vanilla recurrent neural networks are discrete-time machine learning models. Its dynamics are similar to the equations that govern sampled physical systems in Section 1.4. The current state vector h_t and the next input vector x_{t+1} determine the next state vector h_{t+1} and output vector y_{t+1} deterministically. In this model, all the past inputs are implicitly encoded in the current state vector. This implicit encoding entails a big challenge for computer scientists, as computers only allow states of finite size and finite precision, unlike our physical environment, which results in an information bottleneck in the state vector. The next state of a vanilla recurrent neural network h_{t+1} and its output y_t is typically computed by equations like the two proposed in [ASB16, p. 2] by using a non-linear bias-parametrized activation function σ, three matrices (W, V and U) and the output bias vector b_o:

$$h_{t+1} = \sigma(W * h_t + V * x_{t+1}) \tag{1.7}$$

$$y_t = U * h_t + b_o \tag{1.8}$$

Without the time shift on the input in the next state equation given in Equation (1.7), the equations are similar to those describing sampled physical systems. Equation (1.7) can be visualized by the following figure:

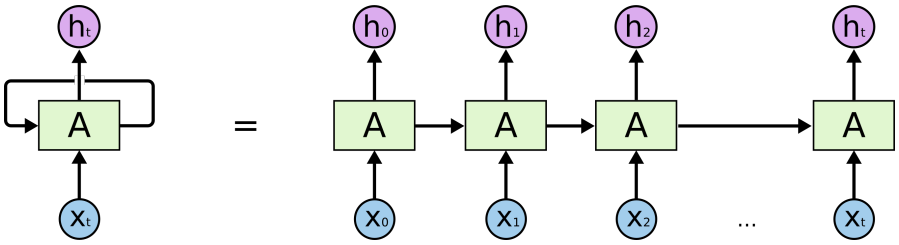

Figure 1.5: visualization of an RNN state update [Ma16]

The following inequality from [ASB16, p. 2] using norms shows the relation between the loss derivative, a recent state h_T and a state from the distant past h_t where $T \gg t$. The notation is kept similar to the examples before. A subscript 2 after a vector norm denotes the Euclidean norm, and a subscript $2, ind$ after a matrix norm denotes the spectral norm:

$$\left\| \frac{\partial L}{\partial h_t} \right\|_2 \leq \left\| \frac{\partial L}{\partial h_T} \right\|_2 * \|W\|_{2,ind}^{T} * \prod_{k=t}^{T-1} \left\| diag(\sigma'(W * h_k + V * x_{k+1})) \right\|_{2,ind} \tag{1.9}$$

This inequality contains all essential parts to understand why capturing long-term dependencies with vanilla recurrent neural networks is difficult. Some problems that machine learning tries to solve require incorporating input data from the distant past

to make good predictions in the present. As these inputs are implicitly encoded in the states of the distant past, $\left\|\frac{\partial L}{\partial h_t}\right\|_2$ should not decay to zero or grow unboundedly to effectively tune the parameters using the gradient descent update rule shown above in Equation (1.1). This persistence of the gradient ensures that distant past inputs influence the loss function reasonably and makes it feasible to incorporate the knowledge to minimize the loss function. As known, the spectral norm of the diagonal matrix in Equation (1.9) is just the largest magnitude out of all diagonal entries. Therefore, if the diagonal matrix's norm is close to zero over multiple time steps k, the desired loss gradient will decay towards zero. Otherwise, if the diagonal matrix's norm is much larger than one over multiple time steps k, the desired loss gradient may grow unboundedly. Using this knowledge, it is now clear that a suitable activation function must have a derivative of one in almost all cases to counteract the above-described problems. A good fit would be a rectified linear unit (relu) activation function with an added bias term. The relu activation function with a bias b can simply be discribed by the function $max(0, x + b)$. The max function should be applied element-wise. As the requirements for the activation function candidates are precisely formulated now, the next thing to discuss is the norm of the matrix W. If $\|W\|_{2,ind} > 1$, $\left\|\frac{\partial L}{\partial h_t}\right\|$ may grow unboundedly, making it difficult to apply the gradient descent technique to optimize parameters. If $\|W\|_{2,ind} < 1$, $\left\|\frac{\partial L}{\partial h_t}\right\|$ will decay to 0, making it impossible to apply the gradient descent technique to optimize parameters. These problems are identical to the norm of the diagonal matrix and have the same implications. The first case is called the exploding gradient problem, and the second case is called the vanishing gradient problem for given reasons. Both phenomena are explained in more detail in [BSF94].

1.6 Aim of the Work

This work should help to objectively compare various machine learning models used to process regularly sampled time series data. It should outline the weaknesses and strengths of the benchmarked models and determine their primary domain of use. Moreover, as there are many models benchmarked, their relative expressivity across various application domains can be compared reasonably well. Another aim is to provide an overview of what architectures are currently available and how they can be implemented. Furthermore, the implemented benchmark suite should be reusable for future projects in the machine learning domain.

1.7 Methodological Approach

The first part of this thesis was to determine the most influential models for processing time series data. Some models that were benchmarked against each other in this thesis were taken from [LH20], even though this paper focuses primarily on irregularly sampled time series. The other models were implemented according to the following architectures: Long Short-Term Memory [HS97], Differentiable Neural Computer [GWR+16], Unitary

Recurrent Neural Network [JSD+17], Transformer [VSP+17] and Neural Circuit Policies [LHA+20]. These nine models are then complemented by five models that were newly introduced. All these models are benchmarked against each other. Additionally, the time-continuous Memory Cell architecture should be introduced. This architecture must have a dedicated benchmark test and should not be benchmarked against all other fully-fledged machine learning models as it is only a proof-of-concept implementation. All mentioned models should be implemented in the machine learning framework Tensorflow [AAB+15]. After implementing all models, an extensible benchmark suite had to be implemented to compare all implemented models. A basic benchmark framework should be implemented, which automatically trains a given model and saves all relevant information regarding the training process, including generating plots to visualize the data. All that should be needed to implement a new benchmark is to specify the input, the expected output data, the loss function, and the model's required output vector size. The benchmarks regarding person activity classification, sequential MNIST classification, and kinematic physics simulation were taken from [LH20] and were modified slightly to be compatible with the benchmark framework. The other two benchmarks regarding the copying memory and the adding problem were taken from [ASB16] but were also slightly modified to fit the benchmark framework's needs. The sixth benchmark that had to be implemented was the Cell Benchmark that should check if the Memory Cell can store information over many time steps. When this step is also done, all benchmarks should be run on all applicable models, and then the results should be thoroughly compared to filter out the strengths and weaknesses of the diverse models. Only after that, a summary should be written to concisely summarize the most important discoveries and fallacies that were made.

1.8 State of the Art

The whole field of sequence modeling started with recurrent neural networks. More and more modern machine learning architectures exploit the fact that continuous-time models are very well suited for tasks related to dynamical physical systems as explained in Section 1.3. A few examples for such models would be the CT-GRU [MKL17], the LTC network [HLA+20] and the ODE-LSTM architecture [LH20]. Nevertheless, some older architectures exploit continuous-time dynamics in machine learning models like the CT-RNN architecture [iFN93]. The other problem described in the previous chapters is the challenging task of capturing long-term dependencies in time series. One solution for the problem was proposed in [ASB16], which introduced the Unitary RNN architecture. In principle, this architecture uses the vanilla RNN architecture described above. The difference is that the matrix W fulfills $\|W\|_{2,ind} = 1$ to tackle the vanishing and exploding gradient problem. This idea was later refined by [JSD+17]. The vanishing gradient problem was also tackled by the LSTM architecture [HS97] using a gating mechanism. This mechanism changes the subsequent state computation of the vanilla RNN. Another possible mitigation to the vanishing gradient problem is the transformer architecture proposed in [VSP+17] using a mechanism called attention. In principle, the Transformer

architecture model has access to all past inputs simultaneously and directs its attention to the inputs most relevant for solving the required task. This global access eliminates the need to backpropagate the error through multiple time-steps, which keeps the number of backpropagation steps low.

CHAPTER 2

Models

2.1 Model Factory

As all the benchmarks require variants of the same models with different output vector sizes, a model factory function was implemented to produce an output tensor given the model's name, the output vector size, and the input tensor tuple. This function was called `get_model_output_by_name` and can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. This mechanism of creating the output tensor, including the internal computational graph of the Tensorflow library [AAB+15] is called the Functional API. Most of the models were parameterized such that they have roughly 20000 trainable parameters in the Walker Benchmark described in Section 3.4 as this benchmark features the largest input and output vector size of all benchmarks. The exceptions of the parameter count are the Unitary RNN model given in Section 2.8, the NCP model given in Section 2.7, the Unitary NCP model given in Section 2.10, the Recurrent Network Augmented Transformer given in Section 2.12 and the Recurrent Network Attention Transformer given in Section 2.13. All these models' computational graphs lead to high computation costs during backpropagation, which leads to training durations up to a whole day for a single benchmark. This high computational cost was unacceptable, and therefore their parameter count was reduced to present at least some results for these models.

2.2 LSTM

The LSTM (Long Short-Term Memory) recurrent neural network architecture is a discrete-time machine learning model introduced in Section 1.4. The model has an ordinary (hidden) state vector and a cell state vector, which should store information over a longer time horizon than the hidden state vector. This thesis uses the open-source LSTM

implementation provided by the Keras library [C+15] which is based on the original LSTM paper [HS97] as well on its successor paper [GSC00] that introduces a forget mechanism for the LSTM. The function the LSTM model is applying to its inputs to produce the outputs is given as follows with inputs denoted as x_t and outputs which equals the hidden states denoted as h_t [GSC00, p. 4-8]:

$$f_t = sigmoid(W_f * x_t + U_f * h_{t-1} + b_f) \tag{2.1}$$
$$i_t = sigmoid(W_i * x_t + U_i * h_{t-1} + b_i) \tag{2.2}$$
$$o_t = sigmoid(W_o * x_t + U_o * h_{t-1} + b_o) \tag{2.3}$$
$$\tilde{c}_t = tanh(W_c * x_t + U_c * h_{t-1} + b_c) \tag{2.4}$$
$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t \tag{2.5}$$
$$h_t = o_t * tanh(c_t) \tag{2.6}$$

The term f_t in Equation (2.1) is the forget gate's activation vector, i_t in Equation (2.2) is the input gate's activation vector, o_t in Equation (2.3) is the output gate's activation vector, $\tilde{c}_t$ in Equation (2.4) is the cell input activation vector, c_t in Equation (2.5) is the cell state vector and h_t in Equation (2.6) is the hidden state vector or also called output vector of the LSTM model. The initial hidden state h_0 and the initial cell state c_0 are picked to the all-zero vector. Matrices are denoted with capital letters, and vectors are denoted with lower case letters. The LSTM model has a configurable state size. The multiplication sign between two vectors denotes a scalar product, and it denotes matrix multiplication between matrices and vectors. This convention is used throughout this thesis. Dimensions of matrices are picked such that the resulting vector has the required state size, which is configurable. The bias vectors denoted with b also have the required state dimension. The matrices denoted by W map the input vector in each time step and the matrices denoted by U map the hidden state vector at each time step to a resulting vector. This architecture was also visualized in the successor paper as follows:

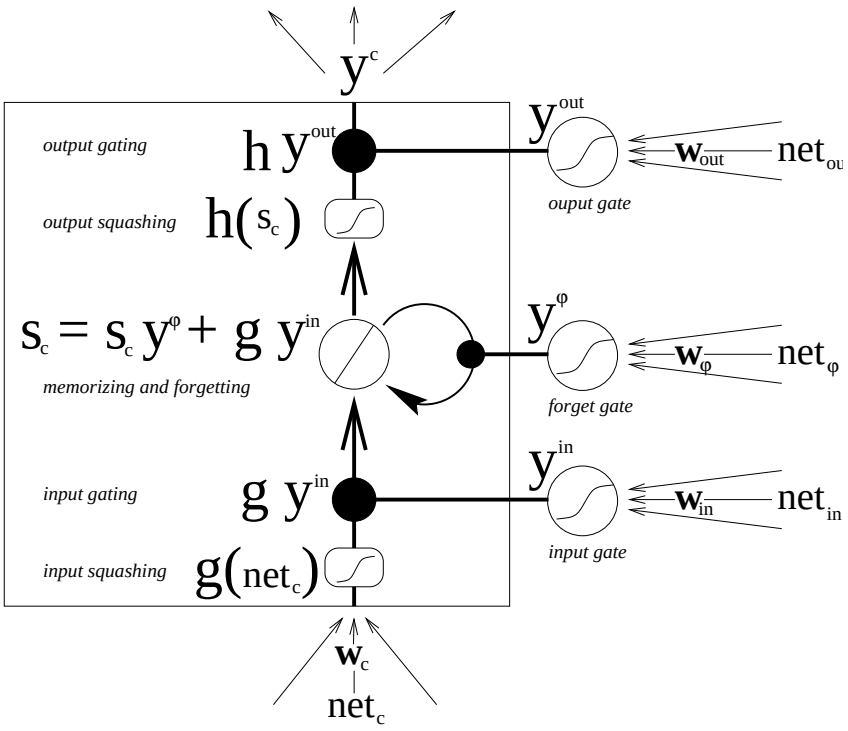

Figure 2.1: visualized LSTM architecture [GSC00, p. 6]

The model structure allows it to capture long-term dependencies by setting f_t equal to one and i_t equal to zero in some common vector indices i, and only the previous cell state is used to build the next cell state in these following cell state vector entries. This will lead to $\frac{\partial c_{t,i}}{\partial c_{t-1,i}} = 1$, as this clearly approximates the identity function for a specific index i in the cell state vector. Backpropagation to activations in the distant past is feasible using this model function as gradients are not vanishing or exploding when the model's parameters are correctly learned. This mechanism is called the constant error carrousel described in [HS97, p. 7]. LSTMs can incorporate this mechanism to store essential information from the distant past, making accurate predictions when long-term dependencies are present. Furthermore, the model can also decide to forget the previous cell state entirely if the current input vector makes the stored cell state obsolete in the corresponding application. This forgetting is done by learning to set the forget gate's activation vector close to zero, and the cell input activation vector is then used to fill the cell state again if the input gate's activation vector is set accordingly. The output gate's activation vector determines which portion of the cell state is used to build the hidden state or output vector of the LSTM model. Throughout the thesis, an LSTM model with a fixed state vector size of 64 was used. As mentioned in the benchmark framework section, each model must support an arbitrary output vector size. The correct model output vector size is accomplished by postprocessing the hidden state outputs of

the LSTM by a dense layer with the required amount of output neurons and without an activation function. The output y of a dense layer without an activation function and input vector x can simply be given by: $y = W * x + b$. In this notation, W is a matrix such that it maps the input vector x to the required output size, and b is just a bias vector as in the functions describing the LSTM model. The following figure visualizes three dense layers with parameters denotes as arrows:

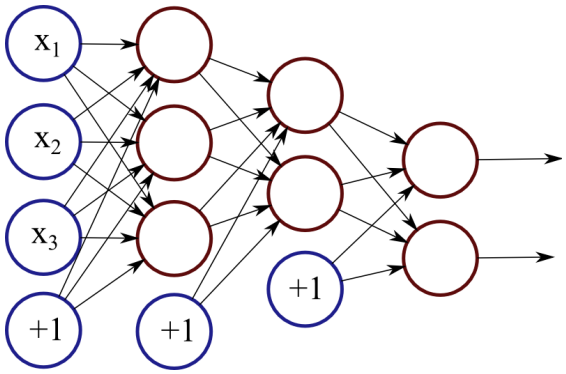

Figure 2.2: visualized dense layers [Rei14]

Training the LSTM model from the Keras library is fast as it uses an optimized cuDNN [CWV+14] implementation. The LSTM model implementation used in this thesis is exposed under the `get_lstm_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`.

2.3 GRU

The GRU (Gated Recurrent Unit) recurrent neural network architecture is a discrete-time machine learning model introduced in Section 1.4. The model has only a single ordinary hidden state vector. This thesis uses the open-source GRU implementation provided by the Keras library [C+15] which is based on the original GRU paper [CGCB14]. The GRU model tries to simplify the LSTM architecture by removing the output gate, for example, without sacrificing expressivity. This simplification leads to a smaller parameter count of a GRU model with the same hidden state vector size as an LSTM model. The function the GRU model is applying to its inputs to produce the outputs is given as follows with inputs denoted as x_t and outputs which equals the hidden states denoted as

h_t [CGCB14, p. 4]:

$$z_t = sigmoid(W_z * x_t + U_z * h_{t-1} + b_z) \tag{2.7}$$
$$r_t = sigmoid(W_r * x_t + U_r * h_{t-1} + b_r) \tag{2.8}$$
$$\tilde{h}_t = tanh(W_h * x_t + U_h * (r_t * h_{t-1}) + b_h) \tag{2.9}$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \tag{2.10}$$
$$\tag{2.11}$$

The term z_t in Equation (2.7) is the update gate vector, r_t in Equation (2.8) is reset gate vector, $\tilde{h}_t$ in Equation (2.9) is the candidate activation vector and h_t in Equation (2.10) is the hidden state or output vector of the GRU model. The initial hidden state h_0 is picked to the all-zero vector. The notation of operations, matrices, and vectors stay the same as for the LSTM architecture introduced in Section 2.2. Subtraction in Equation (2.10) is meant element-wise, and the 1 should denote the all-one vector. As in the LSTM architecture, the hidden state vector size is configurable, and all matrices map their inputs to a vector of the corresponding hidden state vector size. This architecture was also visualized in the original paper as follows:

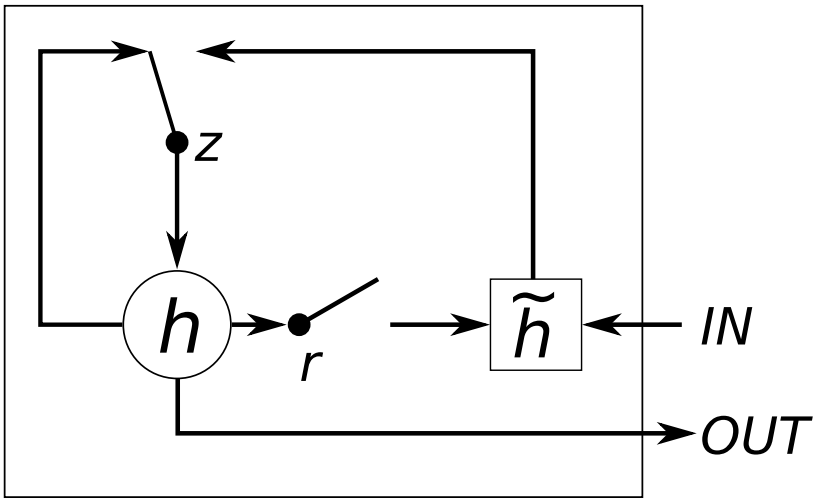

Figure 2.3: visualized GRU architecture [CGCB14, p. 3]

The model structure allows it to capture long-term dependencies as by setting z_t equal to zero for some vector entries, only the previous hidden state vector is used to build the next hidden state vector for these indices i. This will lead to $\frac{\partial h_{t,i}}{\partial h_{t-1,i}} = 1$, as this clearly approximates the identity function for a specific index i in the hidden state vector. Backpropagation to activations in the distant past is feasible using this model function as gradients are not vanishing or exploding when the model's parameters are correctly

learned. As also mentioned in [CGCB14, p. 5], the LSTM architecture does not expose its entire cell state in the output vector as the cell state is further processed using the output gate. However, the GRU architecture exposes its entire cell state at each time step as it does not have an output gate, as mentioned before. Another critical difference between the LSTM and GRU architecture is that the LSTM architecture controls the portions of the previous cell state and the portions of the cell input activation that add up to the next step cell state separately using the forget gate's activation vector and the input gate's activation vector in Equation (2.5). The GRU model simplifies this mechanism by providing just a single update gate vector z. The other vector controlling the portion from the previous hidden state added together to build the next step hidden state vector is then determined by subtracting z from the all-one vector in Equation (2.10). This subtraction is feasible as the sigmoid activation function produces only outputs lying in the interval $[0, 1]$. Furthermore, the reset mechanism works differently in the GRU architecture as the reset vector solely operates on the previous step hidden state vector when computing the next state candidate activation vector. Throughout the thesis, a GRU model with a fixed state vector size of 80 was used. As mentioned in the benchmark framework section, each model must support an arbitrary output vector size. This variable output vector size is accomplished by postprocessing the hidden state outputs with a dense layer, just like in the LSTM architecture introduced in Section 2.2. Training the GRU model from the Keras library is fast as it uses an optimized cuDNN [CWV+14] implementation. The LSTM model implementation used in this thesis is exposed under the `get_gru_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`.

2.4 CT-RNN

The CT-RNN (continuous-time recurrent neural network) was first proposed in [iFN93] and is a continuous-time machine learning model as described in Section 1.3. This thesis uses an implementation taken from the repository of the paper [LH20] which can be found under the URL `https://github.com/mlech26l/ode-lstms`. The CT-RNN has a configurable hidden state vector size, and its output vector is equal to its hidden state vector at each time step. The hidden state vector is parametrized as follows [iFN93, p. 2] with the same notation as introduced in Section 2.2:

$$\dot{h}(t) = -\frac{h(t)}{\tau} + W * sigmoid(h(t)) + i(t) \tag{2.12}$$

The division between $h(t)$ and the vector τ is understood element-wise. The vector τ is also called the time constant as it is the time constant of the exponential decay of the hidden state vector over time. As the input has not, in general, the same dimension as the hidden state vector, the input is preprocessed by mapping it to the proper dimension with matrix multiplication. Furthermore, the implementation used for benchmarking has a tanh activation function as it is applied at a different position in the formula to allow for negative activations. There is also an additional bias vector b and scaling vector α

introduced whose multiplication is to understand element-wise. The derivative in the CT-RNN implementation used for benchmarking is given by:

$$\dot{h}(t) = -\frac{h(t)}{\tau} + \alpha * tanh(W_h * h(t) + W_i * i(t) + b) \tag{2.13}$$

The idea of parameterizing the derivative (change) of the hidden vector (activation) rather than computing a completely new hidden vector or activation was extensively reused in recent research. For example, ResNets [HZRS15] used the idea in a discrete-time model, and Neural ODEs [CRBD19] reused it in a continuous-time model, which features a similar model function as the CT-RNN. In discrete-time models, residual connections are added, which help backpropagation in a deep machine learning architecture as they are just representing the identity function, which is easily differentiable. For more information on residual connections, consult the corresponding paper [HZRS15]. As the benchmark input samples are only regularly sampled vectors and not a function $i(t)$ as needed by the Equation (2.13) of the CT-RNN model, each input sample is continuously held for 1 time unit to form the input function. This mechanism is used for all continuous-time models throughout this thesis. Therefore, the input function is defined on the interval $[0, T]$ where T is the input sequence length. The output of the CT-RNN after consuming the whole input function $i(t)$ from time 0 to time T is then given by the hidden state vector $h(T)$ at time T. There is also the possibility to evaluate the hidden state at intermediate time points, for example, at $T - 1$, which equals $h(T - 1)$. With this mechanism, any continuous-time model can also map an input vector sequence to an output vector sequence. If additional timing information is available about the input vectors, it can be used to hold this specific input constantly in the input function. This variable time input leads to an irregularly sampled time series where time-continuous models are exceptionally well suited as machine learning models, as discrete-time models as given in Section 1.4 implicitly model a regularly sampled continuous-time system. This statement was also shown to be valid by [LH20]. The initial state of the CT-RNN is given by $h(0)$, which is picked to the all-zero vector. To compute the final hidden state $h(T)$, the ODE (ordinary differential equation) from Equation (2.13) must be solved given the initial condition $h(0)$. This solving procedure can be done by incorporating ODE solvers, which compute $h(T)$ by approximately integrating $\dot{h}(t)$ with guarantees on the error bound. Then $h(T)$ is given by $h(0) + \int_0^T \dot{h}(t)\ dt$. In all continuous-time models implementations the ODE solver is called at each time step computing the next step hidden state $h(t + 1)$ as $h(t) + \int_t^{t+1} \dot{h}(t)\ dt$. Examples for ODE solvers are the explicit Euler method, the RK4 (Runge-Kutta 4^{th} order) method, or the Dormand-Prince method. The Dormand-Prince method is the default ODE solver used in the `ode45` solver of MATLAB [MAT20]. All of these are members of explicit methods and the Runge-Kutta methods to solve ODEs. Explicit methods calculate the state at a later time only from the state at the current time. There are also implicit methods, which find a solution by solving an equation involving both the state at the current time and the state at the next time. Implicit methods are primarily used for stiff ODEs, characterized by minor numerical deviations that may lead to a considerable output change. For the CT-RNN implementation, the RK4 method was used to solve the ODE. The hidden state

vector size was picked to 128, and the number of unfolds was set to 3. The number of unfolds determines how often an ODE solver is called on a single input sample. This number means that instead of integrating the whole interval of length 1 at each time step, the ODE solver integrates an interval of length $\frac{1}{3}$ three times, which yields more accurate results. Computing the loss gradient with respect to the model parameters is still possible for continuous-time models as the ODE solvers are just functions that can be differentiated. The ODE solver can also be run as a black-box without knowing its internal operations as shown in [CRBD19]. The gradients for the functions applied by the ODE solver can be computed by the adjoint sensitivity method [Pon62]. As pointed out by [HLA+20, p. 3], this memory-efficient procedure, however, comes with numerical errors as it forgets the forward-time computational trajectories. The CT-RNN model implementation used in this thesis is exposed under the `get_ct_rnn_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/ct_rnn.py`.

2.5 CT-GRU

The CT-GRU (continuous-time gated recurrent unit) recurrent neural network architecture is a continuous-time machine learning model firstly introduced in [MKL17]. The implementation of the CT-GRU architecture used in this thesis was taken from the repository of [LH20]. It shares many concepts with the GRU architecture introduced in Section 2.3, but the update gate in Equation (2.7) and reset gate in Equation (2.8) operate on multiple hidden state vectors stored across various time scales. This redundancy was introduced because some information may become obsolete quickly, whereas some other information may also be vital in the longer term. These rates of information decay are referred to as time scales. The time scales are represented using time constants, and the number of time scales was fixed to 8 in this thesis. Therefore, the update gate is then called the storage scale, and the reset gate is then called the retrieval scale as they operate not only on a single hidden vector but across hidden vectors stored across multiple time scales. They can be thought of as multi-dimensional gates. As the amount of time scales is fixed, input data that matches a particular time scale not present in the fixed set must be approximated using a combination of the available time scales. This approximation is indeed possible with a small error when the time scale to approximate is in a specific range as pointed out in [MKL17, p. 5-6]. Then the half-life of the exponentials' combination approximately matches the corresponding exponential half-life to the correct time scale. A good match for time constants τ_i representing the various time scales is the set of constants where $\tau_0 = 1$ and $\tau_{i+1} = \sqrt{10} * \tau_i$. This set was also used in the benchmarked implementation. The explicit time input called Δt_k of this model was not used as an interval to integrate an ODE but instead as the time duration of exponential decay between two input vectors. As all benchmarks do not provide time inputs and the benchmarks' input vectors are regularly sampled, Δt_k was set to constant

1. The function the GRU model is applying to its input vectors to produce the output vectors or hidden state vectors is given as follows with inputs denoted as x_k and outputs which equals the hidden states denoted as h_k [MKL17, p. 7]:

$$\ln \tau_k^R = W^R * x_k + U^R * h_{k-1} + b^R \tag{2.14}$$

$$r_{ki} = softmax_i(-(\ln \tau_k^R - \ln \tau_i)^2) \tag{2.15}$$

$$q_k = tanh(W^Q * x_k + U^Q * (\sum_i r_{ki} *_{ew} \tilde{h}_{k-1,i}) + b^Q) \tag{2.16}$$

$$\ln \tau_k^S = W^S * x_k + U^S * h_{k-1} + b^S \tag{2.17}$$

$$s_{ki} = softmax_i(-(\ln \tau_k^S - \ln \tau_i)^2) \tag{2.18}$$

$$\tilde{h}_{ki} = [(1 - s_{ki}) *_{ew} \tilde{h}_{k-1,i} + s_{ki} *_{ew} q_k] * e^{-\frac{\Delta t_k}{\tau_i}} \tag{2.19}$$

$$h_k = \sum_i \tilde{h}_{ki} \tag{2.20}$$

Multiplication, which is meant element-wise, is denoted with subscript ew. Otherwise, the notation is kept the same as in previous models. The equations, Equation (2.14) and Equation (2.15), determine the retrieval scale and compute the weighting for each time scale. Equation (2.17) and Equation (2.18) determine the storage scale and compute the weighting for each time scale. The retrieval scale vector r_{ki} is the multi-dimensional equivalent to the GRU architecture's reset vector. The storage scale vector s_{ki} is the multi-dimensional equivalent to the GRU architecture's update vector. Equation (2.16) describes how the next candidate hidden state vector s_k is computed. Finally, Equation (2.19) describes how the hidden state for each time scale is updated, and Equation (2.20) describes how the output vector h_k is computed out of the multi-dimensional hidden state vector. It can be said that the CT-GRU architecture is a GRU model with a multi-dimensional state and exponential decay of its state between input vector observations with different time constants. Most of the features discussed for the GRU model are also applicable to the CT-GRU architecture. This architecture was also visualized in the original paper as follows:

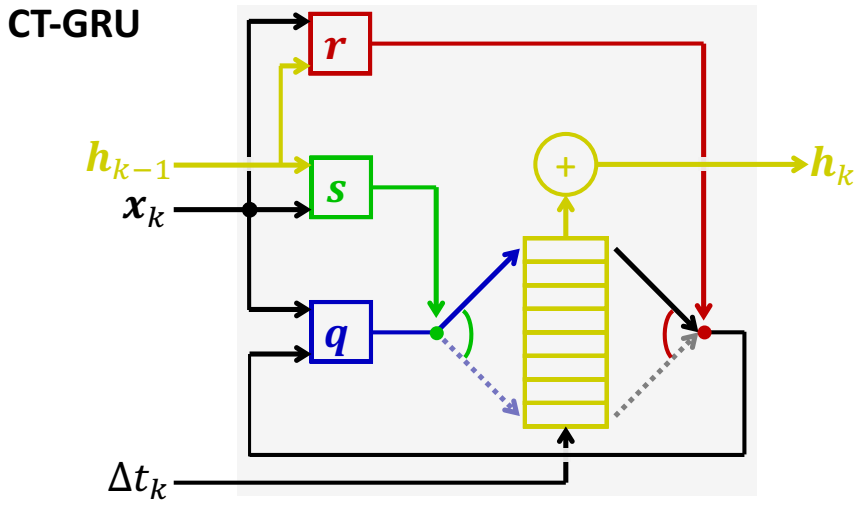

Figure 2.4: visualized CT-GRU architecture [MKL17, p. 4]

It should also capture long-term dependencies as time scales featuring an ample time constant have minor decay on their corresponding hidden state, and then simply the argument used in the GRU architecture in Section 2.3 can also be applied here. Like other models, the CT-GRU has a configurable hidden state vector size, picked to 32 throughout this thesis. The CT-GRU model implementation used in this thesis is exposed under the `get_ct_gru_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/ct_gru.py`.

2.6 ODE-LSTM

The ODE-LSTM recurrent neural network architecture is a continuous-time machine learning model firstly introduced in [LH20]. The implementation of the ODE-LSTM architecture used in this thesis was taken from the repository of its original paper [LH20]. This model's idea is to combine the LSTM architecture's ability to capture long-term dependencies and the ability of CT-RNNs to accurately model dynamical physical systems, even if an irregularly sampled time series is provided to the model as input. As in this thesis, only regularly sampled time series are used. The continuous-time model is continually fed with the time input 1 as mentioned in Section 2.4. This constant time input should be no problem as the ability to model dynamical physical systems generalizes to any time input very well. Like the LSTM architecture, the ODE-LSTM has two state vectors: one hidden state vector h_i and one cell state vector c_i. Both vectors

are initialized to the all-zero vector. The function the ODE-LSTM model is applying to its input vectors to produce the output vectors or hidden state vectors is given as follows with inputs denoted as x_i and outputs denoted as h_i [LH20, p. 5]:

$$(c_i, h_i') = LSTM(x_i, (c_{i-1}, h_{i-1})) \tag{2.21}$$

$$h_i = CTRNN(h_i', (h_{i-1})) \tag{2.22}$$

The function $LSTM$ denotes one model function step of the LSTM model introduced in Section 2.2 starting from the given state (c_{i-1}, h_{i-1}) for input x_i. The function $CTRNN$ denotes one model function step of the CT-RNN model introduced in Section 2.4 starting from the given state (h_{i-1}) for input x_i, the input is set to 1 for each time step. Implementation-wise, the CTRNN model function call was done to the implementation described in Section 2.4. The LSTM model function was implemented from scratch, and no library modules were used. As only the hidden state vector of the LSTM architecture is post-processed by the CT-RNN model, the cell state stays untouched, which should enable the architecture to learn long-term dependencies by using the same argument as in Section 2.2. By the postprocessing of the hidden state vector, which controls the LSTM's gates, the gating dynamics become dependent on the time input as well [LH20, p. 4]. Of course, the ODE-LSTM architecture has a configurable hidden state vector size, which was picked to 64. The same hidden vector size was used to initialize the CT-RNN. The number of unfolds was set to 4, and the explicit Euler method was used as an ODE solver. The ODE-LSTM model implementation used in this thesis is exposed under the `get_ode_lstm_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/ode_lstm.py`.

2.7 Neural Circuit Policies (NCP)

Neural Circuit Policies were used in the paper [LHA+20] which shows the high expressivity of the architecture in autonomous driving. The architecture is a subset of all LTC (Liquid Time-Constant) Networks that were introduced in [HLA+18] and further discussed in [HLA+20]. An LTC Network consists of biologically inspired neurons with leakage interconnected using chemical synapses with non-linear activations. LTC Networks model the cell membrane as an integrator and are therefore a continuous-time machine learning model. Neural Circuit Policies were derived from the neuron interconnection structure of the Caenorhabditis elegans nematode [LHA+20, p. 3] which trims the space of all possible LTC networks. The state of each neuron i with incoming chemical synapses from neurons j is given as its potential V_i and the ODE that describes the dynamics of a

single neuron's potential is given by [HLA+18, p. 1-2]:

$$\dot{V}_i(t) = \frac{1}{C_i} * (I_{leak,i} + \sum_j I_{syn,ji}) \tag{2.23}$$

$$I_{leak,i} = G_{leak,i} * (E_{leak,i} - V_i(t)) \tag{2.24}$$

$$I_{syn,ji} = [G_{syn,ji} * sigmoid(\sigma_{ji} * (V_j(t) - \mu_{ji}))] * (E_{ji} - V_i(t)) \tag{2.25}$$

By reordering terms in Equation (2.23), it can be shown that the time constant τ as used in Equation (2.12) in the CT-RNN architecture is varying with time. The capacitance of a neuron i is denoted as C_i and the whole equation will be more familiar when the capacitance is brought to the left hand side which yields $C_i * \dot{V}_i(t) = I_{leak,i} + \sum_j I_{syn,ji}$. This equation is just the differential equation describing the behavior of electrical conductance. The leakage current given in Equation (2.24) and the chemical synaptic current given in Equation (2.25) are written according to Ohm's law $I = \frac{U}{R}$. Using the conductance G instead of the resistance R, which is just the reciprocal value, the equation yields $I = G * U$, precisely the form both current equations are using. As the voltage U is given as the potential difference, all terms in Equation (2.24) and Equation (2.25) should be clear now. Worth mentioning is the non-linear conductance for chemical synaptic currents given as $G_{syn,ji} * sigmoid(\sigma_{ji} * (V_j(t) - \mu_{ji}))$, where the parameter $G_{syn,ji}$ controls the maximum conductance, the parameter μ_{ji} controls the mean conductance potential and the parameter σ_{ji} controls the steepness of the transition between conductance and non-conductance. Note that the non-linear synaptic conductance is only influenced by the presynaptic neuron potential $V_j(t)$. The potentials given by the capital letter E control the targeted potentials for the neuron i. Therefore if the neuron has reached this potential, the corresponding currents will vanish. The NCP architecture builds its output vector by determining output neurons in the same amount as the output vector size. These neurons are called motor neurons, and their vectorized potentials then build the output vector. The input vector entries are fed to neurons as currents using Equation (2.25) and setting the presynaptic potential equal to the input vector entry. Furthermore, before the input vector is provided to the NCP model and before the output vector is returned from the NCP model, an affine transformation is applied to the input and output vector by mapping both vectors with a dense layer as described in Section 2.2. Additionally, to motor neurons, NCP models also have inter and command neurons. Inter neurons receive input vector entries as chemical synaptic currents, and command neurons are the only neuron type where recurrent connections are allowed. Command neurons also are the only neuron type that has synaptic connections to motor neurons. Therefore, the input vector entries are processed using the inter neurons, which feed the processed information to the command neurons that control the motor neurons and, therefore, the output vector entries. This architecture was also visualized in the original paper using different colors for the four layers that represent sensory, inter, command, and motor neurons from left to right and arrows for chemical synapses as follows:

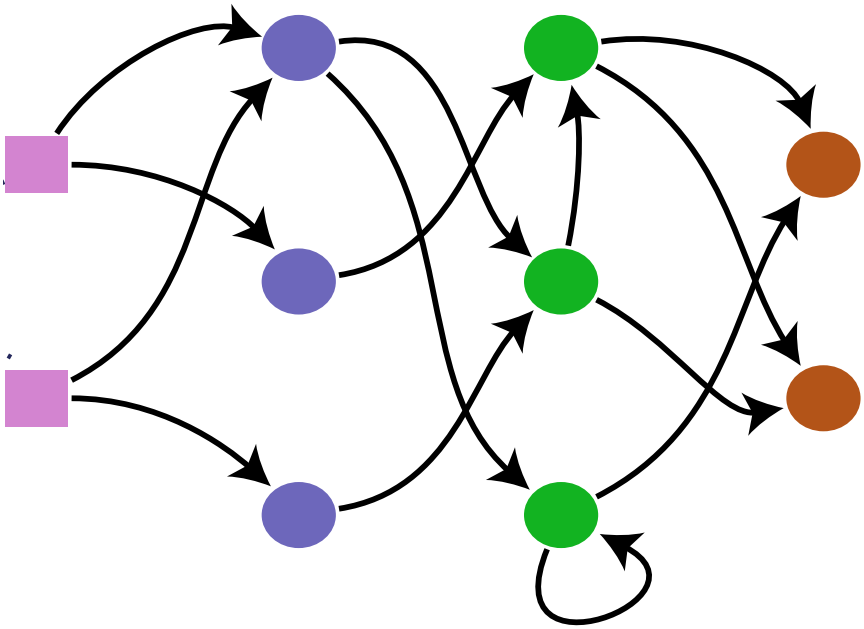

Figure 2.5: visualized NCP architecture [LHA+20, p. 3]

The procedure to create the synaptic wiring is described in detail in [LHA+20, p. 3] and will not be covered in this thesis. The NCP implementation used for benchmarking uses the implementation provided in the repository of the paper [LHA+20] located under the URL `https://github.com/mlech26l/keras-ncp`. It was configured with 9 inter neurons and 7 command neurons. The number of motor neurons was picked according to the required output vector size. There were two incoming synapses from input vector entries to inter neurons and two incoming synapses from inter neurons to command neurons. Each motor neuron receives two incoming synapses from command neurons, and there were 14 recurrent synapses in all command neurons. The time input to solve the ODE was set to 1 per time step, and the ODE was solved using the Fused Solver proposed in [HLA+20] that fuses explicit and implicit Euler methods. The ODE was unrolled 6 times per time step, as there are at least 3 unrolls necessary until the currents from the input vector reach the command neurons via synapses in each time step. The initial potential of all neurons was picked to 0. The NCP model implementation used in this thesis is exposed under the `get_neural_circuit_policies_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/neural_circuit_policies.py`.

2.8 Unitary RNN

The Unitary RNN architecture was first introduced in [ASB16] and later refined in [JSD+17] and is a discrete-time machine learning model. It uses the same vanilla recurrent neural network model function as discussed in Section 1.5. The Unitary RNN implementation used in this thesis is a modified version of the original paper's implementation, which can be found under the URL `https://github.com/jingli9111/EUNN-tensorflow/blob/master/eunn.py`. The next hidden state vector of a Unitary RNN h_{t+1} which also equals its output vector is computed as given in [JSD+17, p. 2] by using a non-linear bias-parametrized activation function σ and two matrices W and V:

$$h_{t+1} = \sigma(W * h_t + V * x_{t+1}) \tag{2.26}$$

The bias-parametrized activation function σ was set to the modrelu function firstly introduced in [ASB16, p. 4]. The modrelu function applied to a complex vector z is defined as follows for each vector entry z_i: $moderelu(z_i) = max(0, |z_i| + b_i) * \frac{z_i}{|z_i|}$ with a real-valued bias parameter b_i per vector entry. The initial hidden state vector h_0 was picked to the all-zero vector. The difference with this model is that it does not use real parameters, which is the standard in machine learning. It uses complex parameters that are represented by two single-precision floating-point parameters each. The parameter count for each model, however, is always given in terms of single-precision floating-point parameters. The matrices W and V are parametrized as complex matrices. Matrix V does not have to follow any particular restrictions. Therefore, two real matrices V_{real} and V_{imag} for the real and imaginary part are employed for parameterization. As explained in detail in Section 1.5, a matrix W that fulfills $\|W\|_{2,ind} = 1$ and a suitable activation function σ would solve the vanishing and exploding gradient problem for the vanilla recurrent neural network architecture and precisely this was done in the case of Unitary RNNs. Unitary matrices W fulfill the requirement $\|W\|_{2,ind} = 1$, as all eigenvalues of unitary matrices have a magnitude of 1 from which follows that 1 is always the largest singular value as unitary matrices are square. As the spectral norm is just the largest singular value, it is proven that unitary matrices fulfill the proposed requirement. The difficulty now is to parametrize unitary matrices efficiently as they are only a subset of all complex matrices and therefore cannot be parametrized as simple as the matrix V. The method to parametrize unitary matrices as used in [JSD+17, p. 3] was proposed by [CHM+17] and is called the square decomposition method. The core statement is that any unitary matrix of dimension $N \times N$ can be represented by matrix multiplications involving a diagonal matrix D and rotational matrices R_{ij} as follows:

$$W = D \prod_{i=2}^{N} \prod_{j=1}^{i-1} R_{ij}. \tag{2.27}$$

The diagonal matrix D has only the entries e^{iw_j} on its diagonal which results in N parameters w_j. The matrices R_{ij} which are parameterized by two real parameters θ_{ij}

and ϕ_{ij} are defined as N-dimensional identity matrices whose four entries at positions given as $(row, column)$ are replaced with given entries as follows:

$$\begin{bmatrix} (i,i) & (i,j) \\ (j,i) & (j,j) \end{bmatrix} \mapsto \begin{bmatrix} e^{i\phi_{ij}} \cos(\theta_{ij}) & -e^{i\phi_{ij}} \sin(\theta_{ij}) \\ \sin(\theta_{ij}) & \cos(\theta_{ij}) \end{bmatrix} \tag{2.28}$$

By reordering and grouping rotational matrices as shown in [JSD+17, p. 4], the unitary matrix W with even capacity L can also be written as:

$$W = D * F_A^{(1)} * F_B^{(2)} * F_A^{(3)} * F_B^{(4)} * \ldots * F_B^{(L)} \tag{2.29}$$

Whenever the capacity L matches the dimension N of the unitary matrix W, this expression spans the entire space of all unitary matrices. Whenever the capacity L is smaller than the dimension N of the unitary matrix W, this expression spans a subspace of the space of all unitary matrices. The matrices $F_A^{(l)}$ and $F_B^{(l)}$ are constructed as follows where superscript (l) denotes different instances of the same type of rotational matrices when the subscript matches:

$$F_A^{(l)} = R_{1,2}^{(l)} * R_{3,4}^{(l)} * R_{5,6}^{(l)} * \ldots * R_{N/2-1,N/2}^{(l)} \tag{2.30}$$

$$F_B^{(l)} = R_{2,3}^{(l)} * R_{4,5}^{(l)} * R_{6,7}^{(l)} * \ldots * R_{N/2-2,N/2-1}^{(l)} \tag{2.31}$$

Furthermore, each matrix F of the above two types is a general rotational matrix, and its mapping performed on a vector x can also be written as [JSD+17, p. 4]:

$$F * x = v_1 *_{ew} x + v_2 *_{ew} permute(x) \tag{2.32}$$

The vectors v_1 and v_2 are computable from the parameters θ_{ij} and ϕ_{ij} that are used to parameterize the rotational matrices R_{ij} that build the matrix F. The permutation given by the function *permute* is fixed and set only at the machine learning model's first instantiation. The formula used to generate both vectors v_1 and v_2 is given under [JSD+17, p. 4]. This way of applying the mapping of the F matrices to the input vector avoids matrix multiplications and uses element-wise multiplications and permutation operations. It is an efficient way to parameterize unitary matrices. As the output vector of this machine learning model is complex, the real part of the output was used for further processing as this was also done in benchmarks from the official repository of the paper [JSD+17] which can be found under the URL `https://github.com/jingli9111/EUNN-tensorflow/blob/master/copying_task.py`. This model also has a configurable hidden vector size, which must be even, and was picked to 128 throughout the thesis. The capacity L was always set to 16. Therefore the matrix W is parameterized as a partial-space unitary matrix. As the output vector size has the be variable, the real part of the model's output vector was then fed to a dense layer to achieve the correct output vector dimension. The Unitary RNN model implementation used in this thesis is exposed under the `get_unitary_rnn_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/unitary_rnn.py`.

2.9 Matrix Exponential Unitary RNN

This machine learning model is an original contribution and a variant of the Unitary RNN architecture introduced in Section 2.8. Therefore, it is a discrete-time recurrent neural network model with the same model function as specified in Equation (2.26) and augments the architecture in various ways. Only the differences between the two architectures will be listed. First, the option to use a trainable initial hidden state vector was added to the architecture, which is initialized to the all-zero vector. Furthermore, there was an option added to use an augmented input for the model. This augmented input consists of the ordinary input vector's concatenation x_k per time step with its 1D discrete Fourier transform given by $FFT(x_k)$. As problems in the signal and system theory domain are either easier to solve in the time or the frequency domain, this feature may help make better predictions in some tasks. Moreover, the DFT matrix used to convert a time-domain vector to the frequency domain is also a unitary matrix, which preserves the input vector's energy and is, therefore, a good fit for this architecture. Both described features are disabled during benchmarking this model, as they showed no substantial decrease in the final test loss. Another difference to the Unitary RNN architecture introduced in Section 2.8 is the output vector's construction with the required size. As the imaginary part of the hidden state vector may also convey useful information, the approach from [ASB16, p. 4] was used in the implementation to construct the output vector. With this method, the final output vector is constructed by passing a concatenated vector consisting of the real and imaginary part of the hidden state vector, which is now solely real through a dense layer to get the correct output vector dimension. The last difference is the unitary matrix W's parametrization used in Equation (2.26). As presented in Section 2.8, the parametrization is quite involved, and therefore the new way of parameterizing the unitary matrix is using an approximated matrix exponential. Any unitary matrix W of dimension $N \times N$ can be written as the matrix exponential of a skew-Hermitian matrix A of dimension $N \times N$ as $W = e^A$. The problem is therefore reduced to parameterizing a skew-Hermitian matrix A. This matrix exponential is the matrix generalization of $|e^j| = 1$ in the scalar case where j is an imaginary number. The approximated matrix exponential implementation used for this model is exposed under the function `tf.linalg.expm` in the Tensorflow library [AAB+15] which uses Padé approximation as desribed in [AMH09]. The fundamental idea is that $e^A = (e^{2^{-s}A})^{2^s} \approx (r_m(2^{-s}A))^{2^s}$ where $r_m(X)$ is the $[m/m]$ Padé approximant to e^X and the non-negative integers m and s are to be chosen [AMH09, p. 1]. An approximation is needed as the matrix exponential e^A is defined by an infinite sum as follows:

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!} \tag{2.33}$$

A skew-Hermitian matrix A fulfills $A^H = -A$ which implies that the individual matrix entries fulfill $a_{ij} = -\overline{a_{ji}}$. This further implies that the diagonal entries of A are purely imaginary. Therefore, the square skew-Hermitian matrix A can be parameterized by only a lower triangular matrix T with complex entries as all other entries follow symmetry.

The diagonal entries in this matrix T can be parametrized with real parameters, therefore saving N parameters, but this optimization was not applied in the implementation. The skew-Hermitian matrix A can easily be constructed by the triangular matrix T by the following formula fulfilling all symmetry requirements:

$$A = T - T^H \tag{2.34}$$

As in Equation (2.34), only the diagonal entries overlap after the transposition, and the diagonal entries will be purely imaginary as the real parts will cancel themselves. All other entries follow the previously described symmetry. In this model's implementation, the matrix T was parameterized by a vector v of size $N * (N + 1)/2$, which equals the number of all non-zero elements in T. This vector v was then converted to a triangular matrix by filling a triangular matrix with all the values from T. With this method, any lower triangular matrix T can be constructed, from which any skew-hermitian matrix A can be constructed, from which any unitary matrix W can be computed by using the matrix exponential. This parameterization allows parameterizing the full-space of unitary matrices. If a partial-space parametrization is favored to reduce the model's parameter count, there is a capacity measure c available in the model's implementation, which should fulfill $0 \leq c \leq 1$. With this, only the first $\lfloor c * N * (N + 1)/2 \rfloor$ entries of the vector v will be trainable, and the remaining entries will be filled up with zeros. The benchmarked model had a hidden vector size of 128 and the capacity measure c set to 1. Therefore the entire space of unitary matrices was parameterizable. The Matrix Exponential Unitary RNN model implementation used in this thesis is exposed under the `get_matrix_exponential_unitary_rnn_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/matrix_exponential_unitary_rnn.py`.

2.10 Unitary NCP

The Unitary NCP model is a novel discrete-time machine learning model that combines the Unitary RNN model introduced in Section 2.8 and the Neural Circuit Policies model introduced in Section 2.7, just like the ODE-LSTM model combines the LSTM and the CT-RNN architecture. This combination, however, is not as tightly coupled as the ODE-LSTM architecture. This architecture uses a Unitary RNN to preprocess all input vectors of the input sequence x_k to an intermediate sequence by storing the real part of the hidden state vector at each step without feeding it through a dense layer afterward. This intermediate sequence is fed to the Neural Circuit Policies model, which treats it as its regular input sequence and maps it to the output vector sequence o_k. The Unitary NCP model function is given as follows where x_k is the input vector at time step k, $h_{k,unitary}$ is the hidden state vector of the Unitary RNN model, $h_{k,ncp}$ is the state vector

of the Neural Circuit Policies model and o_k is the output vector at time step k:

$$h_{k+1,unitary} = UnitaryRNN(x_{k+1}, h_{k,unitary}) \tag{2.35}$$
$$(h_{k+1,ncp}, o_{k+1}) = NCP(\mathrm{Re}\{h_{k+1,unitary}\}, h_{k,ncp}) \tag{2.36}$$

The $UnitaryRNN$ function is just a pointer to the corresponding model function described in Equation (2.26). The NCP model maps the input sequence to an output sequence as denoted by the function NCP. The model function is described in detail in Section 2.7. The architecture should combine the NCP model's excellent expressiveness and the Unitary RNN model's ability to capture long-term dependencies. The Unitary RNN was configured with a hidden state vector size of 32, and the capacity was set to 4. The NCP model uses 4 inter and command neurons and no recurrent command synapses, as the Unitary RNN should handle memory-related tasks. For details on both architectures, consult their sections. The Unitary NCP model implementation used in this thesis is exposed under the `get_unitary_ncp_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/unitary_ncp.py`.

2.11 Transformer

2.11.1 Introduction

The Transformer architecture introduced in [VSP+17] is no recurrent neural network architecture in the strict sense like the LSTM or the GRU architecture. It encodes its input sequence using an encoder, whose output is then decoded to the output sequence by a decoder. However, this model has a recurrence in its decoder part, as explained later. The problem of capturing long-term dependencies as described in Section 1.5 originates as the input time series is provided as one input vector per time step to the machine learning models. This nesting of functions results in deep computational graphs for longer time series, leading to vanishing or exploding gradients. The Transformer architecture overcomes this issue by considering the whole input vector sequence at a single time step for prediction. Attention mechanisms were used to deal with that much input data at a single time step, which means the model learns to weight the input data vectors according to their relevance in solving the required problem. Therefore, the computational graph becomes much shallower and easier to backpropagate through, overcoming the unwanted deep computational graphs and their problems. Before describing the exact structure of the encoder and decoder of the Transformer architecture, this visualization from the original paper summarizes the structure of the architecture as follows:

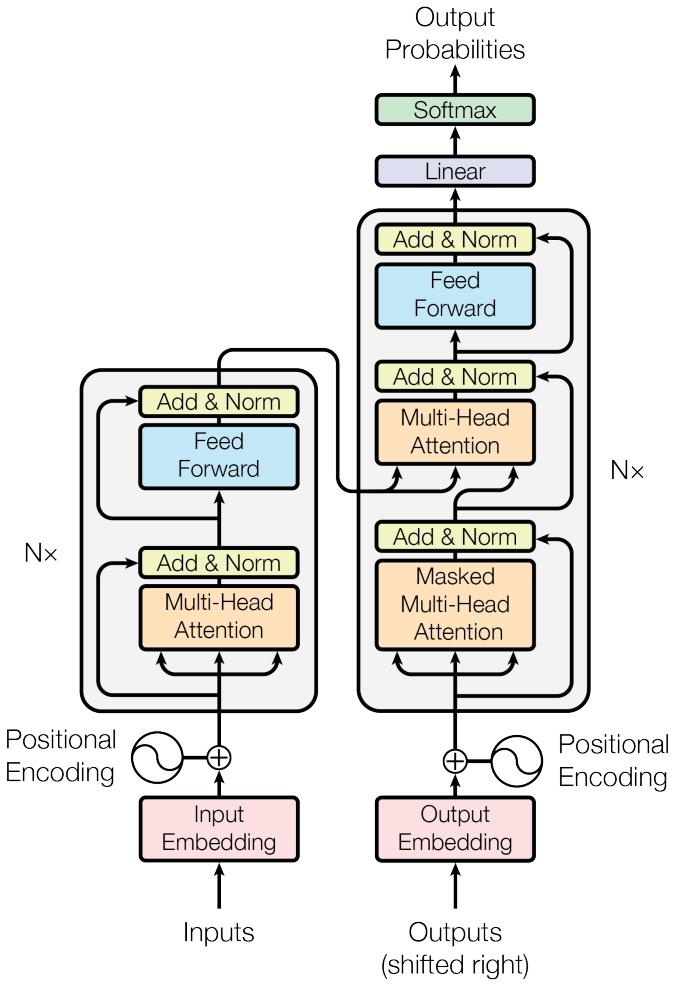

Figure 2.6: visualized Transformer architecture [VSP+17, p. 3]

2.11.2 Encoder

At first, the Transformer architecture passes its input vectors to the encoder. The encoder embeds its input vectors into vectors of length d_{model} (hyperparameter of the architecture) by passing them through a dense layer. As the input time series' embedded input vectors are not labeled with their corresponding time index k, the Transformer architecture adds a positional encoding vector to each embedded input vector. This positional encoding vector is only dependent on the absolute position in the input time series of the vector, and the hyperparameter d_{model} as described in-detail in [VSP+17, p. 6] and should allow the architecture to infer its absolute position in the input time series. After adding the positional embedding vectors, dropout with a configurable architecture-wide dropout rate $0 \leq r \leq 1$ is applied to all embedded input vectors. Dropout was introduced in [SHK+14] and randomly sets vector entries to 0 with a frequency of r, and vector entries not set to 0 are scaled up by $1/(1-r)$. According to [SHK+14, p. 1], this helps neural networks

to prevent overfitting. The output vectors after dropout are then fed to a configurable amount of encoder layers. Each encoder layer consists of two sub-layers, one multi-head attention layer, and one fully-connected feed-forward layer. Dropout is applied to each sub-layer output, whose result is then added to the input creating a residual connection [HZRS15]. Then layer normalization [BKH16] is applied to the sum of both which means the mean μ and variance σ^2 of all entries x in a vector are computed and these entries x are then mapped to $\frac{(x-\mu)}{\sigma+\varepsilon}$. The mapped vector entries are then normally distributed with mean 0 and variance 1. The ε in the formula is only added for numerical stability. This whole procedure of postprocessing the sub-layer output to the final outputs y given the inputs x can also be written in pseudocode [VSP+17, p. 3]:

$$y = LayerNormalization(x + Dropout(SubLayer(x))) \tag{2.37}$$

The multi-head attention layer requires three mandatory input arguments (queries, keys, and values) and an optional attention mask. There must be as many keys as values as they are used as a key-value-pair. The number of heads h, the dimension of the projected queries and keys d_k, and the dimension of the projected values d_v can be configured. All queries, keys, and values of the input arguments are mapped through three dense layers for queries, keys, and values to the projected query, key, and value vectors of the specified dimensions. This procedure is repeated h times with different dense layers but the same input. By writing the output vectors of this procedure in matrix form as queries Q, keys K and values V (vectors in rows), the scaled dot-product attention function output Y can be given as follows [VSP+17, p. 4]:

$$Y = softmax\left(\frac{Q * K^T}{\sqrt{d_k}}\right) * V \tag{2.38}$$

The matrix multiplication of Q and K^T corresponds to computing the scalar product of all combinations between query vectors and key vectors. The scalar product result of a single combination should describe how well the "question" or query matches the "answer" or key. If this result is high, it is said that the vectors attend to each other. These scalar products are then scaled, the attention mask is applied, and after that, the softmax function ($\frac{e^{x_i}}{\sum_j e^{x_j}}$) is applied to each row and row entry x_i in the corresponding matrix. The attention mask is responsible for setting the scalar products of certain query-key combinations to $-\infty$ before the softmax function is applied to prohibit information flow from the corresponding value vector. This normalization now results in a matrix where the entry in row i and column j corresponds to the attention weight between the query vector in row i and the key vector in row j. All attention weights of a single query vector to all possible key vectors add up to 1. The final output Y is then computed by doing a matrix multiplication of the attention weight matrix with the value matrix V, which equals computing a new representation for each query according to a weighted sum of value vectors. Scaled dot-product attention with multiple heads was also visualized by the original paper as follows:

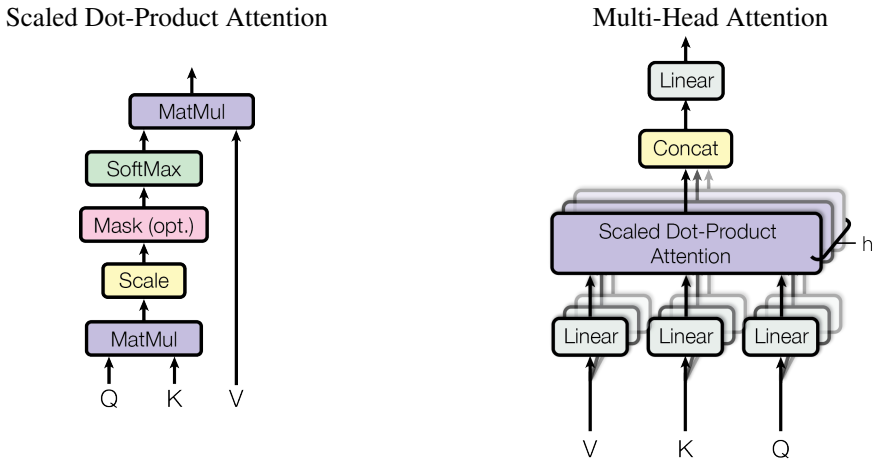

Figure 2.7: visualized scaled dot-product attention [VSP+17, p. 4]

Therefore, the corresponding key vector to a value vector describes how to access the value vector's information. This process can also be thought of as a continuous hash map where the key vector and value vector are the key-value-pairs, and the indexing is done with a query vector. As the query and key vector may never be exactly equal, the values are weighted according to their relevance. The output Y of the individual heads are then concatenated together and projected back to output vectors of dimension d_{model} with a dense layer. The encoder layer uses this multi-head attention mechanism as self-attention, which means query, key, and value vectors are just the same input vectors each encoder layer gets as input. This process can be thought of as exchanging information between all vectors. The second sub-layer in the encoder layer is the fully-connected feed-forward layer, which consists of a dense layer that maps the input vectors to size d_{ff} with a relu activation function and a second dense layer that maps the vector back to size d_{model} without an activation function. This process can be thought of as exchanging information within all vectors.

2.11.3 Decoder

The last encoder layer's output is then used in decoder layers of the same amount as encoder layers. The decoder gets a single start vector as input vector, which was picked to the all-one vector. All input vectors to the decoder are then embedded, positional encoded, and dropout is applied in the same fashion as for input vectors to the encoder. The self-attention sub-layer in a decoder layer sets the attention mask correspondingly such that input tokens to the decoder layer can only attend to other tokens up to the own token index ensuring the Transformer's auto-regressive property [VSP+17, p. 5]. The decoder and encoder layers' difference is that decoder layers have a third sub-layer function added between the encoder layer's two functions. The third sub-layer function also uses multi-head attention, but the key and value vectors are provided by the encoder's output, whereas the query vectors are provided from the previous sub-layer

output. This mechanism is called encoder-decoder attention, and it is responsible for transferring information from the input vector sequence to the output vector sequence. The last decoder layer's output vectors are then passed through a dense layer, which maps the outputs to the required dimension of *token_size*. The *token_amount* parameter determines how often the decoder architecture should be run. After a complete run of the decoder architecture, the output vector of size *token_size* corresponding to the last decoder input is concatenated to the list of all decoder inputs, and the whole decoder architecture is rerun, now with two or more input vectors for the decoder. These reruns lead to the recurrence of the Transformer model. The Transformer's output is then a flattened version of the decoder's output vectors, excluding the first start vector.

2.11.4 Implementation

The implementation used for benchmarking had *token_amount* set to 1 and *token_size* set to the required output vector size. The hyperparameter d_{model} was set to 16, h was set to 2, d_{ff} was set to 64, there were 2 encoder and decoder layers used and the dropout rate was set to 0. The dimension d_k and d_v were always equal to d_{model} in the benchmarked implementation. The Transformer model implementation used in this thesis is exposed under the `get_transformer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/transformer.py`.

2.12 Recurrent Network Augmented Transformer

The Recurrent Network Augmented Transformer architecture is a novel contribution similar to the Transformer architecture introduced in Section 2.11. The only difference is that it uses a slightly changed attention mechanism. As given in Equation (2.38), the final output Y is constructed by matrix multiplication of the attention weights and the value matrix V, which means a new representation for the query vectors is computed by summing up the weighted value vectors per query vector. The idea now is that instead of summing up the weighted value vectors by ordinary summation, maybe the use of a recurrent neural network to accumulate the information present in the weighted value vectors can increase the Transformer architecture's expressivity. Furthermore, the incorporated RNN can directly use positional information, and the sum function is easy to learn for any RNN architecture, which has a similar model function to the one defined in Equation (1.7). It just needs to learn that the W matrix should be an identity matrix. A different RNN with different weights for each head was used. The implementation used to benchmark the architecture uses the LSTM architecture for the described RNNs. The difference in hyperparameters to the Transformer architecture is that this architecture sets d_{model} to 8, the number of heads h to 1, d_{ff} to 32 and the number of encoder and decoder layers to 1. The Recurrent

Network Augmented Transformer model implementation used in this thesis is exposed under the `get_recurrent_network_augmented_transformer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/recurrent_network_augmented_transformer.py`.

2.13 Recurrent Network Attention Transformer

The Recurrent Network Augmented Transformer architecture is a novel contribution similar to the Transformer architecture introduced in Section 2.11. The only difference is that it uses an entirely new attention mechanism called recurrent network attention, which uses recurrent neural networks. As in the Transformer architecture, this attention mechanism gets the four arguments: queries, keys, values, and an attention mask. The attention mask and keys argument is not used in this mechanism. The new representation of each query vector (the output of the attention mechanism) is computed by building a sequence of a single query vector concatenated with all value vectors. This sequence is as long as the amount of value vectors given in the values matrix from the argument, and each concatenated vector in this sequence has a size of $2 * d_{model}$. Computing the new representation is then done by passing this sequence through an RNN and using the output after the last input vector for further processing. Of course, also this attention mechanism supports multiple heads by mapping the same sequence with multiple RNNs using different weights. The results are then concatenated together and projected back to vectors of size d_{model} with a dense layer to get this attention mechanism's output vectors. The implementation used to benchmark the architecture uses the Unitary RNN architecture for the described RNNs. The difference in hyperparameters to the Transformer architecture is that this architecture sets d_{model} to 8, the number of heads h to 1, d_{ff} to 32 and the number of encoder and decoder layers to 1. The Recurrent Network Attention Transformer model implementation used in this thesis is exposed under the `get_recurrent_network_attention_transformer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/recurrent_network_attention.py`.

2.14 Memory Augmented Transformer

The Memory Augmented Transformer architecture is a novel contribution and a discrete-time recurrent neural network architecture incorporating a Transformer model and external memory. This model is, therefore, also a MANN (memory-augmented neural network). The external memory M represents the model's state and has a configurable

number of rows r and a configurable number of columns c. All memory fields are prefilled with a small value set to 10^{-6}. There are two embedding dense layers defined, the input embedding IE to embed the current step input and the memory embedding ME to embed each memory row of the external memory M. Both embeddings map the input vectors to size *embedding_size* and are computed at each time step. The resulting vectors are then concatenated together to a vector sequence of length $r + 1$. Then positional encoding vectors (denoted as PE in matrix form) are added to each vector in this sequence as described in Section 2.11.2. Moreover, dropout with rate dr is further applied on this vector sequence, which is then fed through a single encoder layer with the functionality described in Section 2.11.2. The first vector of the encoder layer vector outputs is used to build the output y of the model by projecting it to the required output vector size through the dense output layer DOL. All other r output vectors of the encoder layers are then projected with the memory control dense layer $MCDL$ to a memory control signal vector per memory row of size $1 + c$ (denoted as MCS in matrix form). This vector's first entry is called the enable signal and is used to activate the memory and write the remaining c entries to the memory. It can also deactivate the memory to mask the remaining c vector entries away, resulting in keeping the current memory state. This masking was done by feeding the enable signal through a *sigmoid* function in the positive and negated form (both results add up to 1), which are then used to weigh the new and old memory state. At each time step, the following model function is executed with the input denoted as i_t and the output denoted as y_t:

$$z_t = Dropout(concat(ME(M_{k-1}), IE(i_k)) + PE) \tag{2.39}$$

$$e_t = EncoderLayer(z_t) \tag{2.40}$$

$$o_t = DOL(e_t[0]) \tag{2.41}$$

$$MCS_t = MCDL(e_t[1..r]) \tag{2.42}$$

$$M_k = sigmoid(-MCS_t[:, 0]) * M_{k-1} + sigmoid(MCS_t[:, 0]) * MSC_t[:, 1..r] \tag{2.43}$$

By incorporating the encoder layer, the architecture can freely choose how many memory rows it wants to read in a single time step, as the corresponding attention weights can determine this. The architecture can focus on the memory contents or the exact location as a positional encoding was used with the attention mechanism. Furthermore, using the memory enable signals for each memory row, the architecture may also freely determine how many memory rows it wants to write to in a single time step. This architecture tries to separate computation and memory just like personal computers do. The CPU equivalent in this architecture is the encoder layer, including all the dense layers, and the external memory is responsible for persisting information. The benchmarked implementation had r and c set to 16, the embedding size set to 32, and the number of heads in the encoder layer set to 2. Furthermore, it had the encoder layer's feed-forward size d_{ff} set to 128, and the dropout rate dr, as well as the encoder layer's dropout rate in the encoder layer, set to 0. The Memory Augmented Transformer model implementation used in this thesis is exposed under the `get_memory_augmented_transformer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNe`

tworkArena/blob/master/experiments/models/model_factory.py. The in-detail implementation is provided in the file https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/memory_augmented_transformer.py.

2.15 Differentiable Neural Computer (DNC)

The DNC is a discrete-time memory-augmented recurrent neural network architecture that consists of a controller, read and write heads, and an external memory M that is not parameterized and may have arbitrary size. Furthermore, the external memory was structured into N rows, where each memory row contains a vector of length C. The architecture was introduced by [GWR+16] and is an enhancement to the NTM (Neural Turing Machine) architecture first proposed [GWD14]. The NTM introduced differentiable read and write functions that act to a greater or lesser degree with all rows in the memory [GWD14, p. 5]. The degree at time step t for row i is determined by the weighting $w_t(i)$ that the corresponding read or write head emits. The weighting is similar to the attention weights used in the Transformer architecture introduced in Section 2.11. They all lie between 0 and 1, and the weightings for all rows add up to 1. The read vector r_t returned by the read function of a single read head is given as follows where $M_t(i)$ denotes row i in the memory at time step t [GWR+16, p. 1]:

$$r_t = \sum_i w_t(i) * M_t(i) \tag{2.44}$$

The write function as executed by a single write head is given as follows where e_t is the erase vector of length C whose elements all lie between 0 and 1 and a_t is the add vector of length C at time step t (both vectors get emitted by the write head additionally to the weighting w_t, 1 represents the all-one vector) [GWR+16, p. 1]:

$$M_t(i) = M_{t-1}(i) *_{ew} (1 - w_t(i) * e_t) + w_t(i) * a_t \tag{2.45}$$

The weightings w_t emitted by the heads are generated by combining three different attention mechanisms. The first mechanism is content lookup, which is based on a key vector k_t emitted by the corresponding head. The cosine similarity measure between each memory row $M_t(i)$ and the single key vector k_t is then computed and normalized such that it forms a probability distribution over all memory rows that is incorporated to compute the final weighting. The second attention mechanism uses a temporal link matrix of dimension $N \times N$, which records transitions between consecutively written locations. The entry $L[i, j]$ is close to 1 if i was the location written next time step after j and is close to 0 otherwise. This matrix smoothly shifts the focus of a given weighting w_t to the locations written after those or written previous those emphasized in w_t. The third attention mechanism keeps track of the usage u_t of each memory row, which lies between 0 and 1. The usage u_t is increased by writing and decreased by reading to the memory row. With this mechanism, write heads' weightings can favor memory rows with low usage to store new information without overwriting other existing

information in memory rows with high usage. All three of these attention mechanisms are described in [GWR+16, p. 1-2] and their exact interplay to create the weightings w_t for each head is described in detail in [GWR+16, p. 7-8]. All the DNC architecture operations are controlled by either a recurrent neural network or a feed-forward neural network controller, which gets the current step inputs x_k and all the read vectors r_{t-1} from all read heads as shown in Equation (2.44) as inputs. They are provided as a single concatenated input vector. It should be noted that the read vectors are computed for the memory at time step $t-1$, which makes sense when looking at the output of the controller. The controller's output vector at time step t is the output vector o_t of the required size and an interface vector ξ_t. This interface vector ξ_t provides all information to the read and write heads, such that they can execute their read function as described in Equation (2.44) and their write function as described in Equation (2.45). At first, the write heads are updating the memory, and then the read heads compute their read function and return their read vectors r_t. These read vectors are again provided to the controller at time step $t+1$. This architecture was also visualized in the original paper as follows:

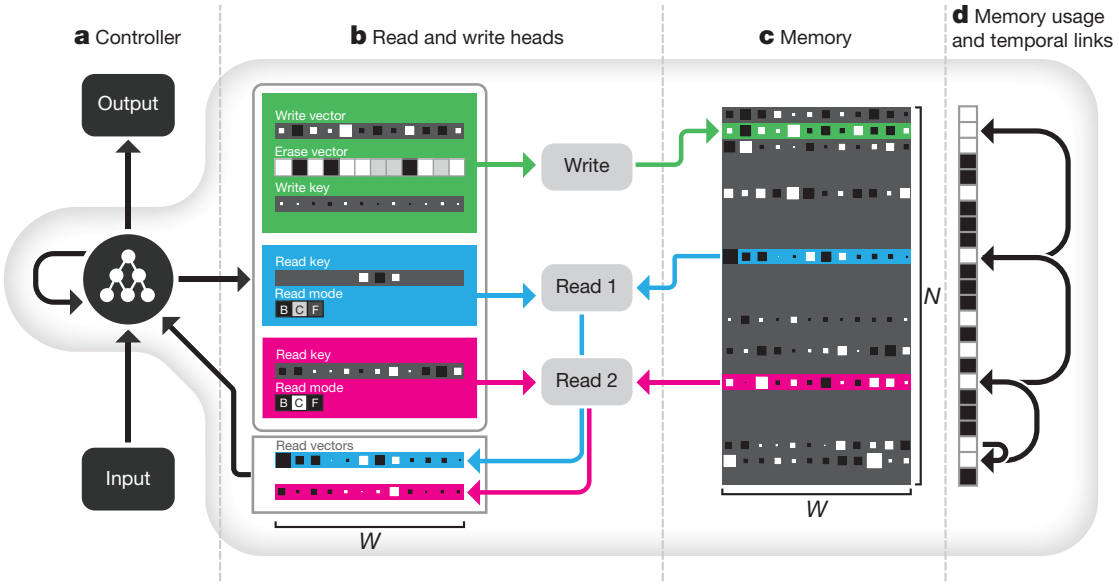

Figure 2.8: visualized DNC architecture [GWR+16, p. 2]

Recurrent network controllers are generally preferred as they complement the external memory, just like the internal registers of a CPU complement the RAM. The implementation used for benchmarking uses the open-source DNC implementation distributed under `https://github.com/willsq/tf-DNC/tree/master/dnc`. The controller was configured as an LSTM with a hidden state vector size of 64. There were 2 read heads and 1 write head used. Even if the architecture supports a memory of infinite size, the memory shape must be fixed for implementation. The number of memory rows N was set to 16,

and the size of each vector C was set to 8. Each memory entry's initial state and each first read vector entry's initial state is picked to 10^{-6}. The DNC model implementation used in this thesis is exposed under the `get_differentiable_neural_computer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/differentiable_neural_computer.py`.

2.16 Memory Cell

The Memory Cell is a continuous-time recurrent neural network architecture consisting of two LTC neurons described in Section 2.7 without the input and output mapping using dense layers. It is a proof-of-concept implementation and tries to build an LTC network [HLA+20] to capture long-term dependencies in time series, namely a single memory bit. For details on how inputs are provided and which outputs are expected, please consult Section 3.7. The model has six synapses in total. Each neuron had 3 incoming synapses. This model can only be used with input vectors and expected output vectors of size 2. The input vector entries are the two scalar inputs passed on to the two neurons with a synaptic activation. As this model is built out of two neurons, this model's output vector size is fixed to 2, and the output vector contains both neurons' potentials. Each neuron has an input synapse, an inhibitory synapse, and a recurrent synapse. This architecture can be visualized using circles for neurons, solid arrows for chemical synapses, and dashed arrows for leakage currents as follows:

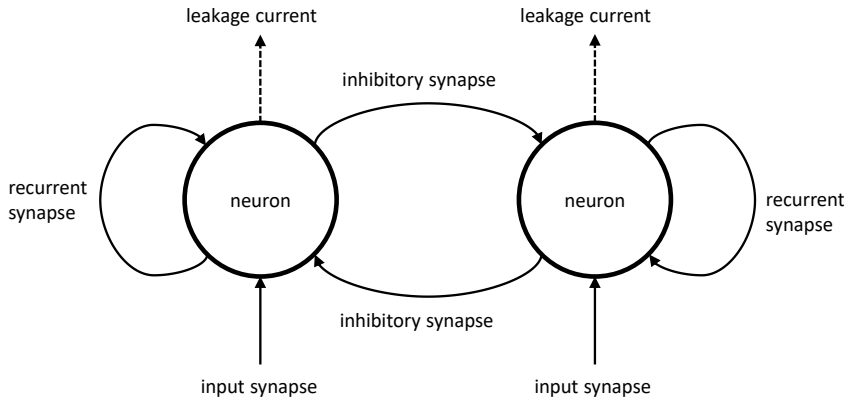

Figure 2.9: visualized Memory Cell architecture

Each neuron's three synapses visible in Figure 2.9 share the same parameters as the Memory Cell model should employ similar mechanisms when storing a 0 or a 1 bit. The behavior should be symmetric, and the decision which bit to store should only be dependent on the current input. The current memory content of this architecture is encoded in the potentials of both neurons. If the first neuron has high potential (≈ 1) and the second neuron low potential (≈ 0), the bit 1 is currently stored, and if the second neuron has high potential (≈ 1) and the first neuron low potential (≈ 0) the bit 0 is currently stored. The purpose of the input synapse, which connects the input vector entry with the synapse, is to supply each neuron with a large input current to increase its potential to ≈ 1 if the input vector entry to the corresponding neuron is ≈ 1, too. It is only applicable that at most one neuron gets a large input vector entry (≈ 1) at a single time step. This single large vector entry leads to a switch of the stored memory bit or the current memory state's persistence. It can also be the case that both neurons receive a small input vector entry (≈ 0). Therefore both neurons receive little to no input current, and the memory state is kept as ensured by the inhibitory and recurrent synapse. The inhibitory synapse that connects a neuron with the other neuron is responsible for suppressing the other neuron when a neuron itself has high potential. Therefore it ensures that the second neuron's potential is kept low such that only one neuron can have a high potential. The recurrent synapse that connects a neuron with itself is responsible that a single neuron keeps its potential if the other neuron does not inhibit it. Three synapses per neuron were at least necessary for a working Memory Cell architecture. The theoretical lower bound may be two synapses per neuron, as only an input synapse and a communication synapse that handles communication between

the two neurons are needed. The communication synapse would be connected from one neuron to the other neuron and must fulfill the recurrent and inhibitory synapse tasks. However, in this scenario, with only two synapses assuming their proper functionality, the communication synapse will have to supply a negative current to inhibit the other neuron at a memory switch. Furthermore, when there is no memory switch, the same communication synapse must provide a positive current to a neuron with high potential to keep its state as there is a leakage current. The sign of a synaptic current $I_{syn,ji}$ as computed in Equation (2.25) is determined by the sign of $E_{ji} - V_i(t)$. The postsynaptic potential $V_i(t)$ may be ≈ 1 in both cases, therefore the different sign of $E_{ji} - V_i(t)$ cannot be determined by the parameter E_{ji} which yields a contradiction to the assumption of proper functionality. Each synapse from neuron j to neuron i has four parameters as shown in Equation (2.25): the maximum conductance $G_{syn,ji}$, the mean conductance potential μ_{ji}, the steepness of the conductance transition σ_{ji} and the target potential E_{ji}. The steepness of the conductance transition was fixed to 100 for all synapses in the implemented model. All neurons' conductances, including the leakage conductance G_{leak}, were parameters and therefore learned. The target potentials for the leakage current and the inhibitory synapse were fixed to 0, the target potentials of the recurrent synapse and the input synapse were parameters and, therefore, also learned. The mean conductance potential of the input synapse was fixed to 0.5, the mean conductance potential of the recurrent and inhibitory synapse was a parameter of the model. The capacitance of all neurons was fixed to 1 and the fixed time input t per time step used to integrate the state derivative in a continuous-time model was also a learned parameter. The state's ODE was unrolled two times per time step and was solved using the explicit Euler method. This unrolling is necessary such that the input currents can propagate to each of the two neurons in the first unroll step, and the inhibitory synapse currents can propagate in the second unroll step in case of a memory switch. Therefore, this architecture has 9 learnable parameters. Validation of the model was performed using the Cell Benchmark introduced in Section 3.7. The first neuron's initial state was 0, and the second neuron's initial state was 1. The Memory Cell model implementation used in this thesis is exposed under the `get_memory_cell_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/memory_cell.py`.

CHAPTER 3

Benchmarks

3.1 Benchmark Framework

3.1.1 Setup

A single code base to run and evaluate the diverse set of benchmarks and models was inevitable. Otherwise, the whole project would have been unmanageable. As the implementation of all models occurred in the `Python` programming language [VRD09] using the framework Tensorflow [AAB+15], also the benchmark framework used the same set of tools. Therefore, a benchmark base class was created in the file `benchmark.py`, which is available under the URL `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/benchmark.py`. The creation of a new benchmark is as easy as subclassing the benchmark base class `Benchmark`. For instructions on how to call the newly created class, please consulate the `README.md` file given under the URL `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/README.md`. After subclassing the base class, the new class has to correctly call the superclass constructor and overwrite the abstract method `get_data_and_output_size`. Furthermore, the new benchmark's name should be added to the `BENCHMARK_NAMES` list. The superclass constructor only has two arguments: `name` and `parser_configs`. The first argument is just the name of the new benchmark passed as a string. The second argument should be a tuple of individual parser configs. A parser config is itself a tuple consisting of the argument name, the argument default value, and the argument type. This argument determines which values should be settable and usable when calling the benchmark from the command line. There are at least three parser configs required that set the loss name, the loss config, and the metric name. A sample `parser_configs` argument would be: `(('--loss_name', 'SparseCategoricalCrossentropy', str), ('--loss_config', 'from_logits': True, dict), ('--metric_name', 'SparseCategoricalAccuracy', str))`. If loss config or metric name does not ap-

ply to the benchmark, set the default loss config to `{}` or the default metric name to `''`. Furthermore, if the benchmark needs additional parameters, extend the `parser_configs` parameter also to include the desired command-line arguments. All individual benchmark implementations use this feature. After calling the superclass constructor, all command-line arguments configured through `parser_configs` will be available by their names as properties of `self.args` without the double hyphen. For example the loss name can be accessed by `self.args.loss_name`. If some parameters were set through the command line, they would have the corresponding value. Otherwise, the configured default values will be applied. After that, the benchmark base class will create paths for some required directories. There are five directories required during benchmark execution: a saved model directory (will be created to save the models together with their best weights during training), a TensorBoard directory (will be created to save TensorBoard logs for eventual later evaluation), a supplementary data directory (already present in the repo to pass input data to the benchmark), a result directory (will be created to save CSV files with relevant information about the training process) and a visualization directory (will be created to save visualizations created after each training of a model). All these paths start in the root folder of the repository called `NeuralNetworkArena`. The structure of how these paths continue is the same for all five kinds of folders. For the next step in path creation, the required folder's name will be appended to the root folder. These names can be passed as a command argument when calling the individual benchmark classes. For a more detailed description of these command-line parameters, call an implemented benchmark class with the `--h` command line parameter as described in the `README.md` file. The name of the individual benchmarks is further added to the path, such that each benchmark has its own five subfolders. Then the benchmark base class calls its `get_data_and_output_size` method that the subclass should have implemented. The function should return a tuple of inputs, a tuple of expected outputs, and an output vector size of the machine learning model. The input and output tuple should only contain NumPy arrays [HMvdW+20]. The output tuple must have a size of precisely one. The input tuple must have a size of at least one. The benchmark base class also has support for time inputs to the models. Please make sure that the time input is the last entry in the input tuple. There is also the command line argument called `use_time_input`. If the model should use time input, make sure that this argument is set to true. Otherwise, if the input tuple has a dimension larger than one, the last entry will be discarded from the input tuple, as it is assumed to be the time input. The benchmark suite works currently only for benchmarks that provide time series input data and only expect a model output after the last input data in the time series. For people familiar with the Tensorflow framework [AAB+15] this is equivalent to setting `return_sequences=False` in an RNN model. All input arrays in the input tuple should have the shape `(SAMPLE_AMOUNT, SEQUENCE_LENGTH, INPUT_DIMENSION)`. Of course, the input dimension can vary between different inputs. Time data should have an input dimension of one. The single output array present in the output tuple should have the shape `(SAMPLE_AMOUNT, OUTPUT_DIMENSION)`. The sample amount should match between input and output data to be valid input

to the benchmark framework. The framework will check all the constraints on the shapes, and then all individual samples are shuffled such that corresponding input and output data are at the same indices in their arrays. Then tensors are created with the same shape as the input tuple's inputs, excluding the first dimension that denotes the sample amount. These are required to use later the Functional API of the Tensorflow framework [AAB+15]. They are created by specifying a fixed batch size, which helps the machine learning framework optimize the corresponding model's computational graph. The default batch size is set to 128 and can be changed by a command-line parameter. After that, the whole samples are divided into the test, validation, and training samples. The amount of test and validation samples can be set via command line parameters, which default to 10% each. It is ensured that each sample set is exactly divisible by the batch size, as the computational graph was optimized by only allowing inputs of a fixed batch size as described above. After all the setup work is done, the folder paths to the result, the saved model, and the TensorBoard directory will be augmented with the model name currently under test and passed via a command-line parameter. The TensorBoard directory for that model will then be deleted, as each training run creates a significant amount of log files. After that, the TensorBoard, the result, the saved model, and visualization directory will be created if they do not already exist. Then it will be checked if the passed model name is present in the list constant `MODEL_ARGUMENTS` in the file `model_factory.py`. When this check is passed, the benchmark framework either loads a saved model with the corresponding model name or creates a new one using the model output functions in the previously described model factory depending on the command line parameter `use_saved_model`. These output functions get an output vector size and the tensor inputs and create an output tensor that contains all the information about the operations in between. The Tensorflow [AAB+15] Functional API can be incorporated to create a machine learning model by knowing the input and the output tensors. If the model is newly created and not loaded from a saved one, the model is also compiled using a customizable optimizer, learning rate, loss, loss config, and metric. Command-line parameters can change these. The default optimizer and learning rate used throughout all benchmarks in this thesis are the Adam optimizer [KB17] and a learning rate of 10^{-3}. The three remaining parameters also discussed in the previous subsection must be passed such that it is conforming with the requirements of the functions `tf.keras.optimizers.get` and `tf.keras.losses.get`. A debug mode can also be enabled via the command line, which puts the newly created model in eager execution mode, making it easier to debug the model. Furthermore, the model will be called on a single batch of inputs without invoking the model's `fit` method. This invocation happens only in debug mode. In any case, a model ready to train should now have been constructed, and all the model characteristics, including input and output shape, will then be printed to the command line enabling to check if all the dimensions match the expectations.

3.1.2 Training

After printing the model's available information to the command line, a UNIX timestamp is retrieved from the system to track the total training duration. Then the training is ultimately started by invoking the model's `fit` method. This method takes the training and validation sample set, the batch size, the number of epochs, and a tuple of callbacks as arguments. The number of epochs is configurable via the command line, but the default value of 128 is used throughout the thesis. The `fit` method calls the machine learning model function for each batch of inputs in the training sample set. After that, the model is validated on the validation sample set. Validation means the loss function is computed only on validation data, which is data that the model has never seen before. Validating the model should help to determine how well the model will perform on actual test data, which is also data that the model has never seen before. If the loss function results for training and validation data are similar, it is said that the model generalizes well. When the validation step is finished, the training loop proceeds with the next epoch. Therefore, it starts the same cycle again by providing the first batch of inputs from the training sample set. This cycle is repeated as often as the set value of the epochs. The callbacks are invoked after each completed epoch. There were five callbacks added: a ModelCheckpoint callback (saves the model with the best validation loss), an EarlyStopping callback (terminates training if the validation loss has not improved for a configurable number of epochs), a TerminateOnNan callback (terminates the training when a nan loss is encountered), a ReduceLROnPlateau callback (multiplies the learning rate by a configurable factor after no improvement of the validation loss for a configurable number of epochs) and a TensorBoard callback (saves TensorBoard log data for eventual later inspection). The default number of epochs used in this thesis for the EarlyStopping callback is 5. Another necessary callback is the TerminateOnNan callback, which terminates the training loop if the loss evaluates to nan. This nan return value can, for example, happen when the loss function diverges towards infinity, therefore, if the exploding gradient problem appears. It may also be the case that there is a division through zero somewhere in the computational graph, which may also lead to a nan loss. The term nan just stands for `not a number`. As all benchmarked models are trained until convergence in this thesis, the ReduceLROnPlateau callback is especially important. The corresponding default parameters are a learning rate factor of 10^{-1} and a default number of epochs equal to 2, both of which are used throughout all benchmark invocations. The EarlyStopping and the ReduceLROnPlateau do not see an improvement if the absolute change in the validation loss is less than 0.0001. This minimum delta can also be configured via the command line, but this thesis uses the default value throughout all benchmarks. Furthermore, all these parameters are configurable by passing alternative values in the command line. After the training loop has terminated, another UNIX timestamp is taken to compute the total training duration.

3.1.3 Evaluation

The model is then evaluated using the parameters that led to the smallest validation loss during the whole training loop. Evaluation means that the model function is applied to the test sample set inputs, and the resulting loss function result on that inputs is saved. The created model also provides an `evaluate` function, which takes the test sample set a batch size and another callback tuple as arguments. The only callback passed in the tuple is the TensorBoard callback already used in the `fit` method invocation.

3.1.4 Data Processing

The return values of the `fit` and `evaluate` method invocations now contain information about the means of the loss and metric function results. These results are available for the training, validation, and test sample set. The arithmetic means for the training and validation sample set are available for each training epoch together with the currently applied learning rate. All information is automatically accumulated in a single CSV file per model for the training and the testing process. All models' testing results are also merged in a single CSV containing all model results for a single benchmark. Data generated during training is automatically visualized by the benchmark base class and presented in Chapter 4 that discusses the benchmark results in more detail. Of course, all generated files will be stored in their respective directories.

3.2 Activity Benchmark

As described in the benchmark base class, all benchmarks feature time series data where the model output is only used after the last time step to compute the loss function. This benchmark uses a slightly modified person activity recognition dataset from the UCI repository [DG17]. The mentioned dataset was distributed under the `https://archive.ics.uci.edu/ml/machine-learning-databases/00196/ConfLongDemo_JSI.txt`. The target function to learn is to map a sequence of measurements from four inertial sensors worn on the person's arms and feet to an activity classification. This benchmark should test a model's capability to model dynamical physical systems and understand what motion patterns belong to what class. The ability to capture long-term dependencies is not tested with this benchmark, as the most recent input vectors should be enough to make good predictions. At each time step, only the single inertial sensor's measurement is presented as input to the model. The model can differ between the individual sensors as the modified dataset of person activity has a one-hot encoding to mark the sensor from which the current measurement is coming. All benchmarks feature an additional time input, where the time interval since the last input is passed on to the model if the feature is activated. However, this thesis has not used an additional time input for any benchmark. All the measurements used for this dataset were stored in the file `activity.csv` located in the supplementary data folder described in the benchmark framework section. The dataset is annotated with an activity classification for each time step. However, this benchmark only requires the

model to predict the classification corresponding to the last measurement data received. As the benchmark is a classification task, a categorical cross-entropy loss was used that was computed from the output logits of the model. A categorical accuracy metric is used to judge better how accurately the model predicts the activity class annotation corresponding to the last measurement input. Each model had an output vector size of seven, as there were seven different activity classes with their respective indices in brackets: lying (0), sitting on a chair (1), standing up (2), walking (3), falling (4), on all fours (5) and sitting on the ground (6). The processing of the UCI dataset was similarly done as in [LH20]. The benchmark had a configurable sequence length, maximum sample amount, and sample distance. For this thesis, a sequence length of 64, a maximum sample amount of 40000, and a sample distance of 4 were used. The sequence length means that each model gets a history of 64 measurements before predicting the activity corresponding to the last measurement. The maximum sample amount bounds the number of samples, and in the case of 40000 samples at maximum and a sample distance of 4, there were enough entries in the dataset file, so the benchmark was run with 40000 samples in total. The sample distance is the indices offset in the dataset file between two drawn sample sequences. A model will get a sequence of 64 input vectors of size seven that look like: $[0, 0, 0, 1, 4.3, 1.8, 0.9]$. The first four entries in that vector represent the one-hot encoding describing from which one of the four sensors the measurement data was taken. The remaining three entries contain the x, y, and z coordinate of the corresponding sensor. The required output vector has just one entry as it is just the index of the corresponding activity class with the mapping as described above. As this is a sparse class encoding, the framework has to extend this output value to a one-hot encoding to apply a cross-entropy loss between the extended one-hot encoding and our model's output vector after a softmax function was applied. The softmax function is necessary to convert the so-called output logits to an output probability for each class. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/activity_benchmark.py`.

3.3 Add Benchmark

This benchmark uses the same structure as the Add Benchmark introduced used in [ASB16]. Data for this benchmark is generated randomly at each instantiation of the benchmark. The target function to learn is adding two marked numbers in a much longer stream of numbers. At each time step, a number and a marker bit are presented as an input vector to the model. As in the Activity Benchmark from Section 3.2, the sequence length and the sample amount are also configurable. For all models, a sequence length of 100 and a sample amount of 40000 was used. As described above, the input vector has size two. The second entry is set to one only in one input vector of the first and last 50 input vectors. Their distribution is uniform across the whole first and second half of the time series. In all other input vectors, this second entry is set to zero. The first entry of all input vectors is filled with random numbers taken independently and

uniformly from the interval $[0, 1)$. A single input vector out of the 100 input vectors each model gets during the benchmark looks like $[0.5, 1]$. In this example, the random number is 0.5, and it is marked as the second entry is one. As described, there are only two marked numbers, and the expected output vector has size one and is simply the addition of both marked numbers. This benchmark simply uses the mean squared error loss function, as the smaller the mean square error is, the more similar the expected and the model output will be. Furthermore, there is no metric used in this benchmark. As this benchmark uses an increased sequence length of 100 and the error signal is only provided after the last input vector, the model will only learn this function when it can capture long-term dependencies. This condition means the model function must be designed so that the gradient does not vanish or explode during backpropagation through the model's function. These problems were discussed in detail in Section 1.5. When the model cannot capture these long-term dependencies and cannot store seen marked values in its state, it will be forced to learn the naive memory-less strategy of always predicting one. Predicting 1 will be the case in this strategy as the expectation of each unique number out of the two marked ones is 0.5, as they were drawn uniformly from the given interval. The addition of both expectation values reveals the output of the memory-less strategy. As also pointed out in [ASB16, p. 6], this naive strategy will lead to a mean squared error of $\frac{1}{6}$. This result can be verified as the mean squared error when predicting the mean equals the distribution's variance. As both random numbers were picked independently of each other, the random number sum's variance is just the sum of their variances. The distribution from which the random numbers are drawn has variance $\frac{1}{12}$. Therefore, adding this value to itself proves the mean square error of the memory-less strategy. For this benchmark, the model output vector size is simply one, as it should just contain the sum of both marked numbers. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/add_benchmark.py`.

3.4 Walker Benchmark

This benchmark evaluates how well a model can predict a dynamic, physical system's behavior. It was taken from [LH20]. The training data is acquired simulation data of the `Walker2d-v2` OpenAI gym [BCP+16] controlled by a pre-trained policy. The objective was to learn the MuJoCo physics engine's kinematic simulation [TET12] in an auto-regressive fashion using imitation learning. The simulation data was acquired from various training stages of the pre-trained policy (between 500 and 1200 Proximal Policy Optimization iterations) to increase the task difficulty. Furthermore, 1% of actions were overwritten by random actions. The simulation environment can be visualized as follows:

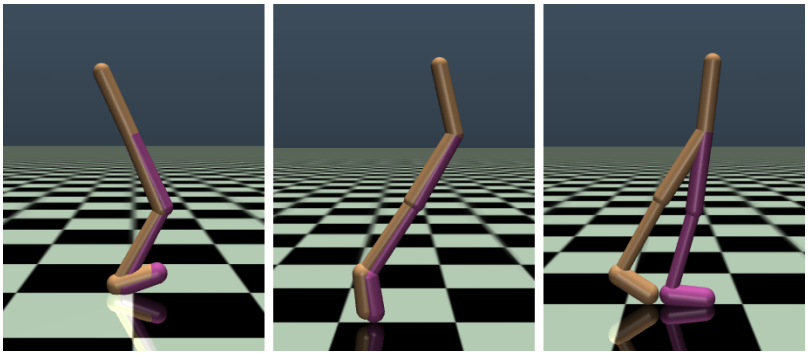

Figure 3.1: visualized `Walker2d-v2` OpenAI gym [LH20, p. 7]

Furthermore, the benchmark implements eventual frame-skips that would create an irregularly sampled time series. This feature was not used in this thesis as it covers only regularly sampled time series. If the model understands the dynamics guided by differential equations, it will produce accurate predictions. The ability to capture long-term dependencies is not tested with this benchmark, as the most recent input vectors should be enough to make good predictions. The benchmark had a configurable sequence length, a maximum sample amount, and a sample distance, just like the Activity Benchmark from Section 3.2. Throughout the thesis, a sample length of 64, a maximum sample amount of 40000, and a sample distance of 4 were used. All parameters have the same meaning as before. There was enough training data provided in `.npy` files by the creators of [LH20], therefore 40000 different samples were available that were partitioned in training, validation, and test samples. The acquired simulation data can be downloaded from `https://pub.ist.ac.at/~mlechner/datasets/walker.zip`. The input sequence consists of input vectors of size 17, which contains the physics engine's current state at this specific time step. These values represent the angles of the joints and the absolute position of the bipedal robot. The function to learn for this benchmark is to predict the physics engine's state in the next time step by giving the machine learning model history of the past 64 physic engine's states. Therefore, the model output vector size was set to 17, and the expected output data were also vectors of size 17. As both vectors have the same size and the more similar they are, the better the prediction is, a mean squared error loss was used. There was no metric used for this benchmark. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/walker_benchmark.py`.

3.5 Memory Benchmark

This benchmark evaluates how well a model can capture long-term dependencies by letting the model recall past seen categories exactly. It is a slightly changed version of the

copying memory problem described in [ASB16]. Input data of the benchmark input is randomly created at each invocation of the benchmark. There is a configurable memory length to test for, a configurable length of the sequence to memorize, a configurable number of categories, and a configurable number of randomly generated samples. The benchmark had set the memory length to 100, the sequence length to 1, the category amount to 10, and the sample amount to 40000 throughout the thesis. Each single input vector sequence is created by concatenating three subsequences. The first sequence is the sequence to memorize of length 1. It contains category indices sampled uniformly from 0 to 9. The second sequence is then just a sequence of the filler symbol 10 repeated 100 times. The third sequence is just the index of the category in the sequence to memorize what the model should recall, which is also sampled uniformly from all available indices in the sequence. This sequence is obviously of length 1 and always filled with 0 in the previously described setup. In total, this makes up for a total sequence length of 102 and a vector size of 1 per time step. The expected output category is encoded sparsely as in the Activity Benchmark from Section 3.2 and contains a category index from 0 to 9 that matches the category at the index the model got the last time step in the sequence to memorize. The model's output vector size is 10, and each output logit represents a single category. As this is a classification problem, a categorical cross-entropy loss was used between the model's output logits passed through a softmax function and the one-hot encoding extension of the sparsely encoded expected category index. To better visualize how good a model can recall the category, a categorical accuracy metric was added to this benchmark. It must be pointed out that a model is only capable of recalling the category seen in the first input vector if the gradient does not vanish or explode, as the error signal is only provided after the last time step. A model that cannot capture the long-term dependencies in this benchmark will be forced to learn the memory-less strategy, which entails that all output logits have the same value, i.e., all categories are equally likely. This will lead to a categorical crossentropy loss of $-\ln \frac{1}{10} \approx 2.303$ and a sparse categorical accuracy of roughly 0.1. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/memory_benchmark.py`.

3.6 MNIST Benchmark

This benchmark evaluates how well a model can capture long-term dependencies. For correct classification, the model needs to incorporate input vectors from the distant past, further explained below. The idea to incorporate this benchmark was taken from [LH20], which also features an event-based sequential MNIST classification problem. Input sequences for this benchmark were constructed from the MNIST dataset [LCB10] of the Keras framework [C+15]. The MNIST dataset contains images of hand-drawn digits of 28 by 28 pixels where a single integer encodes each pixel from 0 to 255. All images are in grey-scale, and a higher integer represents a darker pixel. Some examples of these images are given in the following figure:

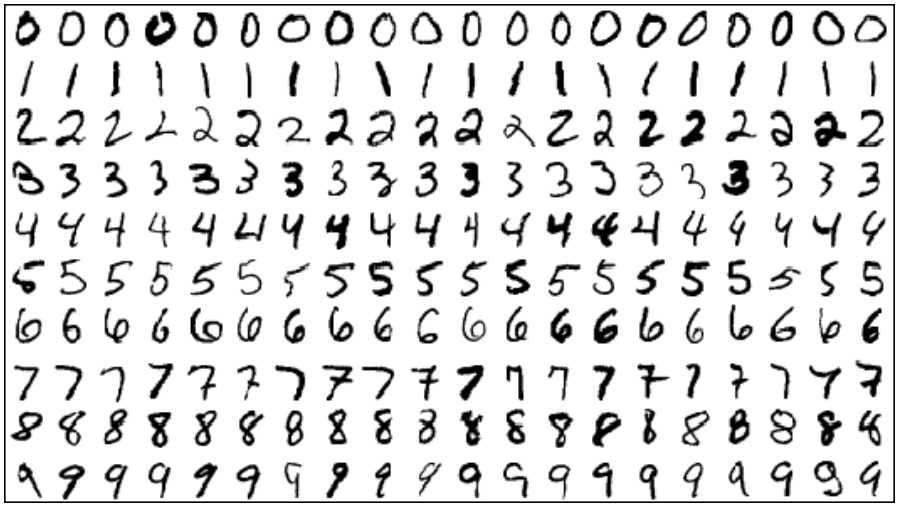

Figure 3.2: images from the MNIST dataset [LCB10]

The images were vectorized to a vector containing 784 entries and then split up to a sequence of vector chunks of size 8, which results in an input sequence length of 98. The expected output class index is just the digit the current image is representing. Furthermore, the benchmark has a configurable maximum amount of samples, which was set to 40000. As the MNIST dataset had enough image samples, all specified 40000 samples were used. Long-term memory of seen input chunks is necessary to produce an accurate category prediction, as digits like 1, 4, and 9 may be indistinguishable when only considering the most recent seen input chunks. This limitation corresponds to classifying the image only based on a lower fraction of the image visible to the model, where the upper fraction was cut away. A model that yields accurate results must not suffer from the vanishing or exploding gradient problem, as only then the whole picture can be taken into account for classification. The model output vector size was set to 10, as each output logit should represent a single digit. As the expected output digit is encoded sparsely, the same procedure as in the Memory Benchmark from Section 3.5 is applied to compute the categorical cross-entropy loss. The models' performance was also measured using a categorical accuracy metric, which produces a more human-interpretable result than the chosen loss function. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/mnist_benchmark.py`.

3.7 Cell Benchmark

This benchmark evaluates if the newly introduced Memory Cell architecture can repeatedly store a single bit of information, including switching the memory state. Furthermore, it should be checked if the memory state vanishes or successfully persists over a long time

horizon. Memory persistence requires capturing long-term dependencies as the input is provided sparsely to the model as described below. The benchmark has a configurable memory high symbol, memory low symbol, memory length, amount of cell switches, and amount of samples generated at each benchmark invocation. The memory high and low symbols represent the expected output symbol when either memory state is active, but the memory high symbol is also used as an input symbol to activate a specific memory state sparsely. All other inputs are then set to the memory low symbol. The memory high symbol was picked to 1, the memory low symbol was picked to 0, the memory length was picked to 128, the number of cell switches was set to 2, and the sample amount was set to 40000. As the Memory Cell architecture is a bistable memory element, two memory states can be activated sparsely. The input vector at each time step has a size of 2. If both entries are 0, the current memory state should be kept. Otherwise, if a single entry is 1 and the other entry is 0, the corresponding memory state should be activated. The first part of the input sequence is constructed by activating any of the memory states sparsely, and the succeeding 127 vectors are all-zero vectors. This subsequence now has a length of 128. The following sequence is built like the first one, but it activates the cell not activated initially, which corresponds to a cell switch. There are 2 further subsequences of this kind. The final input sequence is then the concatenation of all three subsequences and has a length of 384. In half of the samples, either memory state is activated first in the concatenated sequence. The required model output vector is also given as a sequence of vectors of size 2. Therefore the error signal is provided at each time step. The output sequence can be quickly built from the input sequence by continuing to set its entry to 1 at the corresponding index until a new sparsely input is provided to the model. Therefore, the model may get the input sequence consisting of the following vectors: $[1, 0], [0, 0], [0, 0], ..., [0, 1], [0, 0], [0, 0]$ and is required to produce the following vectors of the expected output sequence: $[1, 0], [1, 0], [1, 0], ..., [0, 1], [0, 1], [0, 1]$. The sparse activation of the Memory Cell should lead to permanent storage of the activation until a new sparse input is provided to the model. As described above, the model output vector size is 2, and a mean squared error loss without a metric was used, as more similar vectors lead to a better prediction. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/cell_benchmark.py`.

CHAPTER 4

Results

4.1 Benchmark Hardware and Experiment Clarifications

The benchmark server was equipped with an AMD Ryzen Threadripper 2970WX 24-core processor and two NVIDIA Titan RTX graphics cards. Software-wise, the system used the Ubuntu 18.04.5 LTS operating system and a Python 3.8.7 [VRD09] interpreter to execute all Python scritps. The used Tensorflow [AAB+15] library had version 2.4.1, the used NVIDIA CUDA library had version 11.0.3, the used NVIDIA cuDNN library [CWV+14] had version 8.0.5.39, the used NVIDIA TensorRT library had version 7.2.2 and the NVIDIA GPU driver had version 460.32.03. All benchmarks were started using the script run_all_benchmarks_and_models.py which invokes all applicable benchmark and model combinations once. The CPU and both GPUs were used as computing devices during training. The script can be found under the URL https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/run_all_benchmarks_and_models.py. This script was executed three times, and the produced log data was processed using the script apply_and_save_statistics.py which extracted the statistics (means and standard deviations) in CSV files out of all measured metrics. Time metrics are always reported in seconds. All values in the measured metrics are reported with a precision of three decimal digits. The test loss of models is reported in brackets after their names in the following sections. This script can be found under the URL https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/apply_and_save_statistics.py. To achieve transparency on how the results were obtained, all logs generated during the three benchmark runs by the benchmark framework described in Section 3.1 are available under https://github.com/Oidlichtnwoada/NeuralNetworkArena/tree/master/benchmark_logs.

4.2 Activity Benchmark

The statistics summary for this benchmark is shown in Table 4.1 and the validation losses during training for all models are visualized in Figure 4.1. This benchmark should test whether a model can model a dynamic physical system. The Activity Benchmark is considered to be solved when the categorical accuracy is higher than 0.9. The Transformer architecture achieved the lowest test loss of 0.178 and highest accuracy of 0.937 by incorporating the great expressivity of multi-head attention and the concept of attention. Therefore, the Transformer architecture learns that more recent measurement data will have a more considerable impact on the final classification than measurement data from the distant past. The Transformer architecture is followed by the GRU (0.209), CT-GRU (0.223), DNC (0.229), ODE-LSTM (0.235) and LSTM (0.245) architecture which all delivered a good test loss. This benchmark reveals that GRU and LSTM architectures are not only good in memory-related tasks but can also be used to model a sampled physical system. Remarkably, the GRU architecture, which simplifies the LSTM architecture, outperformed its mother architecture by trading model complexity for hidden state size. Furthermore, the continuous-time variant of the LSTM, the ODE-LSTM, was better suited to model the physical system than the vanilla LSTM architecture, but it also had a larger parameter count. The DNC architecture performed comparatively to the CT-GRU architecture, and the LSTM architecture is followed by the Memory Augmented Transformer (0.257). The DNC and the Memory Augmented Transformer employ an external memory and separate computation from memory. Therefore, they solve each benchmark task by meta-learning, which is a synonym for learning to learn. Gradient descent learns an algorithm to solve each task in these models instead of learning the function that maps input data to output data directly. The DNC has a far more complex model function than the Memory Augmented Transformer and is far more constrained in its operations. These restrictions result in a better test loss than the Memory Augmented Transformer, but the latter also performed exceptionally well with less trainable parameters. Worth mentioning is that all mentioned models except the DNC and Memory Augmented Transformer trained very quickly. The two exceptions needed at least a full hour to train the required function. Until now, all architectures were able to solve the benchmark, i.e., reached a categorical accuracy of more than 0.9, all succeeding models failed to do so. The next best architecture was the Matrix Exponential Unitary RNN (0.336) in its full-space configuration, which outperformed the regular Unitary RNN in test loss and training duration, though both training durations were quite long. As the regular Unitary RNN in its full-space configuration took too long to train, it was benchmarked in its partial-space configuration, therefore the parameter count difference. It should be visible that the unitary matrix parameterization with the matrix exponential is more efficient than the approach with rotational matrices. The Matrix Exponential Unitary RNN architecture is followed by the Recurrent Network Augmented Transformer (0.373) and the Recurrent Network Attention Transformer (0.410). The hypothesis of adding more expressivity to the Transformer architecture by accumulating the weighted value vectors with an LSTM in the Recurrent Network Augmented Transformer is not

valid in this case. The same statement holds for the newly introduced recurrent network attention using Unitary RNNs used in the Recurrent Network Attention Transformer. As both modifications dramatically increase model complexity and training duration compared to the standard Transformer architecture, they are not a viable option for this kind of real-world application. The next best model was the CT-RNN (0.510), followed by the Unitary RNN architecture (0.522). Both architectures have not performed well on this benchmark and needed a long time to train. The CT-RNN architecture should be capable of modeling physical systems as discussed in Section 1.3. The hypothesis is that the additional classification task on top of the physical system modeling was the high test loss's culprit. Perhaps the Matrix Exponential Unitary RNN benefits from using the imaginary part of its hidden state when projecting it to the model output vector using a dense layer. The two remaining models, the Unitary NCP (0.573) and NCP (1.088) architecture, performed very poorly on this benchmark and took a significant amount of time to train. Due to the NCP architecture's complex model function, both models had a small number of neurons that were very sparsely connected with chemical synapses to train them in a reasonable time. This sparseness is a likely reason for the high test loss of both models. Future papers should work on approximations or simplifications of the LTC network architecture, such that more extensive networks are trainable in a reasonable time.

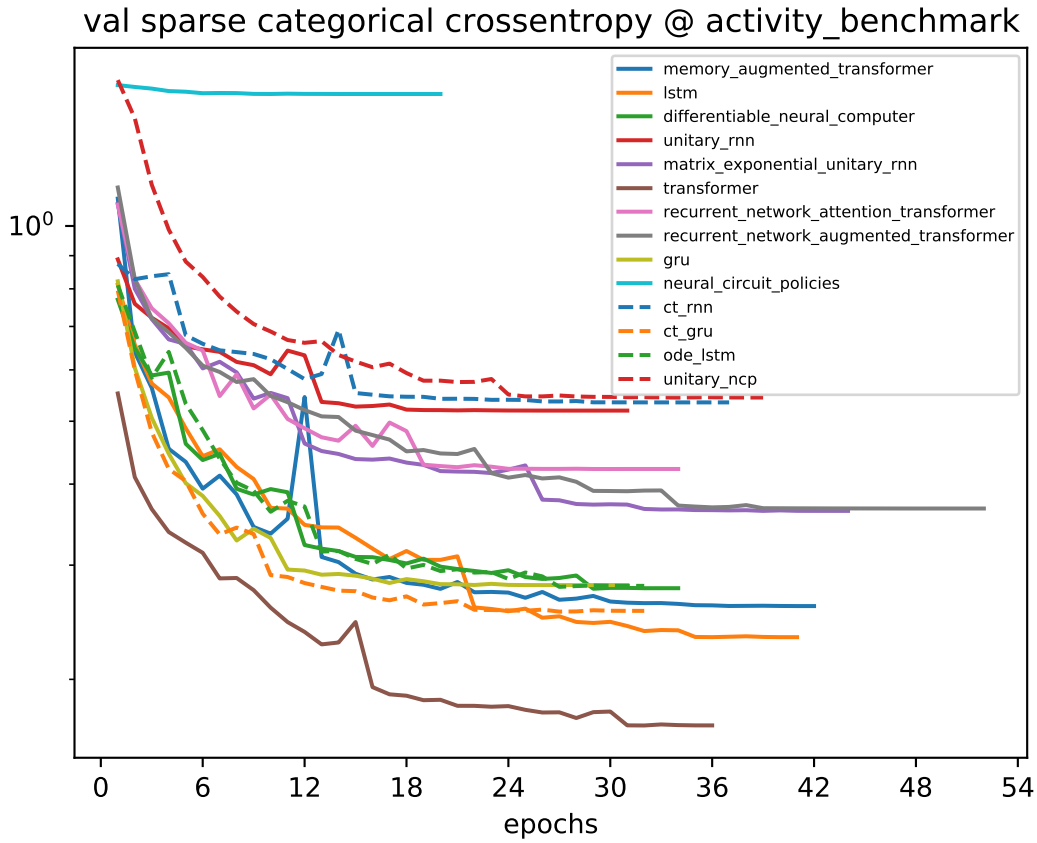

Figure 4.1: validation loss evolution during training for the Activity Benchmark on the second run

model	trainable parameters	training duration total	training duration per epoch	epochs	test sparse categorical crossentropy	test sparse categorical accuracy
transformer	22167	657.964 ± 94.372	19.973 ± 0.446	33.000 ± 5.196	0.178 ± 0.022	0.937 ± 0.010
gru	21927	548.685 ± 97.925	13.752 ± 0.277	40.000 ± 7.810	0.209 ± 0.065	0.929 ± 0.021
ct_gru	21991	1289.666 ± 303.182	31.936 ± 0.483	40.333 ± 9.074	0.223 ± 0.040	0.922 ± 0.014
differentiable_neural_computer	26540	4425.337 ± 929.309	102.129 ± 1.515	43.333 ± 9.018	0.229 ± 0.046	0.915 ± 0.017
ode_lstm	27207	1127.534 ± 168.389	29.398 ± 0.503	38.333 ± 5.508	0.235 ± 0.021	0.914 ± 0.009
lstm	18887	561.106 ± 56.684	13.786 ± 0.216	40.667 ± 3.512	0.245 ± 0.039	0.909 ± 0.013
memory_augmented_transformer	18424	6745.386 ± 1335.496	186.745 ± 6.089	36.000 ± 6.000	0.257 ± 0.024	0.911 ± 0.008
matrix_exponential_unitary_rnn	20231	4206.879 ± 917.213	72.567 ± 2.383	58.000 ± 12.490	0.336 ± 0.047	0.883 ± 0.015
recurrent_network_augmented_transformer	3871	4240.549 ± 859.176	95.246 ± 4.026	44.333 ± 7.095	0.373 ± 0.016	0.868 ± 0.010
recurrent_network_attention_transformer	2491	6245.701 ± 669.365	170.609 ± 4.495	36.667 ± 4.619	0.410 ± 0.019	0.852 ± 0.007
ct_rnn	18439	5310.323 ± 909.814	123.450 ± 0.472	43.000 ± 7.211	0.510 ± 0.074	0.810 ± 0.029
unitary_rnn	4983	6280.606 ± 1197.117	176.368 ± 2.639	35.667 ± 7.234	0.522 ± 0.030	0.808 ± 0.014
unitary_ncp	3579	4386.633 ± 973.768	104.291 ± 1.551	42.000 ± 8.888	0.573 ± 0.046	0.801 ± 0.032
neural_circuit_policies	2857	6353.197 ± 3302.551	126.264 ± 0.247	50.333 ± 26.652	1.088 ± 0.434	0.603 ± 0.197

Table 4.1: statistics of the test loss and other metrics for the Activity Benchmark ($\mu \pm \sigma, N = 3$)

4.3 Add Benchmark

The statistics summary for this benchmark is shown in Table 4.2 and the validation losses during training for all models are visualized in Figure 4.2. This benchmark should test whether a model can capture long-term dependencies in time series. The adding problem is considered to be solved when the mean test loss is under 0.04. Always the same models as reported in Section 4.2 take a significant amount of time to train. Therefore, only exceptions to the norm in terms of training duration will be reported in the following. The Transformer architecture achieved the perfect test loss of 0.000 by incorporating the concept of attention as discussed in Section 4.2. This architecture processes all input vectors of the input vector sequence at once and does not need to save each input vector at a single time step in its hidden state in encoded form. Therefore, it can simply focus on the two marked input vectors in the sequence and add them together without repeatedly applying a model function at each time step. The Recurrent Network Augmented Transformer also achieved the perfect test loss of 0.000 by applying the attention mechanism to the input sequence. The added model complexity of the additional RNN in this architecture has not prevented it from learning the correct model function. The next best model was the GRU architecture (0.001), followed by the CT-GRU architecture (0.001). Both of which performed very well on this benchmark. They both outperformed their LSTM counterparts, given by the LSTM and ODE-LSTM architecture. Furthermore, the Recurrent Network Attention Transformer (0.002) also performed very well on this task. It uses the introduced recurrent network attention mechanism and a Unitary RNN, and this combination has beaten the vanilla Unitary RNN architecture in terms of test loss. This architecture is followed by the DNC (0.007), the CT-RNN (0.019), and the Matrix Exponential Unitary RNN architecture (0.022). The DNC learned the desired function using meta-learning and saved the marked values to its external memory. Surprisingly, the CT-RNN architecture was also able to learn the add function, even though the architecture has no gating mechanism like the LSTM or GRU architecture and no bounded loss gradient like the Unitary RNNs. The Matrix Exponential Unitary RNN was also able to learn the add function and training was significantly more stable than for the standard Unitary RNN as measured by the test loss standard deviation of 0.032 compared to 0.076. As in the Activity Benchmark, the Matrix Exponential Unitary RNN has outperformed the Unitary RNN in test loss and training duration. The following models are the LSTM (0.066), the Unitary RNN (0.094), the Unitary NCP (0.106), and the Memory Augmented Transformer architecture (0.122). These models are not considered to have solved the adding problem as their test loss is larger than 0.04. All four test loss standard deviations of these models are relatively high (larger than 0.075) because they have solved the adding problem in some benchmark runs, whereas, in the other benchmark runs, they failed to do so. In the LSTM and Unitary RNN architecture case, ill-suited initialization values may cause different behaviors on different runs, hindering the optimizer from finding suitable parameters for the models. The Matrix Exponential Unitary RNN always initializes its matrix W to the identity matrix, which somehow helps the optimizer tune the model's parameters

in this benchmark task. Ill-suited initialization values are also a possible cause in the Unitary NCP and Memory Augmented Transformer architecture, but for the Unitary NCP architecture, the very sparsely connected neurons discussed in Section 4.2 may also be a problem. The Memory Augmented Transformer architecture may smoothen its loss surface, which helps the optimizer by setting its embedding size equal to the vector size stored in each memory row. Then the memory embedding can be omitted, and its parameters can be used for the multi-head self-attention mechanism. Furthermore, the memory control dense layer may use a residual connection to specify the memory change instead of building a new memory row vector from scratch at each time step when the enable signal is active. The worst two models which were only able to learn the memory-less strategy discussed in Section 3.3 are the NCP (0.166) and the ODE-LSTM model (0.167). As the NCP architecture has no bounded loss gradient, it suffers from the vanishing gradient problem as discussed in [LHA+20, p. 2] and therefore cannot memorize the two marked numbers. The sparsity problem also applies here. The Unitary NCP architecture performed better on this task because it incorporates a Unitary RNN for memory-related tasks. Surprisingly, the ODE-LSTM architecture was not able to capture the long-term dependencies in this benchmark task. Somehow the added CT-RNN in its architecture changed the loss surface such that it is more difficult for the optimizer to find suitable parameters.

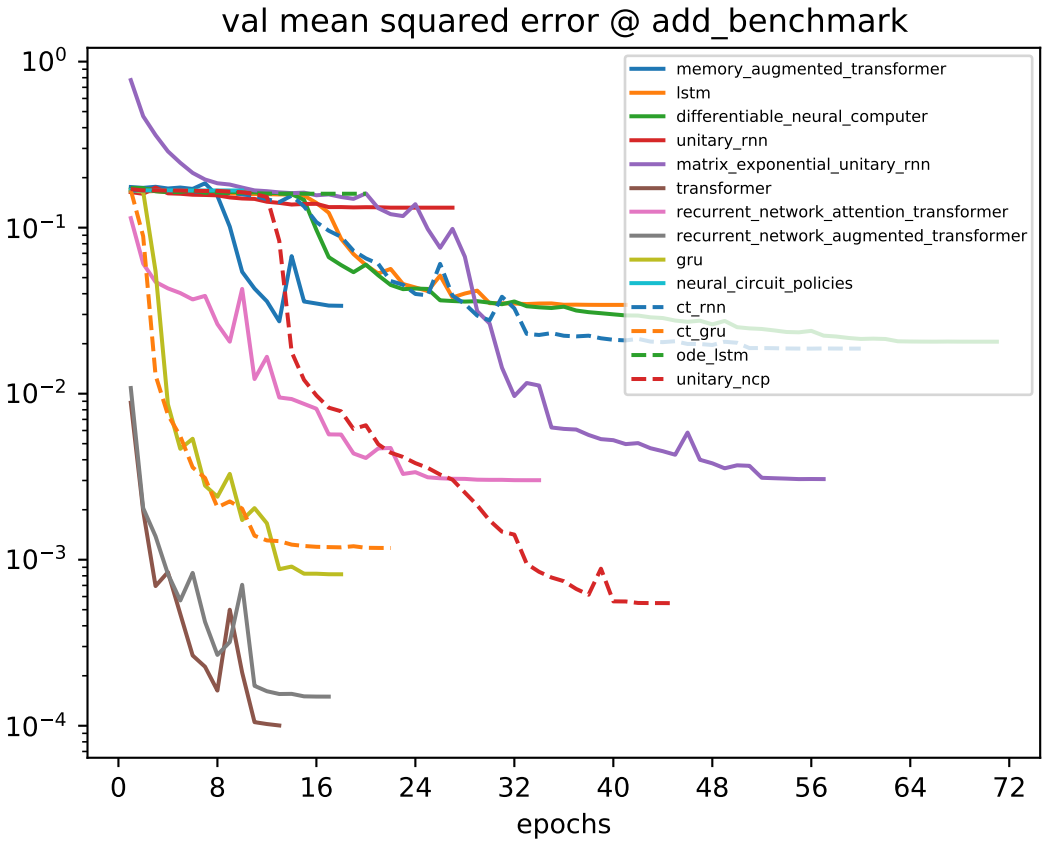

Figure 4.2: validation loss evolution during training for the Add Benchmark on the second run

model	trainable parameters	training duration total	training duration per epoch	epochs	test mean squared error
transformer	21889	421.042 ± 62.125	33.175 ± 3.885	12.667 ± 0.577	0.000 ± 0.000
recurrent_network_augmented_transformer	3729	3369.947 ± 238.022	198.209 ± 6.384	17.000 ± 1.000	0.000 ± 0.000
gru	20241	423.332 ± 52.080	20.821 ± 0.322	20.333 ± 2.517	0.001 ± 0.000
ct_gru	19073	983.703 ± 123.086	48.464 ± 0.901	20.333 ± 2.887	0.001 ± 0.000
recurrent_network_attention_transformer	2349	10433.169 ± 3205.297	370.905 ± 13.507	28.000 ± 7.937	0.002 ± 0.002
differentiable_neural_computer	24774	6138.672 ± 4012.017	152.861 ± 2.424	40.333 ± 26.858	0.007 ± 0.011
ct_rnn	17025	9904.616 ± 1578.008	193.094 ± 1.600	51.333 ± 8.505	0.019 ± 0.003
matrix_exponential_unitary_rnn	17409	5956.370 ± 1220.646	115.400 ± 11.578	51.333 ± 6.028	0.022 ± 0.032
lstm	17217	589.247 ± 181.783	19.420 ± 0.162	30.333 ± 9.292	0.066 ± 0.085
unitary_rnn	2929	9694.727 ± 2483.450	278.442 ± 11.350	35.000 ± 9.849	0.094 ± 0.076
unitary_ncp	1885	3283.152 ± 1513.217	102.520 ± 0.625	32.000 ± 14.731	0.106 ± 0.092
memory_augmented_transformer	18066	3802.347 ± 1303.816	283.514 ± 8.116	13.333 ± 4.163	0.122 ± 0.083
neural_circuit_policies	1349	1007.356 ± 252.374	125.919 ± 0.414	8.000 ± 2.000	0.166 ± 0.001
ode_lstm	25537	748.046 ± 108.977	44.046 ± 0.415	17.000 ± 2.646	0.167 ± 0.002

Table 4.2: statistics of the test loss and other metrics for the Add Benchmark ($\mu \pm \sigma, N = 3$)

4.4 Walker Benchmark

The statistics summary for this benchmark is shown in Table 4.3 and the validation losses during training for all models are visualized in Figure 4.3. This benchmark should test whether a model can model a dynamic physical system. The Walker Benchmark is considered to be solved when the test loss is smaller than 1.5. The ODE-LSTM architecture achieved the lowest test loss of 1.159 by incorporating the capability to model physical systems of the CT-RNN and the capability to capture long-term dependencies of the LSTM architecture. Even though the most recent states accurately determine the next state of the physics simulation, additional memory helps in this task as the CT-RNN is the only second best architecture with a test loss of 1.205, but with a smaller parameter count. This architecture seems to benefit from its model structure as hypothesized in Section 1.3, as the benchmark task is just about finding the input-output relation of a physical system without an additional classification task. The next best architectures were the Memory Augmented Transformer (1.213) and the DNC (1.330). Both of these architectures have an external memory and employ meta-learning. Surprisingly, the Memory Augmented Transformer, with its more flexible model function when compared to the DNC, achieved a lower test loss in this benchmark. The DNC was followed by the GRU (1.339), the LSTM (1.363), and the CT-GRU (1.526) architecture, which all performed exceptionally well on this benchmark task. Until the LSTM all architectures could solve the benchmark, i.e., reached a test loss of less than 1.5, all succeeding models failed to do so. It seems that employing continuous-time models like the ODE-LSTM helps to achieve better results than employing discrete-time models even when regularly sampled input data is used. This ability to generalize to arbitrary time inputs of continuous-time models is discussed in Section 2.6 and computing the state change may be easier than computing a new state at each time step. The next best architecture was the Transformer which achieved a test loss of 1.599. This benchmark task seemed to be challenging for the Transformer architecture, as maybe the positional encoding cannot be appropriately incorporated when attention to several individual positions is required. That is probably why recurrent neural network architectures perform better on this task, as they naturally incorporate positional information in their model functions. This deficiency of the Transformer was not such a significant issue in the Activity Benchmark from Section 4.2, as the activity labels changed very infrequently. The Transformer was followed by the Unitary RNN (1.622), which outperformed the Matrix Exponential Unitary RNN in this benchmark. Probably, a partial-space unitary matrix W and the different parametrization helps with this benchmark. The following two models in the ranking were the Recurrent Network Augmented Transformer (2.207) and the Recurrent Network Attention Transformer (2.490). Both architectures should augment the Transformer architecture in different ways by adding RNNs to it, which should help with exact positional information. As the test loss of both architectures is relatively high, this idea has not worked. The Matrix Exponential Unitary RNN was the next model with a test loss of 3.180 and an outstanding high test loss standard deviation of 1.406. In one benchmark run, the model even achieved a comparative test loss to the Unitary RNN. Maybe an improved

initialization strategy helps to stabilize training for this model. The worst two models were the Unitary NCP (3.438) and the NCP (4.850) architecture, which both suffered from their sparsely connected few neurons.

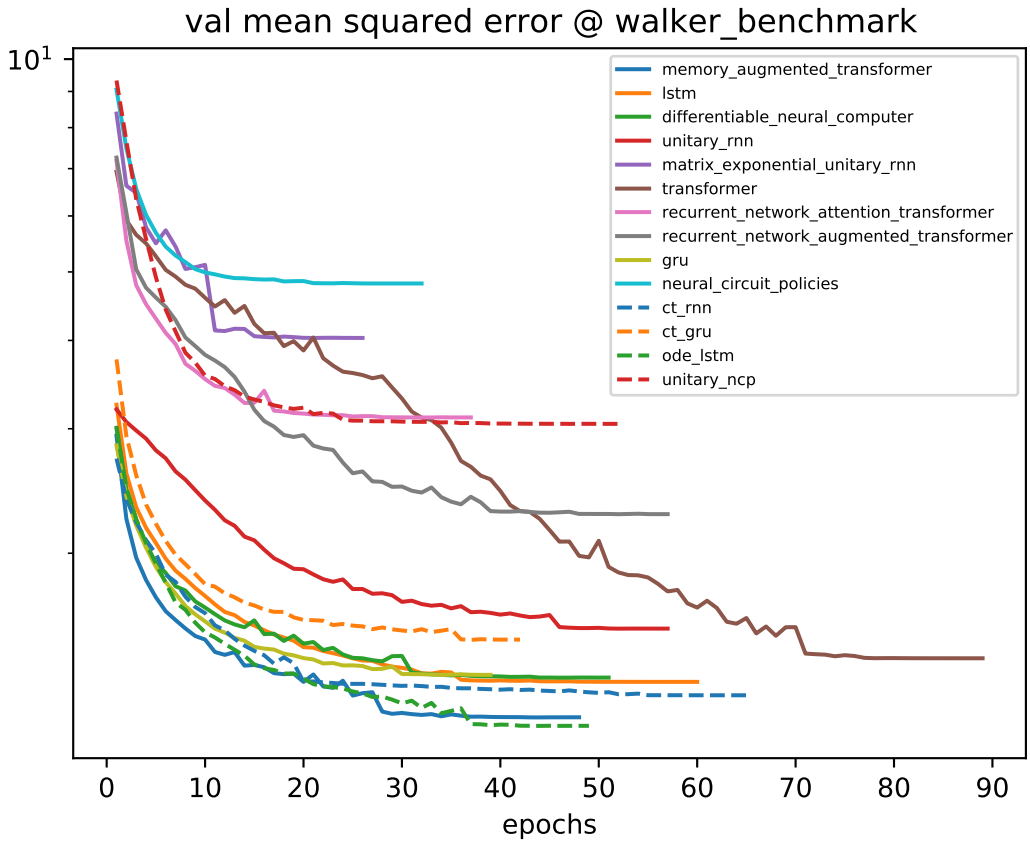

Figure 4.3: validation loss evolution during training for the Walker Benchmark on the second run

model	trainable parameters	training duration total	training duration per epoch	epochs	test mean squared error
ode_lstm	30417	1479.639 ± 82.601	29.407 ± 0.292	50.333 ± 3.215	1.159 ± 0.067
ct_rnn	21009	7284.599 ± 661.406	122.746 ± 0.619	59.333 ± 5.132	1.205 ± 0.040
memory_augmented_transformer	19074	8482.406 ± 397.415	187.573 ± 15.380	45.333 ± 2.517	1.213 ± 0.060
differentiable_neural_computer	29910	5224.531 ± 263.850	101.772 ± 0.303	51.333 ± 2.517	1.330 ± 0.027
gru	25137	574.551 ± 24.867	14.133 ± 0.258	40.667 ± 2.082	1.339 ± 0.025
lstm	22097	775.982 ± 51.555	14.030 ± 0.120	55.333 ± 4.163	1.363 ± 0.040
ct_gru	27761	1298.792 ± 146.468	32.461 ± 0.419	40.000 ± 4.359	1.526 ± 0.031
transformer	22657	1548.471 ± 218.343	19.737 ± 0.384	78.333 ± 9.452	1.599 ± 0.156
unitary_rnn	8833	9231.787 ± 589.190	173.114 ± 1.619	53.333 ± 3.512	1.622 ± 0.030
recurrent_network_augmented_transformer	4121	5438.420 ± 330.703	92.170 ± 0.355	59.000 ± 3.464	2.207 ± 0.059
recurrent_network_attention_transformer	2741	6955.163 ± 1436.109	168.194 ± 0.721	41.333 ± 8.386	2.490 ± 0.469
matrix_exponential_unitary_rnn	25361	3896.433 ± 2610.319	70.771 ± 1.064	55.000 ± 36.510	3.180 ± 1.406
unitary_ncp	7149	8292.842 ± 1105.148	171.287 ± 10.491	48.333 ± 4.726	3.438 ± 0.296
neural_circuit_policies	6767	8099.388 ± 2507.014	174.074 ± 2.597	46.667 ± 15.011	4.850 ± 0.128

Table 4.3: statistics of the test loss and other metrics for the Walker Benchmark ($\mu \pm \sigma, N = 3$)

4.5 Memory Benchmark

The statistics summary for this benchmark is shown in Table 4.4 and the validation losses during training for all models are visualized in Figure 4.4. This benchmark should test whether a model can capture long-term dependencies in time series. The Memory Benchmark is considered to be solved when the categorical accuracy is higher than 0.9. Four architectures achieved the perfect categorical accuracy of 1.000: the Unitary RNN (0.000), the Recurrent Network Attention Transformer (0.008), the Matrix Exponential Unitary RNN (0.062) and the Unitary NCP architecture (0.205). The Unitary RNN has outperformed the Matrix Exponential Unitary RNN, which has again a problem with a high test loss standard deviation of 0.042 when compared to 0.000 of the Unitary RNN. This problem was also discussed in Section 4.4, and a partial-space unitary matrix W might be easier to handle by the optimizer. The Unitary RNN trained very quickly, and with its bounded loss gradient, it also helped the Recurrent Network Attention Transformer and the Unitary NCP architecture produce excellent results. The last architecture that solved the benchmark was the Recurrent Network Augmented Transformer (0.205) with a categorical accuracy of 0.966. It is impressive that both Transformer derivative models outperformed their mother architecture by better incorporating positional information and an RNN with bounded loss gradient. The Recurrent Network Augmented Transformer even outperformed the LSTM architecture, though its core uses an LSTM to accumulate the attention mechanism's weighted value vectors. The Unitary NCP does an excellent job of delegating this memory-related task to the Unitary RNN. The next best model was the CT-GRU (0.321), followed by the Transformer (0.362), the GRU (0.768), and the DNC architecture (1.534). All of these models were able to solve the Memory Benchmark in some benchmark runs and failed in other runs, leading to a high test loss standard deviation of all these models. Maybe an improved initialization strategy for these models helps with this problem. Surprisingly, the Transformer architecture could not solve the Memory Benchmark in every single run, though it just needs to attend to the first input vector that contains the required category. This lousy result means that the Transformer's positional encoding might not work as reliable as expected in [VSP+17, p. 5-6]. The worst models which only learned the memory-less strategy discussed in Section 3.5 were the Memory Augmented Transformer, the LSTM, the NCP, the CT-RNN, and the ODE-LSTM architecture, all of which have achieved a test loss of 2.303 and a categorical accuracy of roughly 0.1. The Memory Augmented Transformer may improve its performance by incorporating the changes discussed in Section 4.3. Outstandingly, both LSTM architectures failed to solve the benchmark, even though they employ gating on their cell state. Not very surprisingly, the NCP and CT-RNN architecture failed to solve the task, as none has a bounded loss gradient, and therefore both suffer from the vanishing gradient problem. If the gradients were exploding, the training loop would have terminated itself as specified by the TerminateOnNan callback described in Section 3.1.

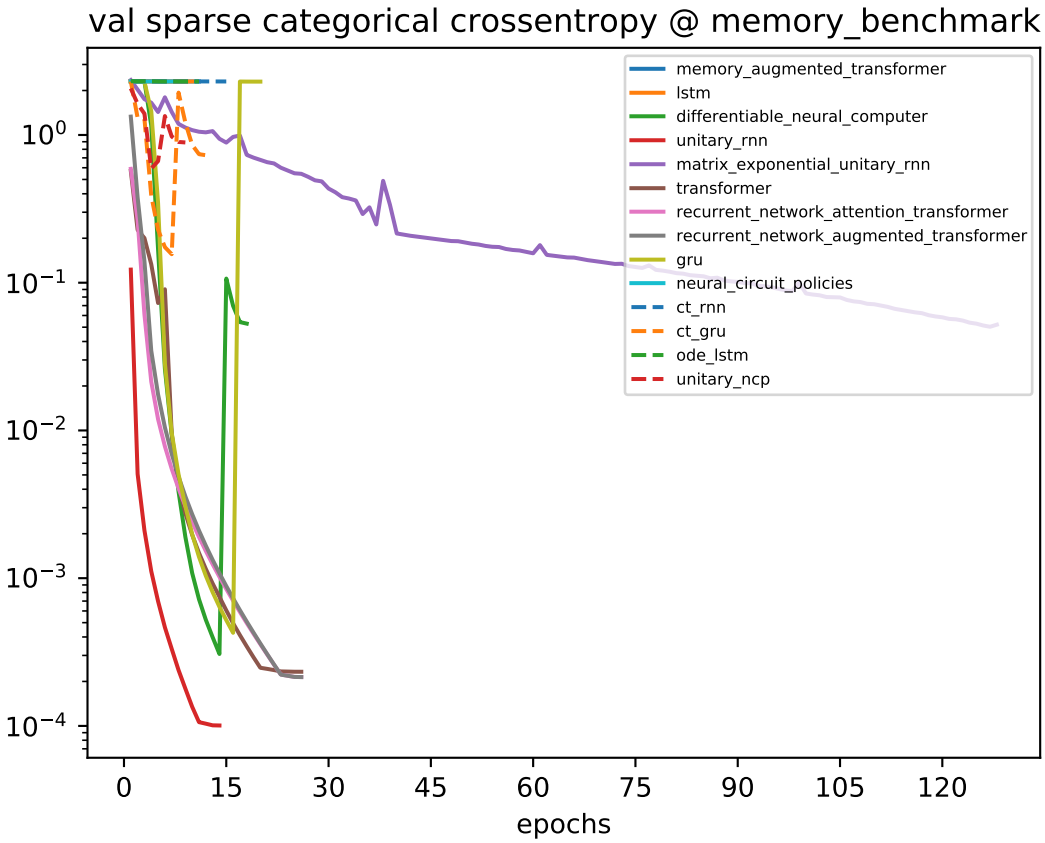

Figure 4.4: validation loss evolution during training for the Memory Benchmark on the second run

model	trainable parameters	training duration total	training duration per epoch	epochs	test sparse categorical crossentropy	test sparse categorical accuracy
unitary_rnn	3834	3576.441 ± 293.414	282.500 ± 2.493	12.667 ± 1.155	0.000 ± 0.000	1.000 ± 0.000
recurrent_network_attention_transformer	2494	5265.193 ± 2737.636	333.093 ± 11.050	16.000 ± 8.888	0.008 ± 0.009	1.000 ± 0.000
matrix_exponential_unitary_rnn	19466	11178.496 ± 4789.471	110.305 ± 3.039	102.000 ± 45.033	0.062 ± 0.042	1.000 ± 0.000
unitary_ncp	3990	4478.233 ± 3554.849	178.365 ± 1.691	25.000 ± 19.698	0.205 ± 0.338	1.000 ± 0.000
recurrent_network_augmented_transformer	3874	4819.569 ± 1163.124	201.743 ± 7.181	24.000 ± 6.245	0.205 ± 0.355	0.966 ± 0.058
ct_gru	18826	1183.943 ± 850.848	49.173 ± 1.886	23.667 ± 16.073	0.321 ± 0.422	0.864 ± 0.157
transformer	22170	641.340 ± 214.464	31.795 ± 1.909	20.000 ± 6.000	0.362 ± 0.627	0.832 ± 0.290
gru	20730	419.598 ± 197.926	20.865 ± 0.572	20.000 ± 9.000	0.768 ± 1.329	0.699 ± 0.521
differentiable_neural_computer	25247	1971.082 ± 744.754	155.491 ± 1.628	12.667 ± 4.726	1.534 ± 1.328	0.399 ± 0.521
memory_augmented_transformer	18331	3457.429 ± 653.695	296.012 ± 3.312	11.667 ± 2.082	2.303 ± 0.000	0.103 ± 0.003
lstm	17546	231.764 ± 52.835	19.923 ± 0.361	11.667 ± 2.887	2.303 ± 0.000	0.099 ± 0.006
neural_circuit_policies	2908	1750.867 ± 515.520	227.517 ± 4.961	7.667 ± 2.082	2.303 ± 0.000	0.101 ± 0.002
ct_rnn	18058	2709.544 ± 224.245	198.286 ± 2.455	13.667 ± 1.155	2.303 ± 0.000	0.100 ± 0.003
ode_lstm	25866	563.901 ± 89.358	44.552 ± 1.623	12.667 ± 2.082	2.303 ± 0.001	0.101 ± 0.004

Table 4.4: statistics of the test loss and other metrics for the Memory Benchmark ($\mu \pm \sigma, N = 3$)

4.6 MNIST Benchmark

The statistics summary for this benchmark is shown in Table 4.5 and the validation losses during training for all models are visualized in Figure 4.5. This benchmark should test whether a model can capture long-term dependencies in time series. The MNIST classification problem is considered to be solved when the categorical accuracy is higher than 0.9. The DNC architecture achieved the lowest test loss of 0.201 by incorporating meta-learning and its external memory, and it also achieved the highest categorical accuracy of 0.940. As mentioned in Section 4.2, this architecture learns an algorithm to solve the problem by gradient descent, and this approach seems to work quite well on this task. The other models able to solve the benchmark were the GRU (0.221), the LSTM (0.285), and the Unitary RNN architecture (0.365). The Recurrent Network Attention Transformer had a slightly lower test loss than the Unitary RNN with 0.345, but its categorical accuracy was slightly lower than 0.9 with 0.893. As the Unitary RNN has a bounded loss gradient and performs exceptionally well on this benchmark, the Transformer derivative model that uses it performed well, too. The GRU and LSTM architecture's gating mechanism seems to work as expected in this benchmark, and as in the memory-related Add Benchmark, the ODE-LSTM performed worse than the vanilla LSTM architecture. It seems that the additional postprocessing of the hidden state vector with a CT-RNN in the ODE-LSTM has a negative influence on the gating mechanism in memory-related tasks. The next best model was the ODE-LSTM (0.372) followed by the CT-GRU (0.588), the Transformer (0.654), the Matrix Exponential Unitary RNN (0.712), and the Recurrent Network Augmented Transformer architecture (0.754). All of these models could not solve the benchmark task as defined, but they all delivered decent results. Interestingly, the CT-GRU with its multi-dimensional exponentially decaying state performs worse than the vanilla GRU model, even though it was constructed to generalize the GRU architecture. The Transformer architecture should have also performed better on this benchmark, as it could merely attend to image chunks that are different between the ten digits. As this was not the case, the hypothesis that the positional encoding might not work as expected formulated in Section 4.5 is further strengthened. The Recurrent Network Augmented Transformer, designed to be a generalization of the Transformer architecture, performed worse than its mother architecture and did not show any advantages in this benchmark task. The Unitary RNN outperformed the Matrix Exponential Unitary RNN in this benchmark, which only uses a partial-space unitary matrix W, and this seems to ease optimization. The worst models in this benchmark were the CT-RNN (1.164), the Memory Augmented Transformer (1.313), the NCP (1.624), and the Unitary NCP architecture (1.876). As already discussed in Section 4.5, the CT-RNN and NCP architecture both suffer from the vanishing gradient problem, which leads to wrong classifications. Furthermore, the sparsely connected few neurons in the NCP and Unitary NCP architecture may be a reason for inferior performance compared to other models. The Memory Augmented Transformer could not apply the concept of meta-learning as effectively as the DNC in this benchmark, but it may improve its performance by incorporating the changes

discussed in Section 4.3.

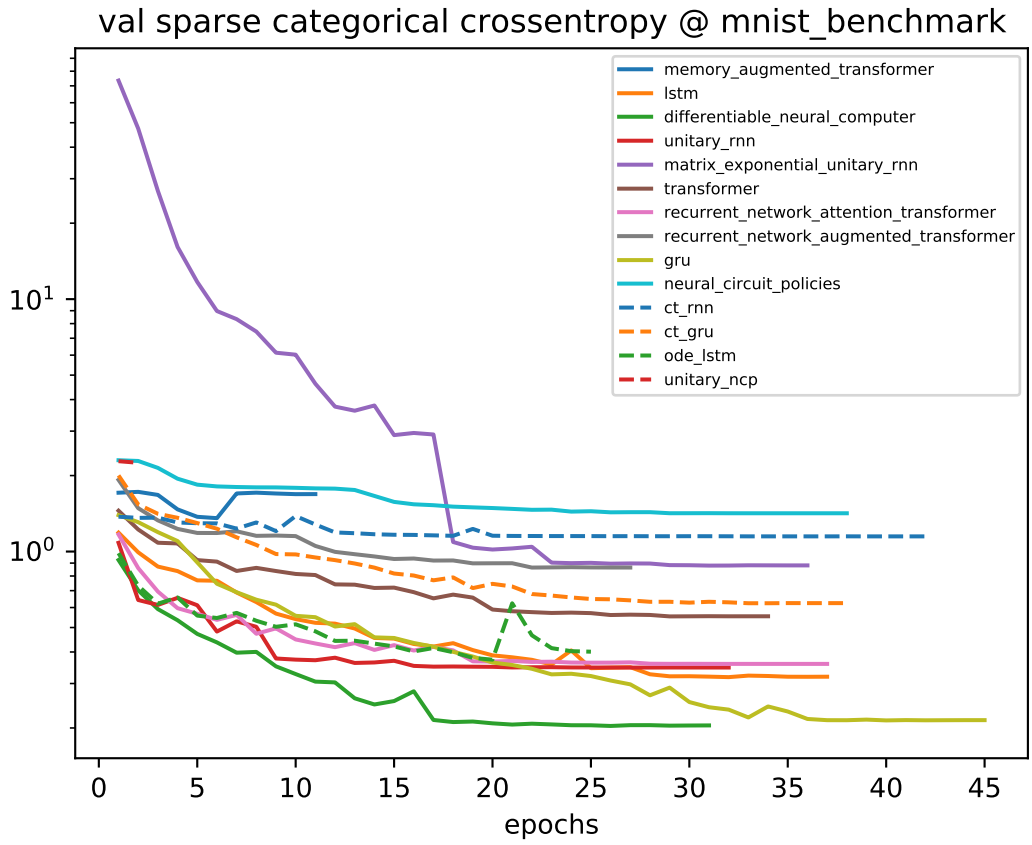

Figure 4.5: validation loss evolution during training for the MNIST Benchmark on the second run

model	trainable parameters	training duration total	training duration per epoch	epochs	test sparse categorical crossentropy	test sparse categorical accuracy
differentiable_neural_computer	27039	4695.074 ± 571.804	148.336 ± 0.894	31.667 ± 4.041	0.201 ± 0.025	0.940 ± 0.006
gru	22410	882.416 ± 132.625	20.510 ± 0.287	43.000 ± 6.245	0.221 ± 0.015	0.935 ± 0.006
lstm	19338	695.434 ± 63.874	19.671 ± 0.268	35.333 ± 2.887	0.285 ± 0.018	0.903 ± 0.008
recurrent_network_attention_transformer	2550	11926.374 ± 3235.949	335.129 ± 4.049	35.667 ± 10.066	0.354 ± 0.070	0.893 ± 0.021
unitary_rnn	5626	6211.725 ± 2629.199	265.320 ± 3.883	23.333 ± 9.609	0.365 ± 0.050	0.903 ± 0.020
ode_lstm	27658	1063.088 ± 74.767	43.666 ± 1.473	24.333 ± 1.155	0.372 ± 0.069	0.872 ± 0.034
ct_gru	22634	2287.787 ± 474.193	47.933 ± 3.654	47.667 ± 9.074	0.588 ± 0.038	0.785 ± 0.012
transformer	22282	1097.666 ± 75.003	29.961 ± 0.477	36.667 ± 3.055	0.654 ± 0.082	0.769 ± 0.037
matrix_exponential_unitary_rnn	21258	5700.014 ± 1943.402	108.716 ± 7.868	53.000 ± 20.664	0.712 ± 0.215	0.779 ± 0.062
recurrent_network_augmented_transformer	3930	9250.855 ± 4071.886	185.565 ± 2.351	50.000 ± 22.517	0.754 ± 0.130	0.738 ± 0.048
ct_rnn	18954	5741.032 ± 2668.638	190.844 ± 5.058	30.333 ± 14.572	1.164 ± 0.026	0.570 ± 0.013
memory_augmented_transformer	18555	3896.794 ± 2556.562	272.368 ± 1.556	14.333 ± 9.452	1.313 ± 0.112	0.488 ± 0.049
neural_circuit_policies	3650	7386.977 ± 723.096	215.100 ± 1.357	34.333 ± 3.215	1.624 ± 0.168	0.380 ± 0.062
unitary_ncp	4438	1590.662 ± 966.124	169.083 ± 9.195	9.333 ± 5.508	1.876 ± 0.319	0.323 ± 0.133

Table 4.5: statistics of the test loss and other metrics for the MNIST Benchmark ($\mu \pm \sigma, N = 3$)

4.7 Cell Benchmark

The statistics summary for this benchmark is shown in Table 4.6 and the validation loss during training for the Memory Cell is visualized in Figure 4.6. This benchmark should test whether the Memory Cell architecture can capture long-term dependencies in time series. This ability was present as the Memory Cell achieved a perfect test loss of 0.000 in each of the three benchmark runs with a test loss standard deviation of 0.000. The total amount of epochs to train was relatively low with 6 as the Memory Cell's initialization values were picked close to a local minimum. Otherwise, the loss gradient kept diverging. This test loss result validates that the architecture described in Section 2.16 is indeed able to capture long-term dependencies in time series similar to the input time series of the Cell Benchmark introduced in Section 3.7. A downside of the architecture is the training duration of nearly two minutes per epoch for an architecture that only has 9 trainable parameters. The Cell Benchmark featured an input sequence length of 384 and 40000 samples in total. Therefore, there is lots of future work to do to speed up this model's training by using simplifications or approximations in the model function. Another open question is how to intelligently couple multiple Memory Cells together. A further question is if these Memory Cells should be coupled tightly or instead loosely like the memory bits in our personal computers. Furthermore, a controller should be introduced like the one in the DNC architecture that provides suitable input currents to the Memory Cells.

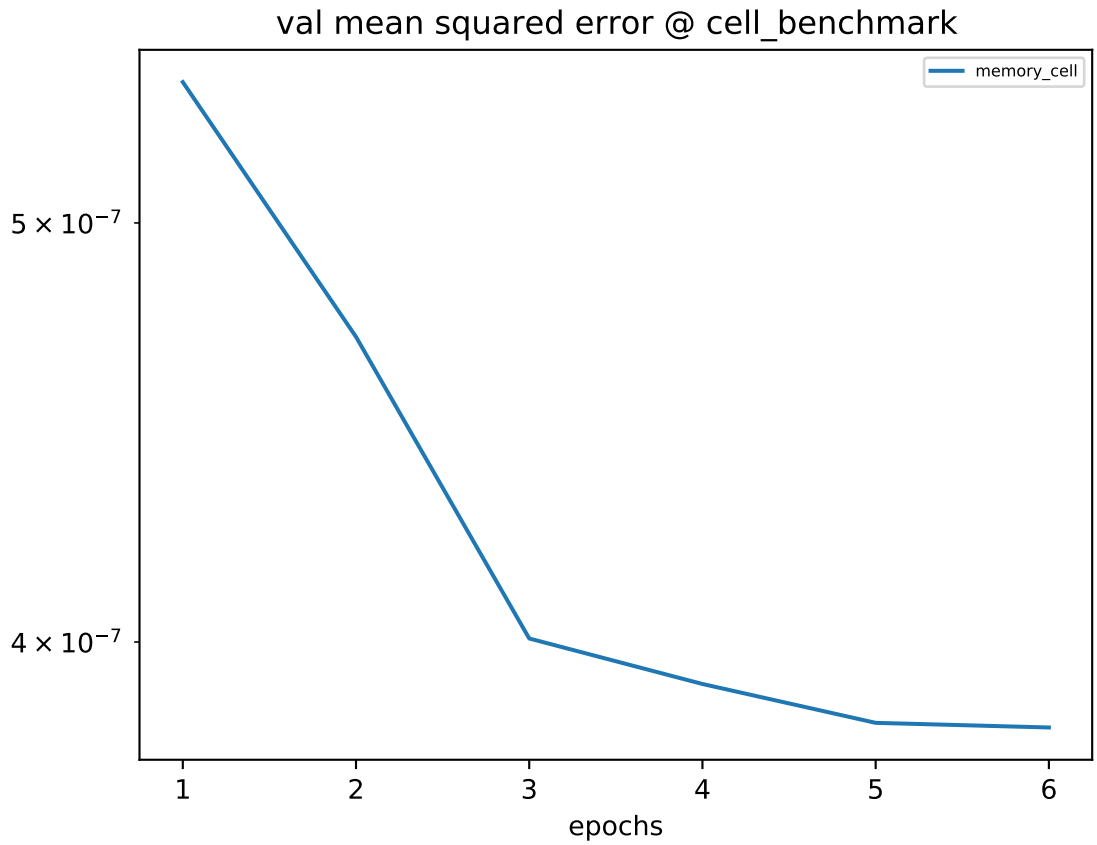

Figure 4.6: validation loss evolution during training for the Cell Benchmark on the second run

model	trainable parameters	training duration total	training duration per epoch	epochs	test mean squared error
memory_cell	9	631.731 ± 7.779	105.288 ± 1.297	6.000 ± 0.000	0.000 ± 0.000

Table 4.6: statistics of the test loss and other metrics for the Cell Benchmark ($\mu \pm \sigma, N = 3$)

CHAPTER 5

Summary and Future Work

The Transformer architecture has a superior expressivity but shows deficiencies in tasks where exact positional information is required. Possible future research should be directed on how the Transformer architecture can more effectively use the positional data in input time series. The GRU and LSTM architecture showed that the gating mechanism works as expected in most cases, and the GRU architecture is a meaningful simplification of the LSTM architecture. The GRU architecture outperformed the LSTM architecture by tradining in model complexity for hidden state size in all benchmarks. In most benchmarks, the CT-GRU performed comparatively to the vanilla GRU and did not show increased performance due to the more general model function. The ODE-LSTM was especially good on tasks invoking dynamic physical systems but showed deficiencies in memory-related tasks. The Unitary RNN and Matrix Exponential Unitary RNN performed very well on memory-related tasks but had their problems modeling physical systems. Some problems might be mitigated by researching how to initialize these models better. Furthermore, the influence of both models' used capacities on the test loss would be quite interesting. The DNC architecture employed the meta-learning mechanism more effectively than the Memory Augmented Transformer, most likely because of its more constrained memory operations. The Memory Augmented Transformer also showed some promising results, but training is not stable enough. It would be interesting to implement the possible improvements mentioned in Section 4.3 in future work and evaluate their effect on test loss and training stability. The Recurrent Network Augmented Transformer and the Recurrent Network Attention Transformer, both Transformer derivatives, performed inferior in all tasks where the Transformer architecture delivered good results. In tasks where the Transformer architecture struggled, the added RNNs in these architectures helped both to achieve superior results compared to the Transformer in some cases. The very sparsely connected few neurons in the NCP and Unitary NCP architecture held both architectures back. A simplification or approximation of the LTC network model function without losing its expressivity should be desired. This improvement would also

help the Memory Cell architecture. Furthermore, there were some other research topics regarding the Memory Cell discussed in Section 4.7. In some benchmarks, the Unitary RNN inside the Unitary NCP was responsible for decent test loss results. The CT-RNN architecture only delivered good results on benchmarks that asked for a physical system's pure input-output relation with the following exception. Surprisingly, the Add Benchmark was also solved by the CT-RNN, even though the loss gradient is not bounded in this model. This unbounded gradient leads to weak results in other long-term dependency benchmarks where the vanishing gradient problem appeared. Moreover, better tuning on each benchmarked model's hyperparameters can be applied in future work to use the fixed number of parameters in the most efficient way. Also, the effect of different batch sizes, learning rates, and optimizers on the test loss may be an exciting subject for future work based on this thesis.

CHAPTER 6

Appendix

6.1 Individual Training Plots

6.1.1 Activity Benchmark

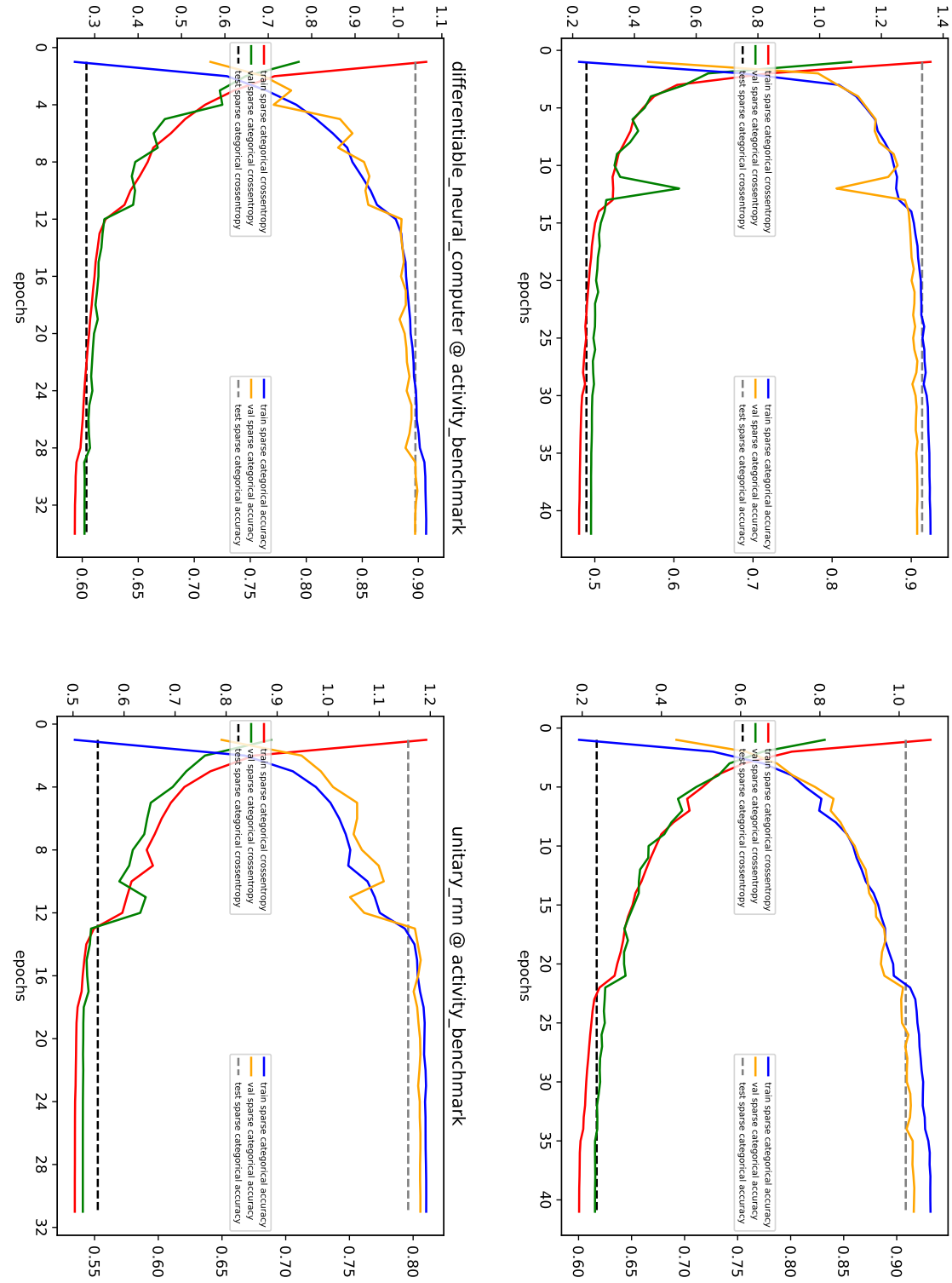

Figure 6.1: individual training plots for the Activity Benchmark on the second run - part 1

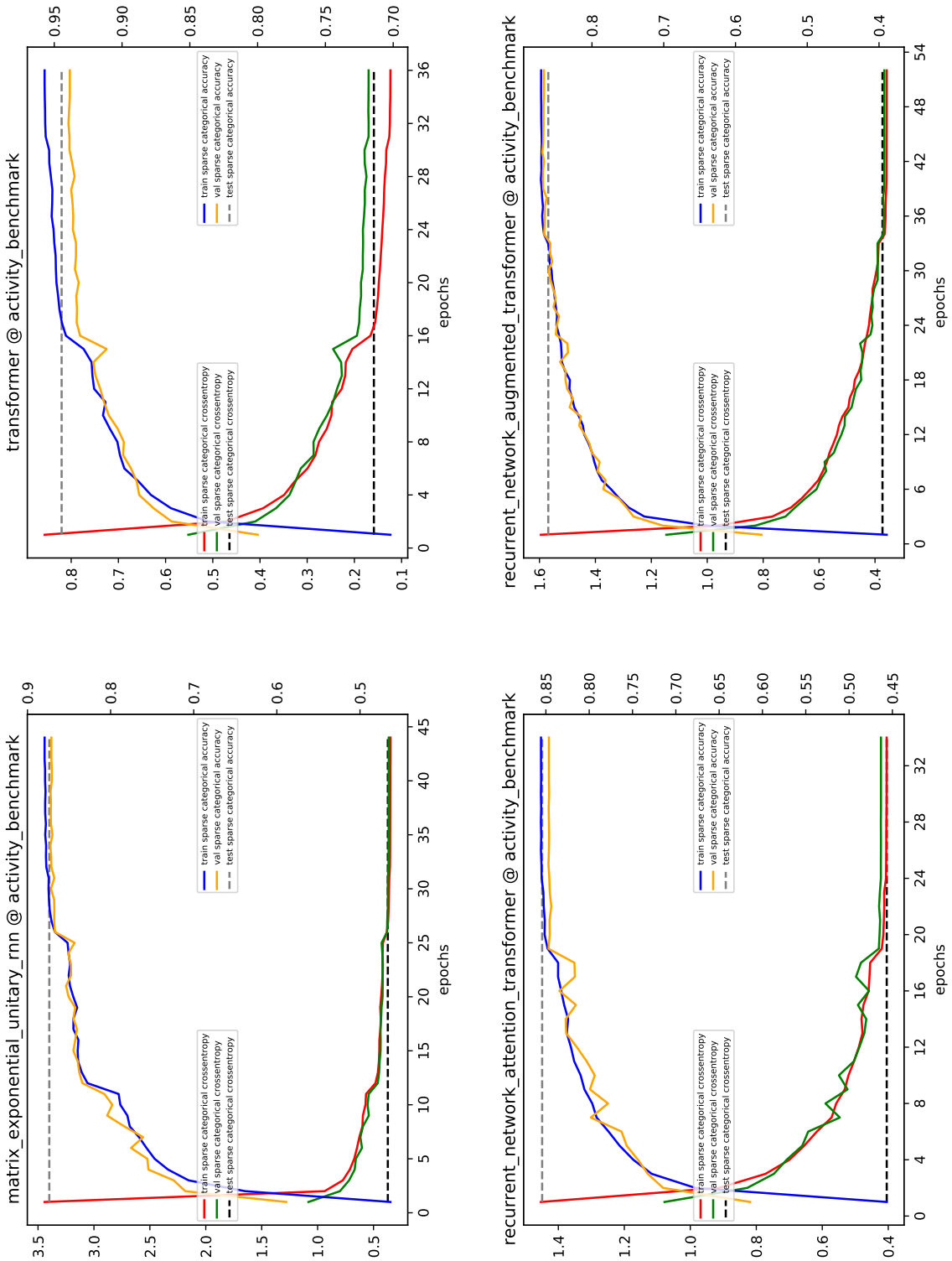

Figure 6.2: individual training plots for the Activity Benchmark on the second run - part 2

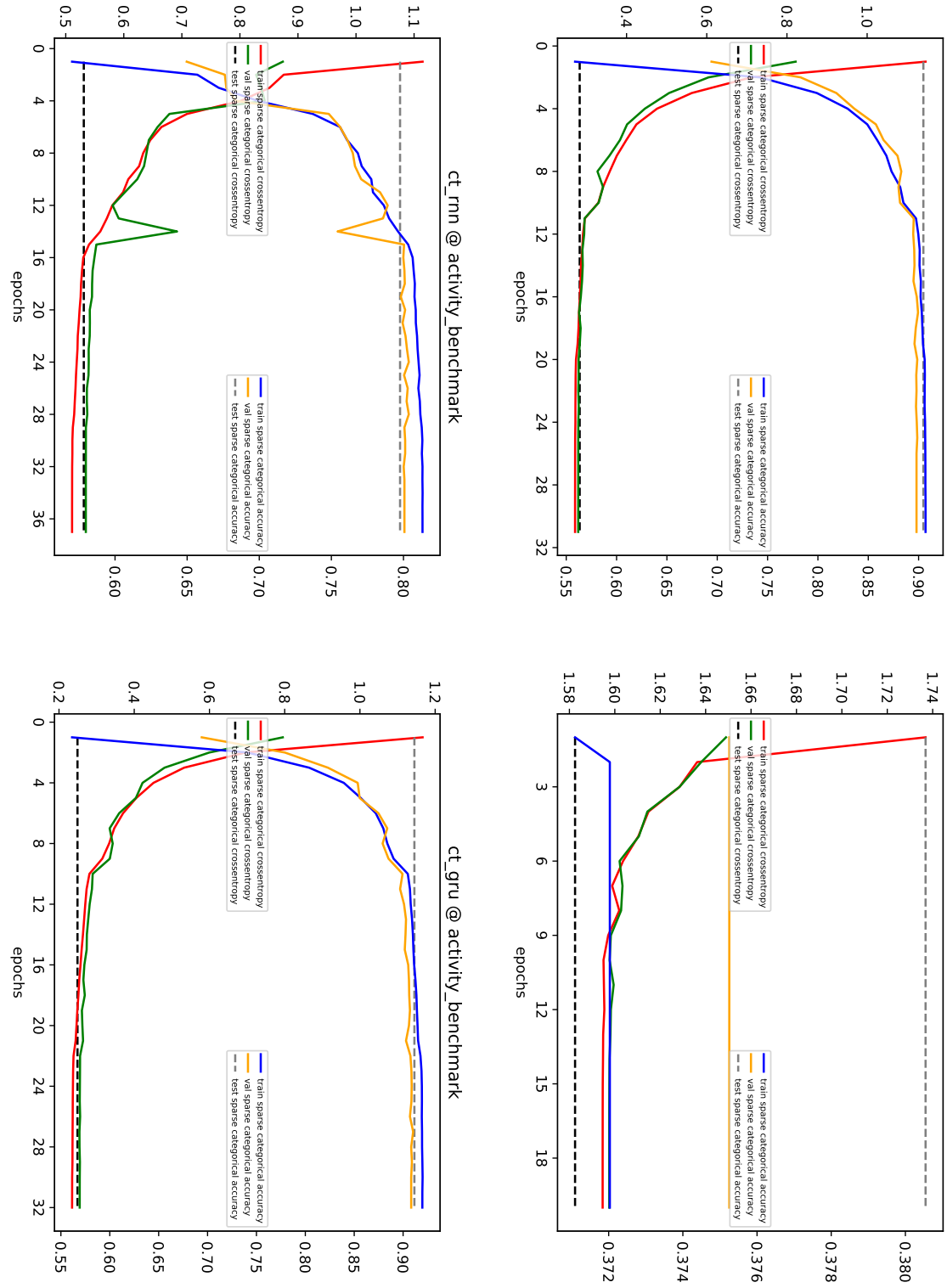

Figure 6.3: individual training plots for the Activity Benchmark on the second run - part 3

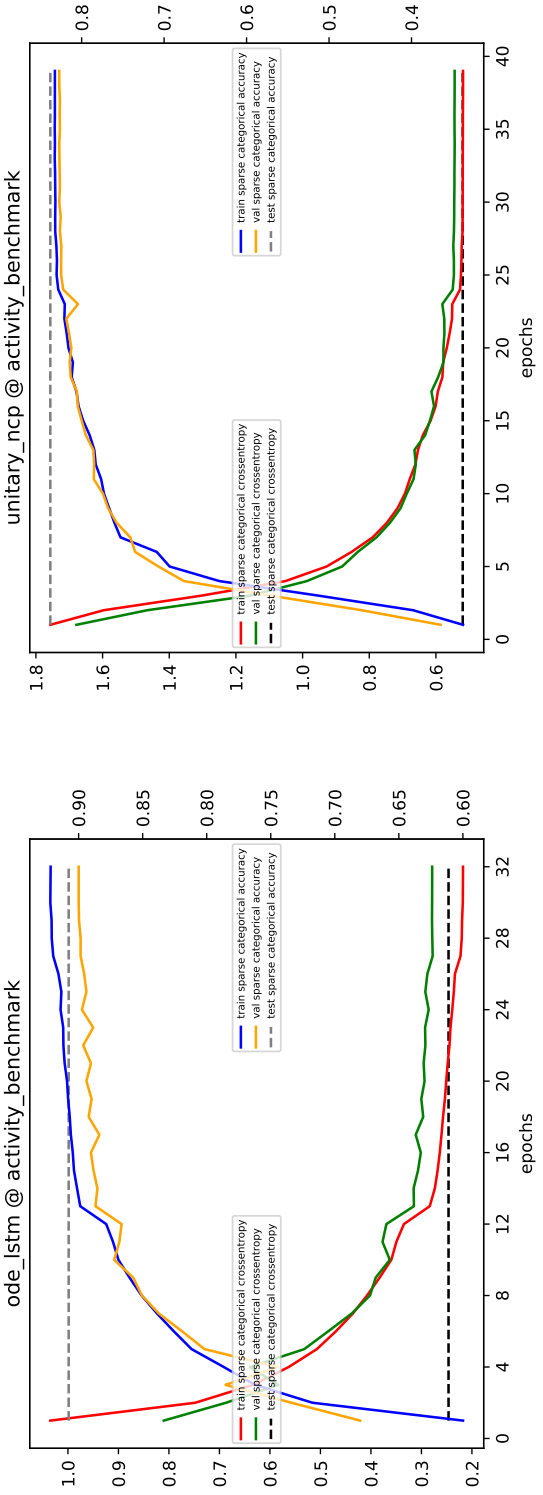

Figure 6.4: individual training plots for the Activity Benchmark on the second run - part 4

6.1.2 Add Benchmark

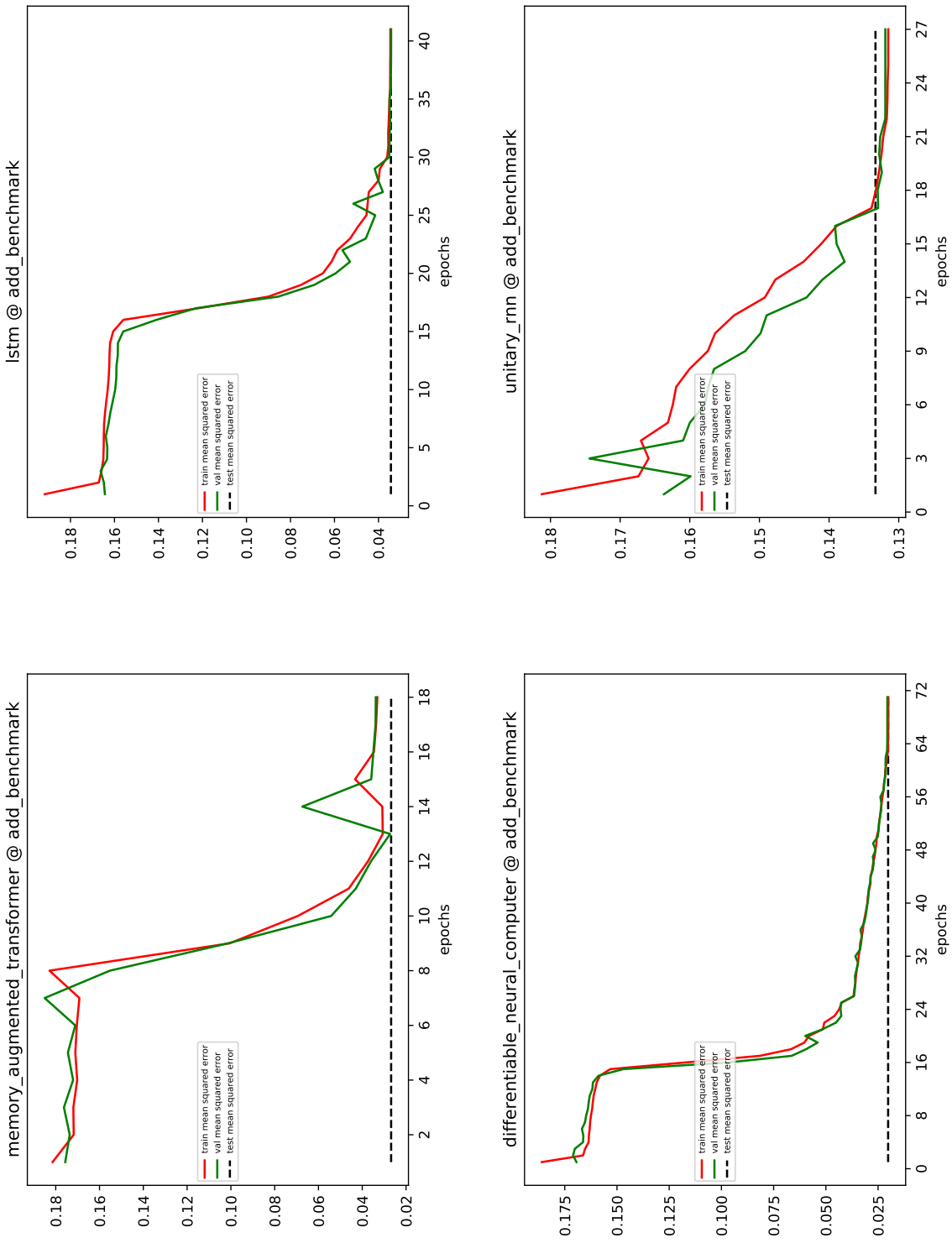

Figure 6.5: individual training plots for the Add Benchmark on the second run - part 1

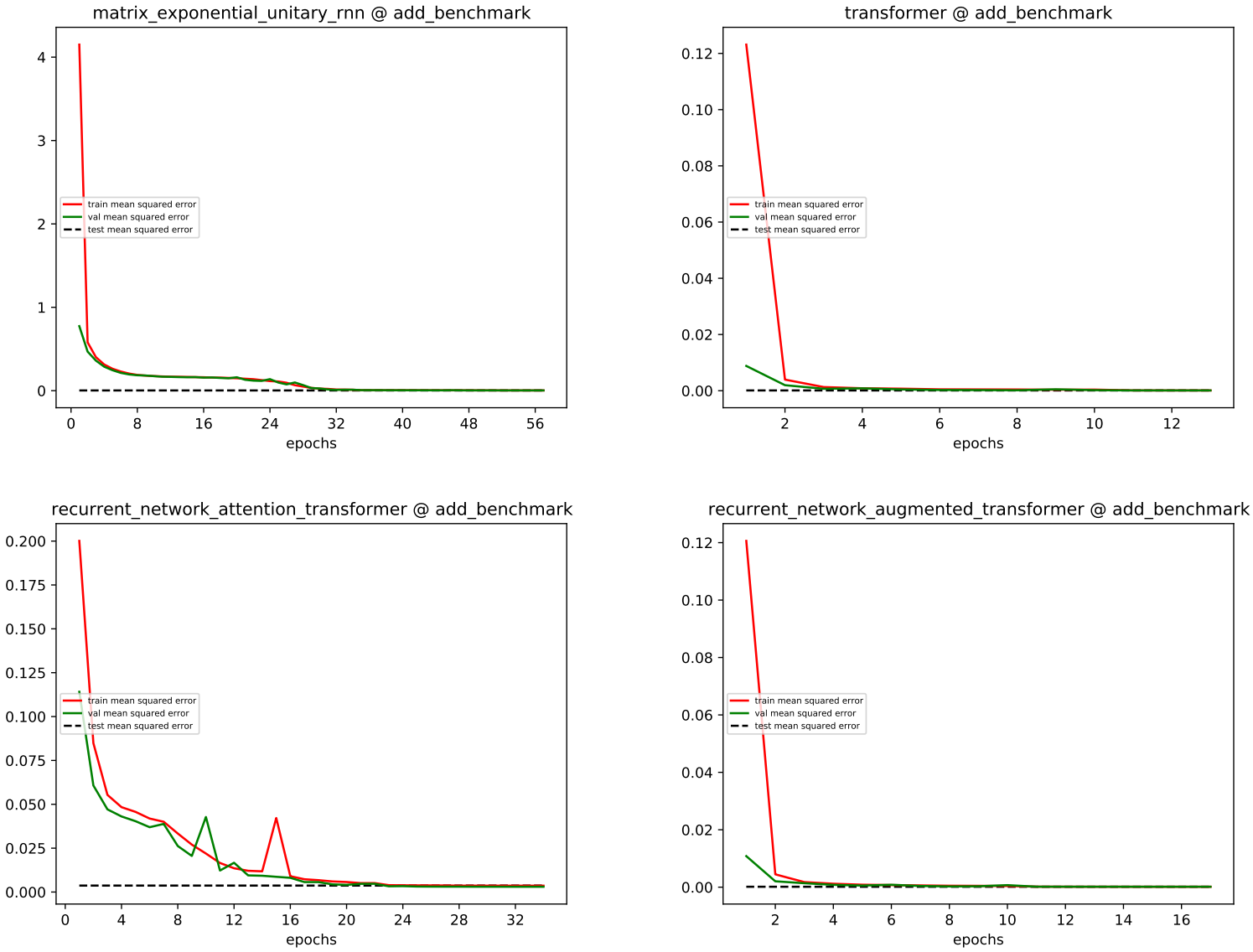

Figure 6.6: individual training plots for the Add Benchmark on the second run - part 2

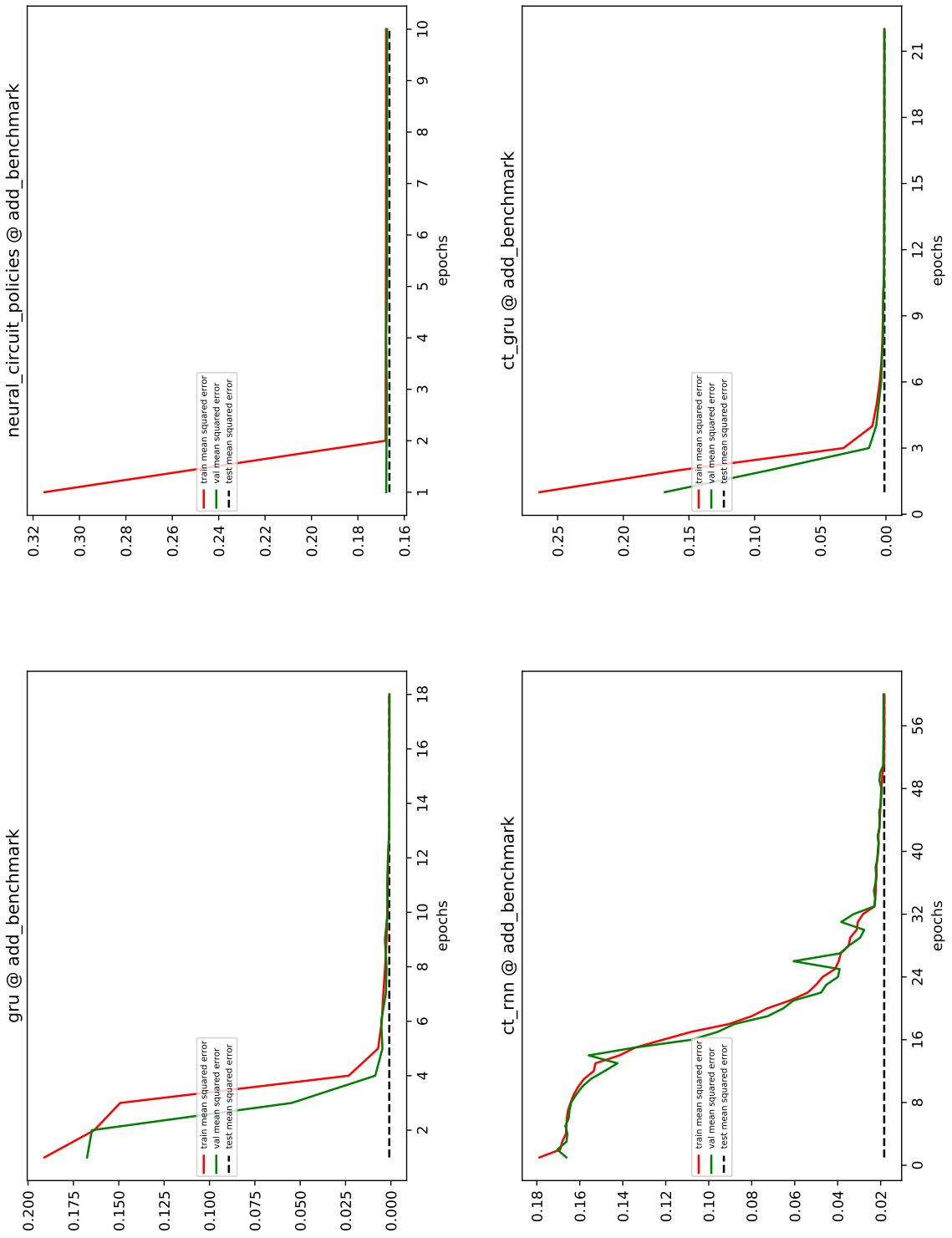

Figure 6.7: individual training plots for the Add Benchmark on the second run - part 3

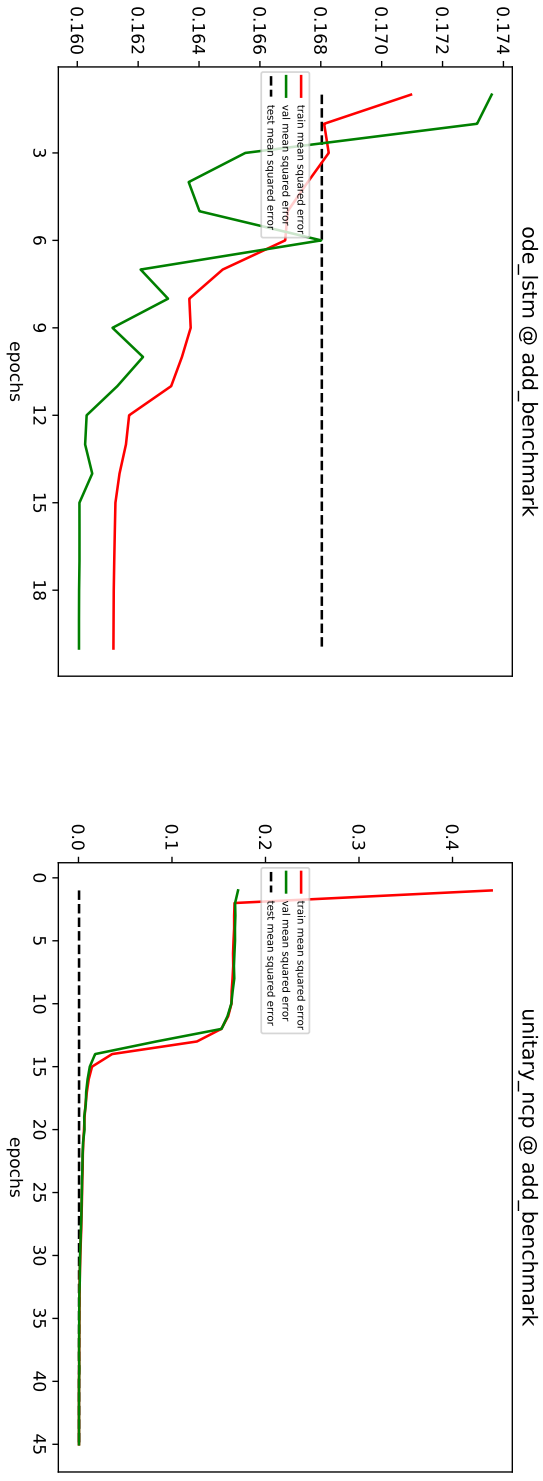

Figure 6.8: individual training plots for the Add Benchmark on the second run - part 4

6.1.3 Walker Benchmark

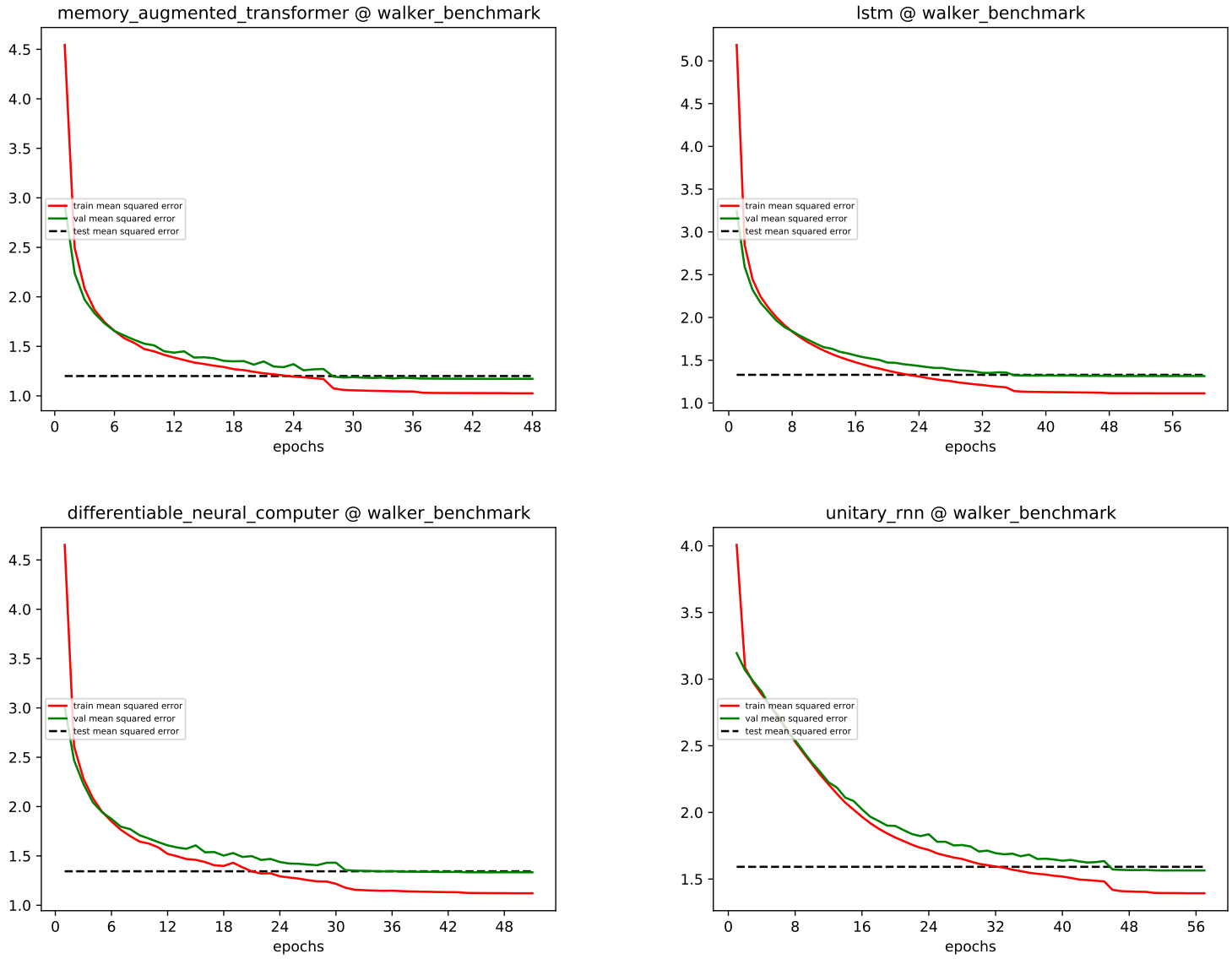

Figure 6.9: individual training plots for the Walker Benchmark on the second run - part 1

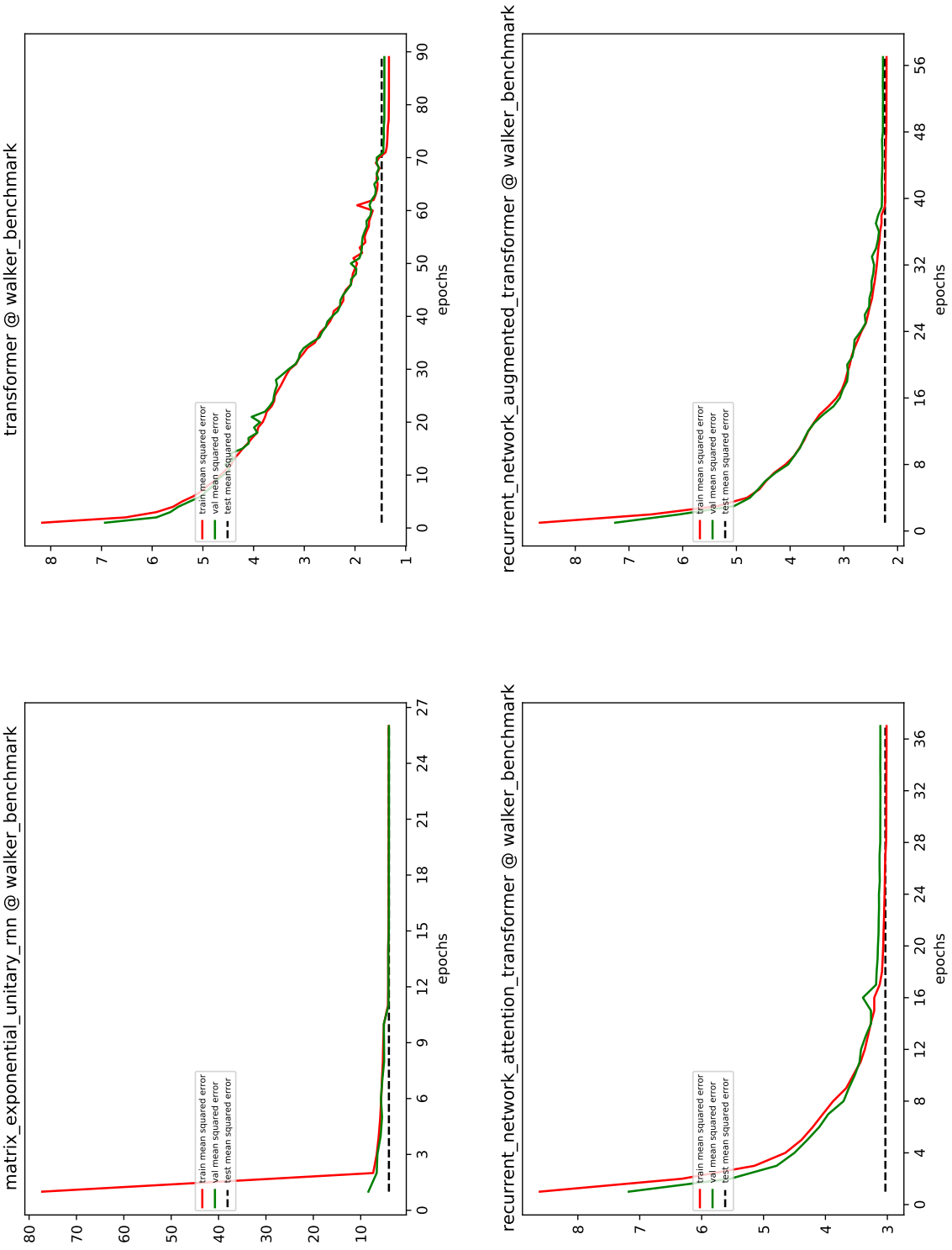

Figure 6.10: individual training plots for the Walker Benchmark on the second run - part 2

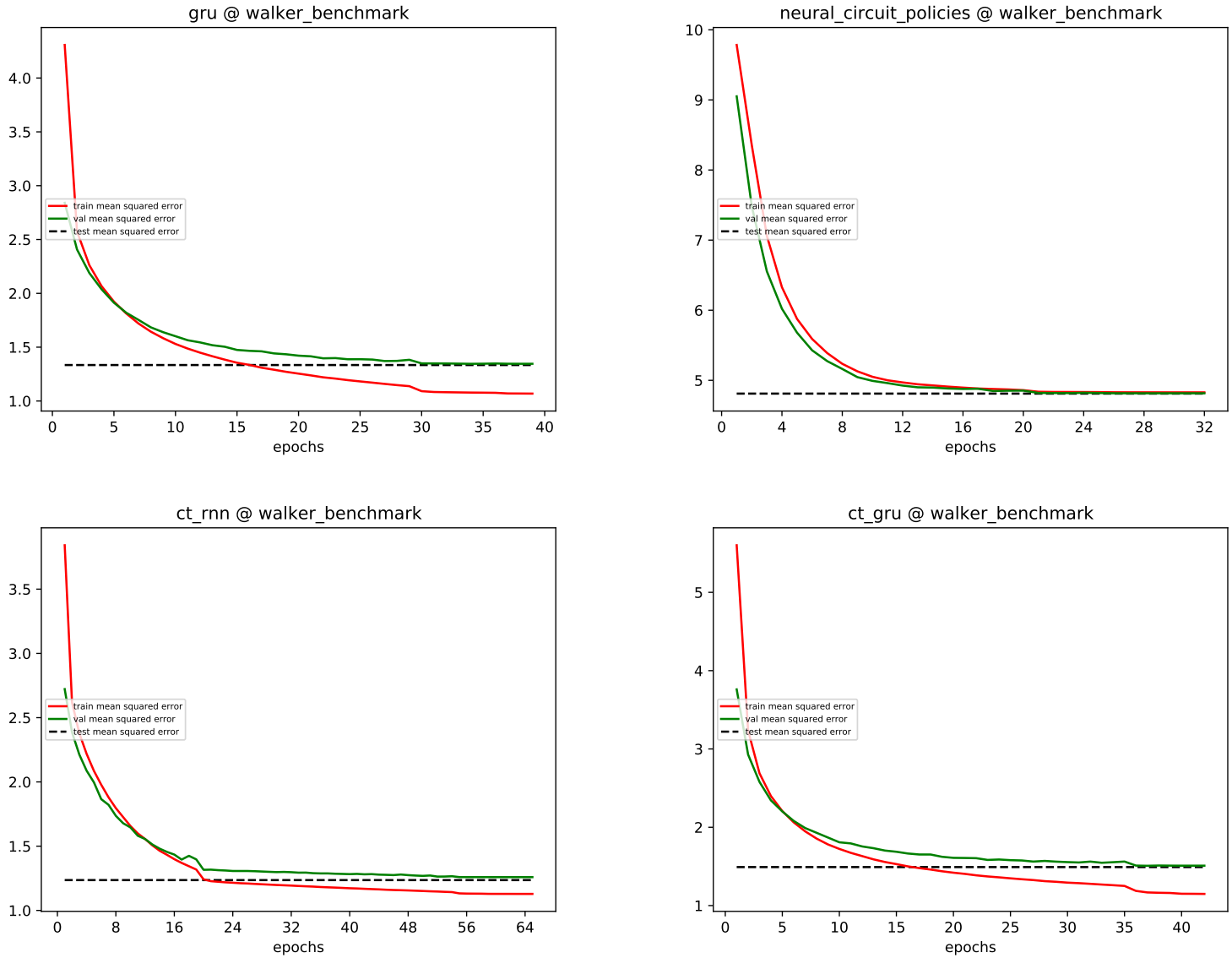

Figure 6.11: individual training plots for the Walker Benchmark on the second run - part 3

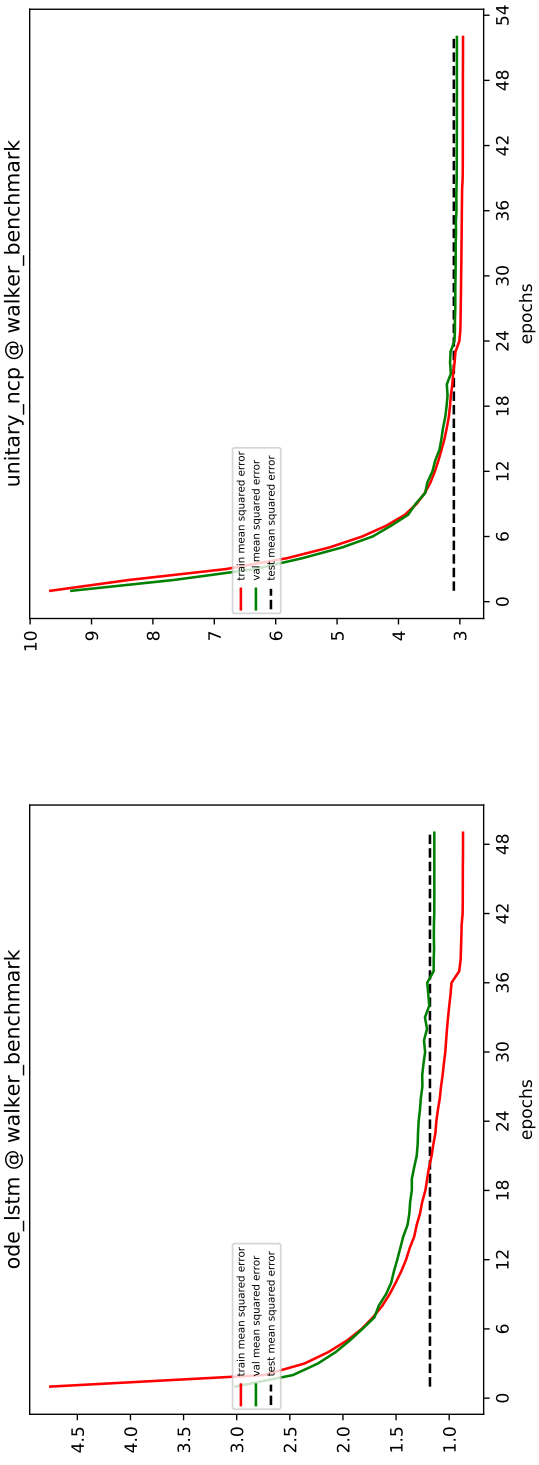

Figure 6.12: individual training plots for the Walker Benchmark on the second run - part 4

6.1.4 Memory Benchmark

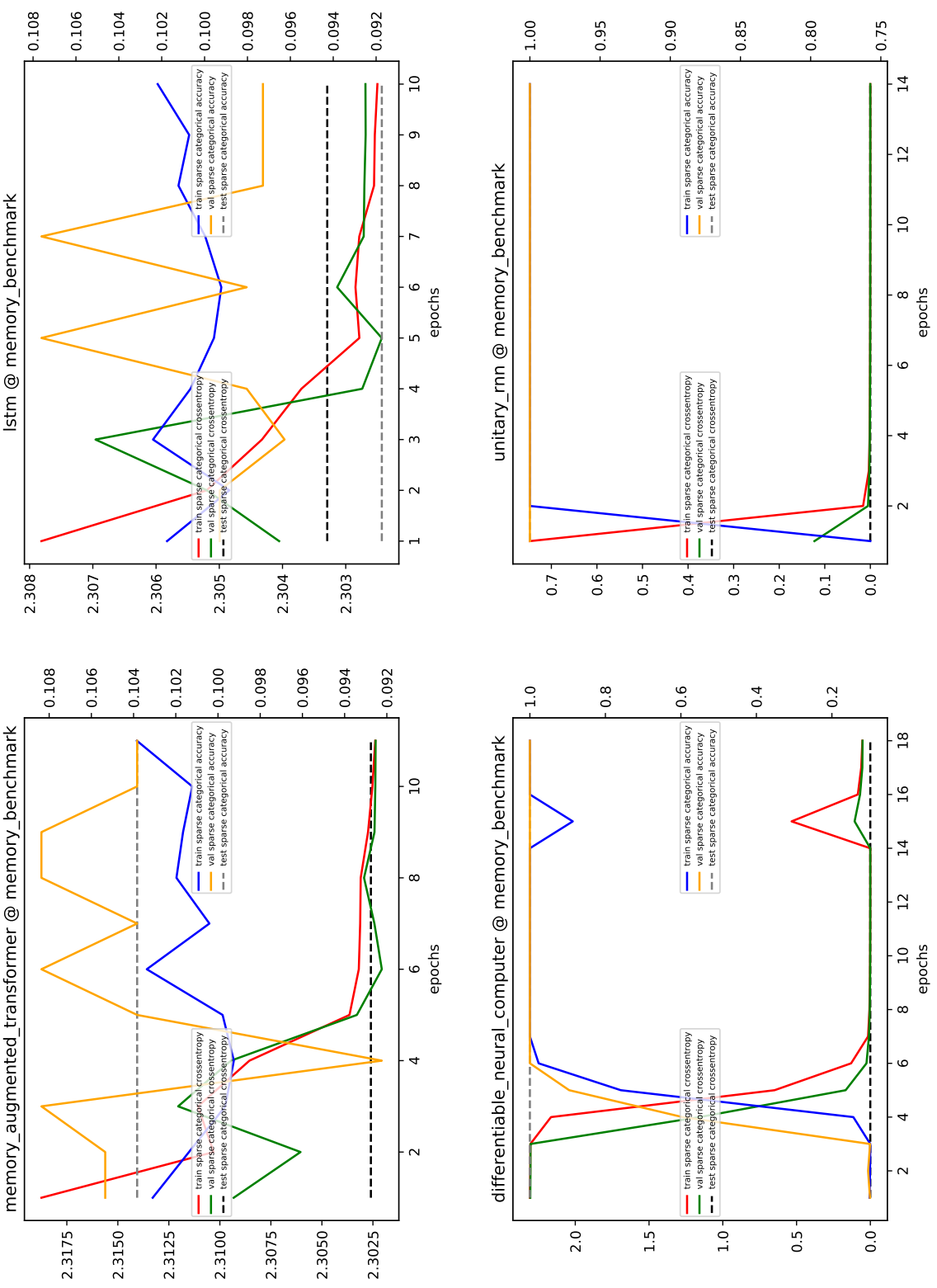

Figure 6.13: individual training plots for the Memory Benchmark on the second run - part 1

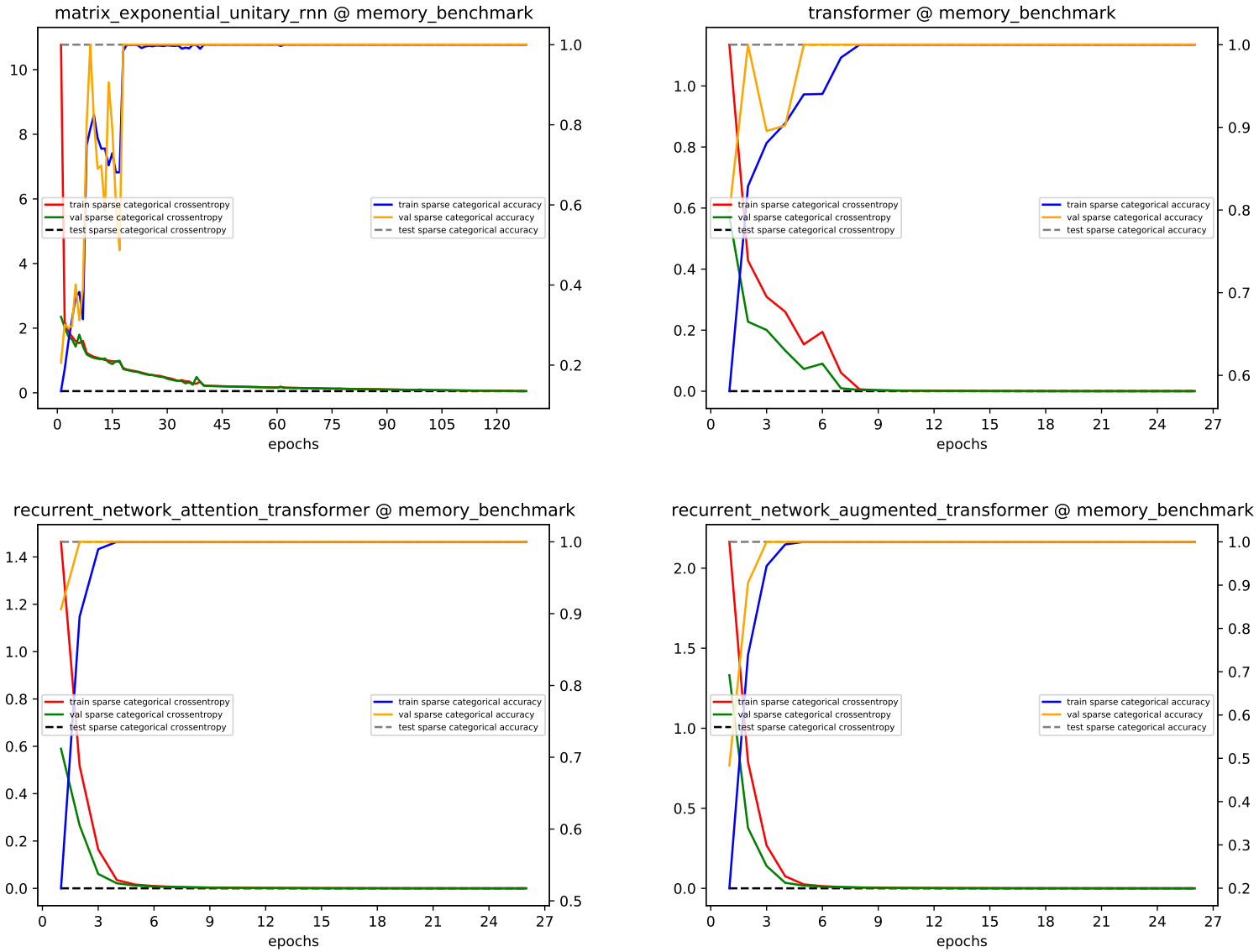

Figure 6.14: individual training plots for the Memory Benchmark on the second run - part 2

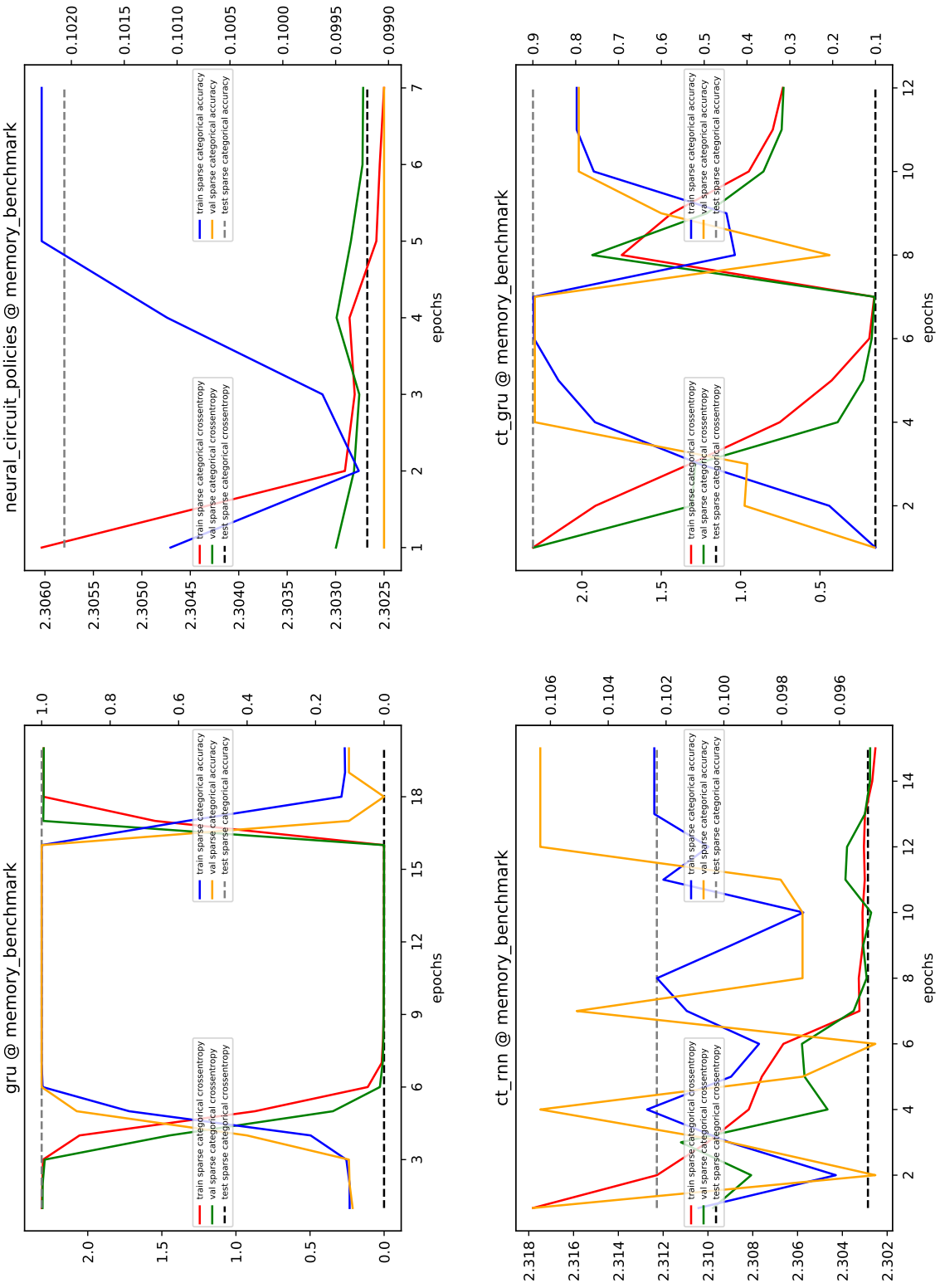

Figure 6.15: individual training plots for the Memory Benchmark on the second run - part 3

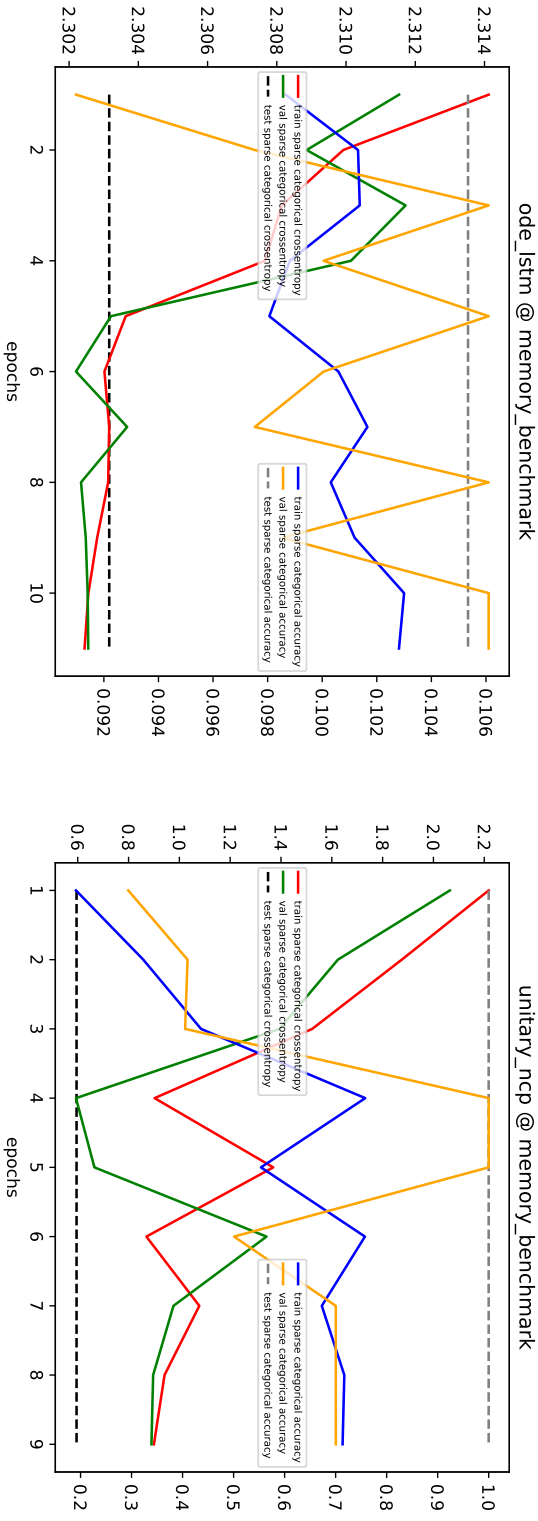

Figure 6.16: individual training plots for the Memory Benchmark on the second run - part 4

6.1.5 MNIST Benchmark

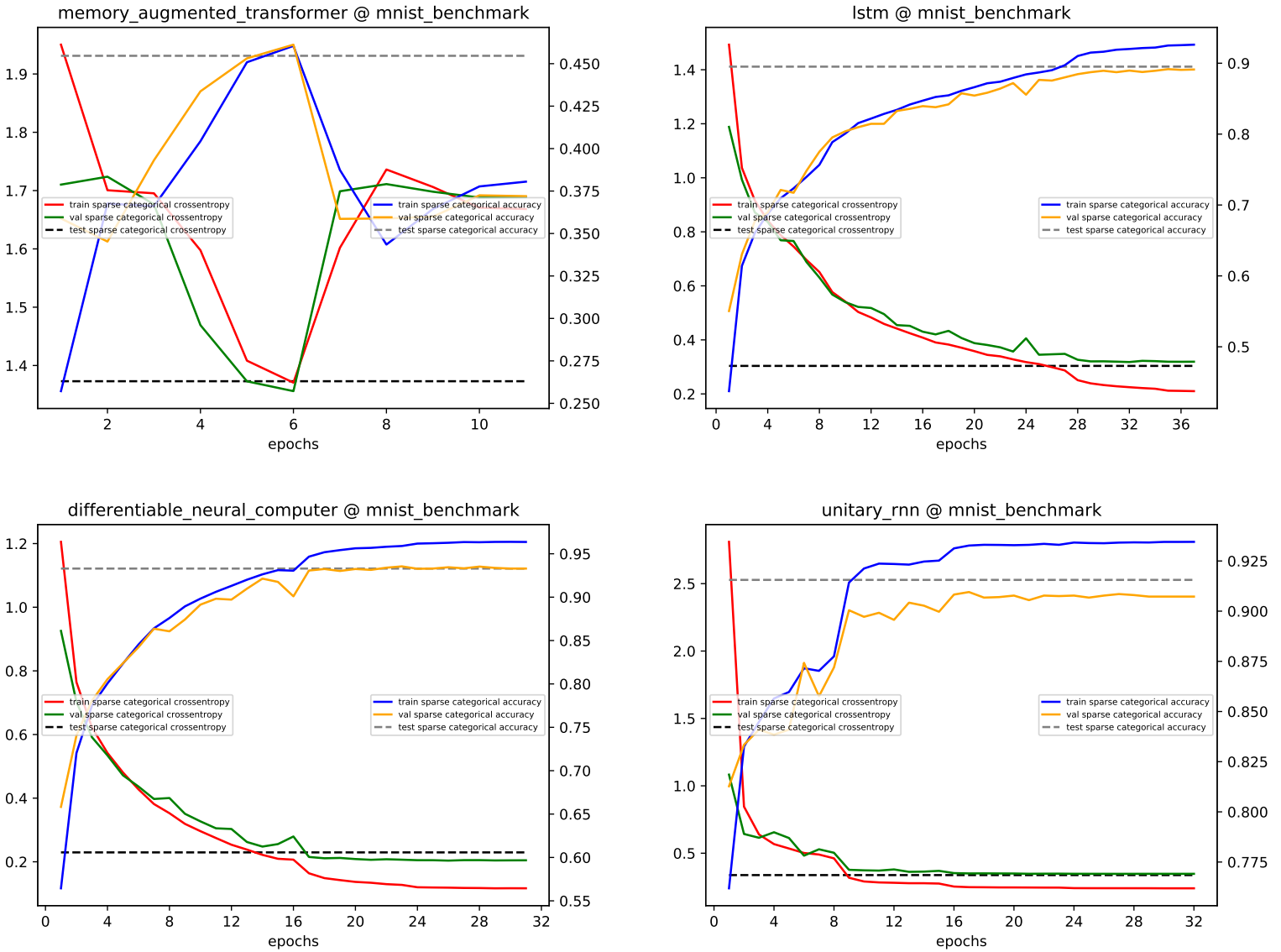

Figure 6.17: individual training plots for the MNIST Benchmark on the second run - part 1

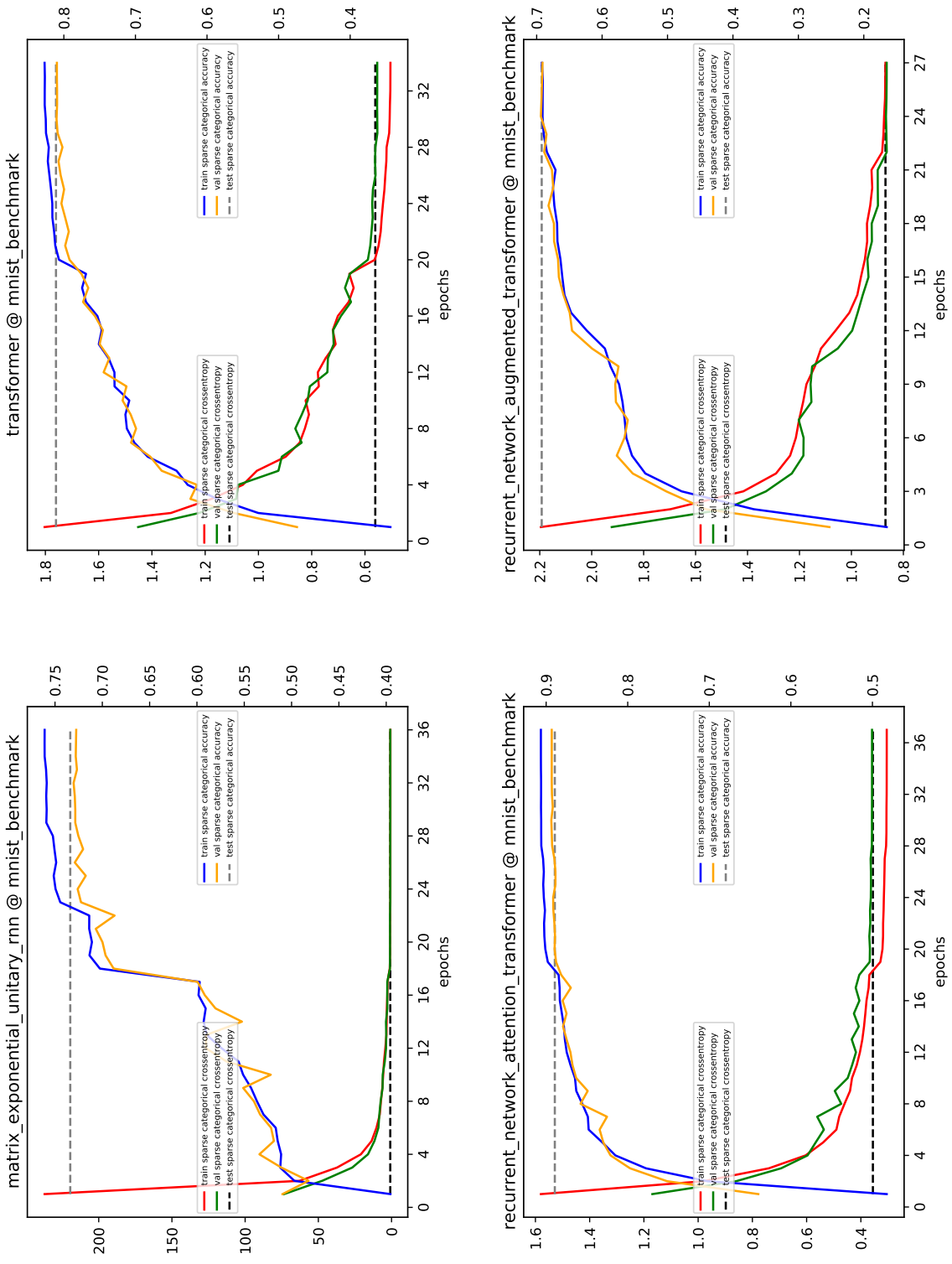

Figure 6.18: individual training plots for the MNIST Benchmark on the second run - part 2

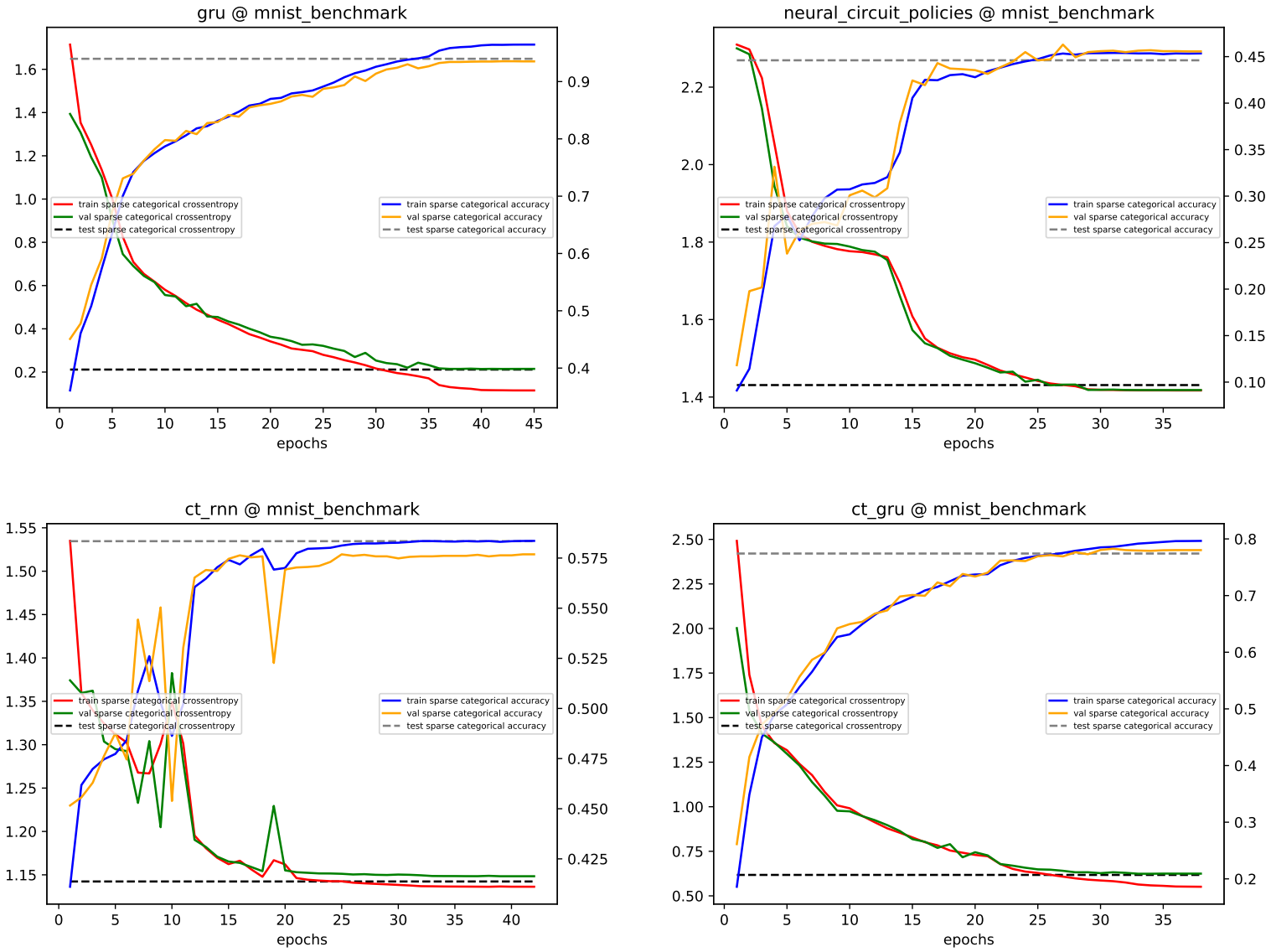

Figure 6.19: individual training plots for the MNIST Benchmark on the second run - part 3

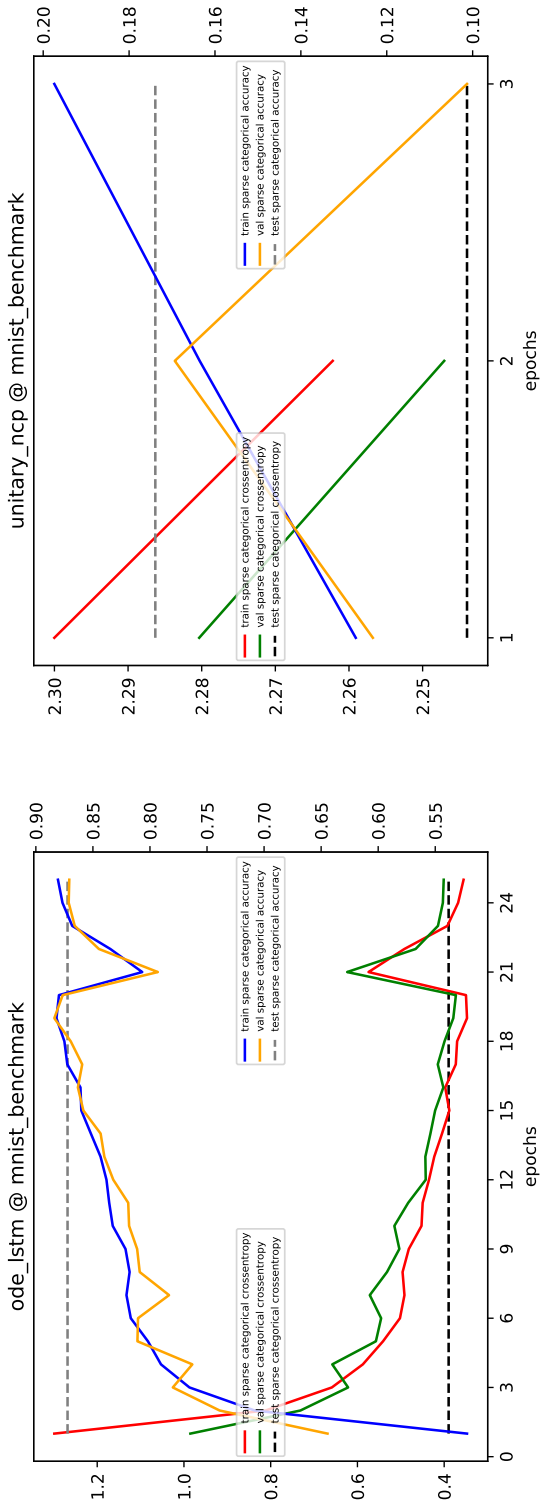

Figure 6.20: individual training plots for the MNIST Benchmark on the second run - part 4

6.2 Benchmark Code

6.2.1 benchmark.py

```
import abc
import argparse
import os
import shutil
import time

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np
import pandas as pd
import tensorflow as tf

import experiments.models.model_factory as model_factory

BENCHMARK_NAMES = ['cell', 'activity', 'add', 'memory', 'mnist'
    , 'walker']


class Benchmark(abc.ABC):
    def __init__(self, name, parser_configs):
        self.name = name
        assert self.name in BENCHMARK_NAMES
        self.args = self.get_args(parser_configs)
        self.saved_model_dir, self.tensorboard_dir, self.
            supplementary_data_dir, self.result_dir, self.
            visualization_dir = self.create_directories()
        self.input_data, self.output_data, self.output_size =
            self.get_data_and_output_size()
        self.data_samples = self.
            shuffle_data_and_return_sample_amount()
        self.inputs = tuple((tf.keras.Input(shape=x.shape[1:],
            batch_size=self.args.batch_size) for x in self.
            input_data))
        self.test_samples, self.validation_samples, self.
            training_samples = self.compute_sample_partition()
        self.test_input_data, self.validation_input_data, self.
            training_input_data = self.process_data(self.
            input_data)
        self.test_output_data, self.validation_output_data,
            self.training_output_data = self.process_data(self.
```

```
            output_data)
        self.train_and_test()

    @staticmethod
    def get_args(parser_configs):
        parser = argparse.ArgumentParser()
        parser.add_argument('--model', default=model_factory.
            MODEL_ARGUMENTS[0], type=str)
        parser.add_argument('--epochs', default=128, type=int)
        parser.add_argument('--batch_size', default=128, type=
            int)
        parser.add_argument('--optimizer_name', default='adam',
             type=str)
        parser.add_argument('--learning_rate', default=1E-3,
            type=float)
        parser.add_argument('--use_saved_model', default=False,
             type=bool)
        parser.add_argument('--debug', default=False, type=bool
            )
        parser.add_argument('--validation_data_percentage',
            default=0.1, type=float)
        parser.add_argument('--test_data_percentage', default
            =0.1, type=float)
        parser.add_argument('--no_improvement_lr_patience',
            default=2, type=int)
        parser.add_argument('--no_improvement_abort_patience',
            default=5, type=int)
        parser.add_argument('--min_delta', default=1E-4, type=
            float)
        parser.add_argument('--saved_model_folder_name',
            default='saved_models', type=str)
        parser.add_argument('--tensorboard_folder_name',
            default='tensorboard', type=str)
        parser.add_argument('--supplementary_data_folder_name',
             default='supplementary_data', type=str)
        parser.add_argument('--result_folder_name', default='
            results', type=str)
        parser.add_argument('--visualization_folder_name',
            default='visualizations', type=str)
        parser.add_argument('--use_time_input', default=False,
            type=bool)
        for parser_config in parser_configs:
            argument_name, default, cls = parser_config
```

```
            parser.add_argument(argument_name, default=default,
                type=cls)
        return parser.parse_args()

    def create_directories(self):
        project_directory = os.getcwd()
        saved_model_directory = os.path.join(project_directory,
            self.args.saved_model_folder_name, self.name)
        tensorboard_directory = os.path.join(project_directory,
            self.args.tensorboard_folder_name, self.name)
        supplementary_data_directory = os.path.join(
            project_directory, self.args.
            supplementary_data_folder_name, self.name)
        result_directory = os.path.join(project_directory, self
            .args.result_folder_name, self.name)
        visualization_directory = os.path.join(
            project_directory, self.args.
            visualization_folder_name, self.name)
        return saved_model_directory, tensorboard_directory,
            supplementary_data_directory, result_directory,
            visualization_directory

    @abc.abstractmethod
    def get_data_and_output_size(self):
        raise NotImplementedError

    def shuffle_data_and_return_sample_amount(self):
        data_samples = None
        random_integer = np.random.randint(2 ** 30)
        for dataset in self.input_data + self.output_data:
            np.random.default_rng(random_integer).shuffle(
                dataset)
            data_samples = len(dataset)
        for dataset in self.input_data + self.output_data:
            assert data_samples == len(dataset)
        return data_samples

    def compute_sample_partition(self):
        test_samples = int(self.data_samples * self.args.
            test_data_percentage)
        test_samples -= test_samples % self.args.batch_size
        validation_samples = int(self.data_samples * self.args.
            validation_data_percentage)
```

```
        validation_samples -= validation_samples % self.args.
            batch_size
        training_samples = self.data_samples - test_samples -
            validation_samples
        training_samples -= training_samples % self.args.
            batch_size
        assert test_samples > 0 and validation_samples > 0 and
            training_samples > 0
        return test_samples, validation_samples,
            training_samples

    def process_data(self, data):
        test_data = tuple((x[slice(None, self.test_samples)]
            for x in data))
        validation_data = tuple((x[slice(self.test_samples,
            self.test_samples + self.validation_samples)] for x
            in data))
        training_data = tuple((x[slice(self.test_samples + self
            .validation_samples, self.test_samples + self.
            validation_samples + self.training_samples)] for x
            in data))
        return test_data, validation_data, training_data

    def check_directories(self):
        shutil.rmtree(os.path.join(self.tensorboard_dir, self.
            args.model), ignore_errors=True)
        for model_name in model_factory.MODEL_ARGUMENTS:
            os.makedirs(os.path.join(self.saved_model_dir,
                model_name), exist_ok=True)
            os.makedirs(os.path.join(self.tensorboard_dir,
                model_name), exist_ok=True)
            os.makedirs(os.path.join(self.result_dir,
                model_name), exist_ok=True)
            os.makedirs(os.path.join(self.visualization_dir),
                exist_ok=True)

    @staticmethod
    def correct_names(names, train, model):
        loss_name = model.loss.name
        lr_name = 'learning rate'
        corrected_names = []
        for name in names:
            if train:
```

```
                if not name.startswith('val'):
                    name = 'train ' + name
            else:
                name = 'test ' + name
            corrected_names.append(name.replace('loss',
                loss_name).replace('lr', lr_name).replace('_', '
                '))
        return corrected_names

    def create_and_save_tables(self, model, fit_result,
        evaluate_result, training_duration):
        fit_results = list(fit_result.history.items())
        fit_header = self.correct_names([x[0] for x in
            fit_results], train=True, model=model)
        fit_data = np.array([x[1] for x in fit_results])
        fit_table = pd.DataFrame(data=fit_data.T, columns=
            fit_header)
        fit_table.to_csv(os.path.join(self.result_dir, self.
            args.model, 'training.csv'), index=False)
        fit_table.drop(fit_table.columns[-1], axis=1, inplace=
            True)
        evaluate_results = list(evaluate_result.items())
        evaluate_header = self.correct_names([x[0] for x in
            evaluate_results], train=False, model=model)
        evaluate_data = np.array([x[1] for x in
            evaluate_results])
        evaluate_table = pd.DataFrame(data=np.expand_dims(
            evaluate_data, 0), columns=evaluate_header)
        evaluate_table.insert(0, 'model', self.args.model)
        evaluate_table.insert(1, 'trainable parameters', np.sum
            ([np.prod(x.shape) for x in model.
            trainable_variables]))
        evaluate_table.insert(2, 'training duration total',
            training_duration)
        evaluate_table.insert(3, 'training duration per epoch',
             training_duration / fit_table.shape[0])
        evaluate_table.insert(4, 'epochs', fit_table.shape[0])
        evaluate_table.to_csv(os.path.join(self.result_dir,
            self.args.model, 'testing.csv'), index=False)
        evaluate_table.drop(evaluate_table.columns[:5], axis=1,
             inplace=True)
        return fit_table, evaluate_table
```

```
def create_visualization(self, fit_table, evaluate_table):
    x_data = np.array(range(1, fit_table.shape[0] + 1))
    figure, first_axis = plt.subplots()
    first_axis.set_xlabel('epochs')
    first_axis.xaxis.set_major_locator(ticker.MaxNLocator(
        integer=True))
    first_axis.set_title(f'{self.args.model} @ {self.name}
        _benchmark')
    first_axis.set_prop_cycle(color=['red', 'green'])
    if self.args.metric_name == '':
        axes = [first_axis]
    else:
        second_axis = first_axis.twinx()
        second_axis.set_prop_cycle(color=['blue', 'orange'
            ])
        axes = [first_axis, second_axis]
    hline_colors = ['black', 'grey']
    legend_positions = ['center left', 'center right']
    for index, column in enumerate(fit_table.columns):
        axes[index % len(axes)].plot(x_data, fit_table[
            column].tolist(), label=column)
    for index, column in enumerate(evaluate_table.columns):
        axes[index % len(axes)].hlines(evaluate_table[
            column].tolist(), x_data[0], x_data[-1], label=
            column, linestyles='dashed', colors=hline_colors
            [index % len(hline_colors)])
    for index, axis in enumerate(axes):
        legend = axis.legend(loc=legend_positions[index %
            len(legend_positions)], prop={'size': 6})
        legend.remove()
        axes[-1].add_artist(legend)
    plt.savefig(os.path.join(self.visualization_dir, f'{
        self.args.model}.pdf'))
    plt.close()

def accumulate_data(self):
    testing_data = []
    for model_name in model_factory.MODEL_ARGUMENTS:
        test_results_path = os.path.join(self.result_dir,
            model_name, 'testing.csv')
        if os.path.exists(test_results_path):
            testing_table = pd.read_csv(test_results_path)
            testing_data.append(testing_table)
```

```
            merged_testing_table = pd.concat(testing_data)
            merged_testing_table.sort_values(merged_testing_table.
                columns[5], inplace=True)
            merged_testing_table.to_csv(os.path.join(self.
                result_dir, 'merged_results.csv'), index=False)
            val_loss_data = []
            val_loss_column = ''
            for model_name in model_factory.MODEL_ARGUMENTS:
                training_results_path = os.path.join(self.
                    result_dir, model_name, 'training.csv')
                if os.path.exists(training_results_path):
                    training_table = pd.read_csv(
                        training_results_path)
                    val_loss_column = training_table.columns[1] if
                        self.args.metric_name == '' else
                        training_table.columns[2]
                    val_loss_data.append((model_name,
                        training_table[val_loss_column].tolist()))
            max_len = max([len(x[1]) for x in val_loss_data])
            x_data = np.array(range(1, max_len + 1))
            fig, axis = plt.subplots()
            axis.set_title(f'{val_loss_column} @ {self.name}
                _benchmark')
            axis.set_xlabel('epochs')
            axis.xaxis.set_major_locator(ticker.MaxNLocator(integer
                =True))
            color_cycle = plt.rcParams['axes.prop_cycle'].by_key()[
                'color']
            axis.set_prop_cycle(color=color_cycle * 2, linestyle=['
                solid'] * len(color_cycle) + ['dashed'] * len(
                color_cycle))
            axis.set_yscale('log')
            for model_name, val_losses in val_loss_data:
                axis.plot(x_data[:len(val_losses)], val_losses,
                    label=model_name)
            axis.legend(loc='upper right', prop={'size': 6})
            plt.savefig(os.path.join(self.visualization_dir, '
                merged_visualizations.pdf'))
            plt.close()

        def train_and_test(self):
            self.check_directories()
            model_name = self.args.model
```

```
assert model_name in model_factory.MODEL_ARGUMENTS
model_save_location = os.path.join(self.saved_model_dir
    , model_name)
tensorboard_save_location = os.path.join(self.
    tensorboard_dir, model_name)
if self.args.use_saved_model:
    model = tf.keras.models.load_model(
        model_save_location)
else:
    inputs_slice = slice(None) if self.args.
        use_time_input or len(self.inputs) == 1 else
        slice(-1)
    model = tf.keras.Model(inputs=self.inputs,
                           outputs=model_factory.
                               get_model_output_by_name(
                               self.args.model, self.
                               output_size, self.inputs[
                               inputs_slice]),
                           name=model_name)
    optimizer = tf.keras.optimizers.get({'class_name':
        self.args.optimizer_name,
                                         'config': {'
                                             learning_rate
                                             ': self.
                                             args.
                                             learning_rate
                                             }})
    loss = tf.keras.losses.get({'class_name': self.args
        .loss_name,
                                'config': self.args.
                                    loss_config})
    if self.args.metric_name == '':
        metric = None
    else:
        metric = tf.keras.metrics.get(self.args.
            metric_name)
    model.compile(optimizer=optimizer, loss=loss,
        metrics=metric, run_eagerly=self.args.debug)
model.summary()
if self.args.debug:
    sample_output = model.predict(tuple((x[:self.args.
        batch_size] for x in self.training_input_data)))
```

```
            sample_loss = model.loss(tuple((x[:self.args.
                batch_size] for x in self.training_output_data))
                , sample_output).numpy()
            assert not tf.math.is_nan(sample_loss)
        training_start = time.time()
        fit_result = model.fit(
            x=self.training_input_data,
            y=self.training_output_data,
            batch_size=self.args.batch_size,
            epochs=self.args.epochs,
            validation_data=(self.validation_input_data, self.
                validation_output_data),
            callbacks=(tf.keras.callbacks.ModelCheckpoint(
                model_save_location, save_best_only=True),
                        tf.keras.callbacks.EarlyStopping(
                            patience=self.args.
                            no_improvement_abort_patience,
                            min_delta=self.args.min_delta),
                        tf.keras.callbacks.TerminateOnNaN(),
                        tf.keras.callbacks.ReduceLROnPlateau(
                            patience=self.args.
                            no_improvement_lr_patience, min_delta
                            =self.args.min_delta),
                        tf.keras.callbacks.TensorBoard(log_dir=
                            tensorboard_save_location)))
        training_end = time.time()
        training_duration = training_end - training_start
        model = tf.keras.models.load_model(model_save_location)
        evaluate_result = model.evaluate(
            x=self.test_input_data,
            y=self.test_output_data,
            batch_size=self.args.batch_size,
            callbacks=(tf.keras.callbacks.TensorBoard(log_dir=
                tensorboard_save_location)),
            return_dict=True)
        fit_table, evaluate_table = self.create_and_save_tables
            (model, fit_result, evaluate_result,
            training_duration)
        self.create_visualization(fit_table, evaluate_table)
        self.accumulate_data()
```

Listing 6.1: benchmark.py

6.2.2 activity_benchmark.py

```
import os

import numpy as np
import pandas as pd

import experiments.benchmarks.benchmark as benchmark


class ActivityBenchmark(benchmark.Benchmark):
    def __init__(self):
        super().__init__('activity',
                         (('--sequence_length', 64, int),
                          ('--max_samples', 40_000, int),
                          ('--sample_distance', 4, int),
                          ('--loss_name', '
                              SparseCategoricalCrossentropy',
                              str),
                          ('--loss_config', {'from_logits':
                              True}, dict),
                          ('--metric_name', '
                              SparseCategoricalAccuracy', str)))

    def get_data_and_output_size(self):
        sequence_length = self.args.sequence_length
        max_samples = self.args.max_samples
        sample_distance = self.args.sample_distance
        activity_table = pd.read_csv(os.path.join(self.
            supplementary_data_dir, 'activity.csv'), header=None
            )
        sensor_inputs = []
        time_inputs = []
        activity_outputs = []
        for activity_marker in activity_table[0].unique():
            activity_series = activity_table[activity_table[0]
                == activity_marker].iloc[:, 1:]
            for start_index in range(0, len(activity_series) -
                sequence_length + 1, sample_distance):
                current_sequence = np.array(activity_series[
                    start_index:start_index + sequence_length])
                sensor_inputs.append(current_sequence[:, 1:8])
                time_inputs.append(current_sequence[:, :1])
                activity_outputs.append(current_sequence[-1,
                    8:])
```

```
        return (np.stack(sensor_inputs)[:max_samples], np.stack
            (time_inputs)[:max_samples]), (np.stack(
            activity_outputs)[:max_samples],), 7


ActivityBenchmark()
```

Listing 6.2: activity_benchmark.py

6.2.3 add_benchmark.py

```
import numpy as np

import experiments.benchmarks.benchmark as benchmark


class AddBenchmark(benchmark.Benchmark):
    def __init__(self):
        super().__init__('add',
                         (('--sequence_length', 100, int),
                          ('--samples', 40_000, int),
                          ('--loss_name', 'MeanSquaredError',
                              str),
                          ('--loss_config', {}, dict),
                          ('--metric_name', '', str)))

    def get_data_and_output_size(self):
        sequence_length = self.args.sequence_length
        assert sequence_length % 2 == 0
        samples = self.args.samples
        number_sequences = np.random.random((samples,
            sequence_length, 1))
        random_indices = np.random.randint(low=0, high=
            sequence_length // 2, size=2 * samples)
        row_indices = np.arange(samples)
        marker_sequences = np.zeros_like(number_sequences)
        marker_sequences[row_indices, random_indices[:samples]]
            = 1
        marker_sequences[row_indices, random_indices[samples:]
            + sequence_length // 2] = 1
        input_sequences = np.concatenate((number_sequences,
            marker_sequences), -1)
        time_sequences = np.ones((samples, sequence_length, 1))
        filtered_input_sequences = np.where(marker_sequences,
            number_sequences, 0)
```

```
        output_data = np.sum(filtered_input_sequences, 1)
        return (input_sequences, time_sequences), (output_data
            ,), 1


AddBenchmark()
```

Listing 6.3: add_benchmark.py

6.2.4 walker_benchmark.py

```
import os

import numpy as np

import experiments.benchmarks.benchmark as benchmark


class WalkerBenchmark(benchmark.Benchmark):
    def __init__(self):
        super().__init__('walker',
                         (('--skip_percentage', 0.1, float),
                          ('--frame_skip', False, bool),
                          ('--sequence_length', 64, int),
                          ('--max_samples', 40_000, int),
                          ('--sample_distance', 4, int),
                          ('--loss_name', 'MeanSquaredError',
                              str),
                          ('--loss_config', {}, dict),
                          ('--metric_name', '', str)))

    def get_data_and_output_size(self):
        max_samples = self.args.max_samples
        datasets = []
        for dataset_filename in [x for x in os.listdir(self.
            supplementary_data_dir) if x.endswith('.npy')]:
            dataset = np.load(os.path.join(self.
                supplementary_data_dir, dataset_filename))
            datasets.append((dataset[:-1, :].tolist(), dataset
                [1:, :].tolist()))
        lossy_data = []
        for input_dataset, output_dataset in datasets:
            lossy_input_dataset, lossy_time_dataset,
                lossy_output_dataset = [], [], []
            interval = 0
```

```
                for index in range(len(input_dataset)):
                    interval += 1
                    if np.random.random() > self.args.
                        skip_percentage or not self.args.frame_skip:
                        lossy_input_dataset.append(input_dataset[
                            index])
                        lossy_time_dataset.append([interval])
                        lossy_output_dataset.append(output_dataset[
                            index])
                        interval = 0
                lossy_data.append([lossy_input_dataset,
                    lossy_time_dataset, lossy_output_dataset])
            input_sequences = []
            time_sequences = []
            output_sequences = []
            for input_dataset, time_dataset, output_dataset in
                lossy_data:
                for start_index in range(0, len(input_dataset) -
                    self.args.sequence_length + 1, self.args.
                    sample_distance):
                    end_index = start_index + self.args.
                        sequence_length
                    input_sequences.append(input_dataset[
                        start_index:end_index])
                    time_sequences.append(time_dataset[start_index:
                        end_index])
                    output_sequences.append(output_dataset[
                        end_index - 1])
            return (np.array(input_sequences[:max_samples]), np.
                array(time_sequences[:max_samples])), (np.array(
                output_sequences[:max_samples]),), 17


WalkerBenchmark()
```

Listing 6.4: walker_benchmark.py

6.2.5 memory_benchmark.py

```
import numpy as np

import experiments.benchmarks.benchmark as benchmark


class MemoryBenchmark(benchmark.Benchmark):
```

```
    def __init__(self):
        super().__init__('memory',
                         (('--memory_length', 100, int),
                          ('--sequence_length', 1, int),
                          ('--category_amount', 10, int),
                          ('--samples', 40_000, int),
                          ('--loss_name', '
                              SparseCategoricalCrossentropy',
                              str),
                          ('--loss_config', {'from_logits':
                              True}, dict),
                          ('--metric_name', '
                              SparseCategoricalAccuracy', str)))

    def get_data_and_output_size(self):
        memory_length = self.args.memory_length
        sequence_length = self.args.sequence_length
        category_amount = self.args.category_amount
        samples = self.args.samples
        memory_sequence = np.random.randint(low=0, high=
            category_amount, size=(samples, sequence_length, 1))
        first_blank_sequence = category_amount * np.ones((
            samples, memory_length, 1))
        marker_sequence = np.random.randint(low=0, high=
            sequence_length, size=(samples, 1, 1))
        input_sequence = np.concatenate((memory_sequence,
            first_blank_sequence, marker_sequence), 1)
        time_sequence = np.ones_like(input_sequence)
        output_sequence = memory_sequence[np.arange(samples),
            np.squeeze(marker_sequence)]
        return (input_sequence, time_sequence), (
            output_sequence,), self.args.category_amount


MemoryBenchmark()
```

Listing 6.5: memory_benchmark.py

6.2.6 mnist_benchmark.py

```
import numpy as np
import tensorflow as tf

import experiments.benchmarks.benchmark as benchmark
```

```
class MnistBenchmark(benchmark.Benchmark):
    def __init__(self):
        super().__init__('mnist',
                         (('--max_samples', 40_000, int),
                          ('--loss_name', '
                              SparseCategoricalCrossentropy',
                              str),
                          ('--loss_config', {'from_logits':
                              True}, dict),
                          ('--metric_name', '
                              SparseCategoricalAccuracy', str)))

    def get_data_and_output_size(self):
        max_samples = self.args.max_samples
        data = tf.keras.datasets.mnist.load_data()
        input_data = np.reshape(np.concatenate((data[0][0],
            data[1][0])), (-1, 98, 8))
        time_data = np.ones_like(input_data)
        output_data = np.concatenate((data[0][1], data[1][1]))
            [..., np.newaxis]
        return (input_data[:max_samples], time_data[:
            max_samples]), (output_data[:max_samples],), 10


MnistBenchmark()
```

Listing 6.6: mnist_benchmark.py

6.2.7 cell_benchmark.py

```
import numpy as np

import experiments.benchmarks.benchmark as benchmark


class CellBenchmark(benchmark.Benchmark):
    def __init__(self):
        super().__init__('cell',
                         (('--memory_high_symbol', 1, int),
                          ('--memory_low_symbol', 0, int),
                          ('--memory_length', 128, int),
                          ('--cell_switches', 2, int),
                          ('--samples', 40_000, int),
```

```
                    ('--loss_name', 'MeanSquaredError',
                        str),
                    ('--loss_config', {}, dict),
                    ('--metric_name', '', str)))

    def get_data_and_output_size(self):
        memory_high_symbol = self.args.memory_high_symbol
        memory_low_symbol = self.args.memory_low_symbol
        memory_length = self.args.memory_length
        cell_switches = self.args.cell_switches
        samples = self.args.samples
        model_input = np.zeros((samples, (cell_switches + 1) *
            memory_length, 2))
        time_input = np.ones((samples, (cell_switches + 1) *
            memory_length, 1))
        model_output = np.zeros((samples, (cell_switches + 1) *
            memory_length, 2))
        for i in range(cell_switches + 1):
            even = int(i % 2 == 0)
            odd = int(i % 2 == 1)
            model_input[0::2, i * memory_length, odd] =
                memory_high_symbol
            model_input[0::2, i * memory_length, even] =
                memory_low_symbol
            model_output[0::2, i * memory_length:(i + 1) *
                memory_length, 0] = even * memory_high_symbol
            model_output[0::2, i * memory_length:(i + 1) *
                memory_length, 1] = odd * memory_high_symbol
            model_input[1::2, i * memory_length, even] =
                memory_high_symbol
            model_input[1::2, i * memory_length, odd] =
                memory_low_symbol
            model_output[1::2, i * memory_length:(i + 1) *
                memory_length, 0] = odd * memory_high_symbol
            model_output[1::2, i * memory_length:(i + 1) *
                memory_length, 1] = even * memory_high_symbol
        return (model_input, time_input), (model_output,), 2


CellBenchmark()
```

Listing 6.7: cell_benchmark.py

6.3 Model Code

6.3.1 model_fatory.py

```
import tensorflow as tf

import experiments.models.ct_gru as ct_gru
import experiments.models.ct_rnn as ct_rnn
import experiments.models.differentiable_neural_computer as dnc
import experiments.models.matrix_exponential_unitary_rnn as
    meurnn
import experiments.models.memory_augmented_transformer as mat
import experiments.models.memory_cell as memory_cell
import experiments.models.neural_circuit_policies as ncp
import experiments.models.ode_lstm as ode_lstm
import experiments.models.transformer as transformer
import experiments.models.unitary_ncp as uncp
import experiments.models.unitary_rnn as urnn

MODEL_ARGUMENTS = ['memory_cell', 'memory_augmented_transformer
    ', 'lstm',
                   'differentiable_neural_computer', '
                      unitary_rnn', '
                      matrix_exponential_unitary_rnn',
                   'transformer', '
                      recurrent_network_attention_transformer',
                       'recurrent_network_augmented_transformer
                      ',
                   'gru', 'neural_circuit_policies', 'ct_rnn',
                   'ct_gru', 'ode_lstm', 'unitary_ncp']


def get_concat_inputs(inputs):
    if isinstance(inputs, tuple):
        return tf.concat(inputs, -1)
    else:
        return inputs


def get_concat_input_shape(input_shape):
    if isinstance(input_shape[0], tuple):
        return sum([x[-1] for x in input_shape])
    else:
        return input_shape[-1]
```

```
def get_ct_gru_output(output_size, input_tensor):
    return tf.keras.layers.Dense(output_size)(
        tf.keras.layers.RNN(ct_gru.CTGRU(32))(input_tensor))


def get_ct_rnn_output(output_size, input_tensor):
    return tf.keras.layers.Dense(output_size)(
        tf.keras.layers.RNN(ct_rnn.CTRNNCell(128, 'rk4', 3))(
            input_tensor))


def get_ode_lstm_output(output_size, input_tensor):
    return tf.keras.layers.Dense(output_size)(
        tf.keras.layers.RNN(ode_lstm.ODELSTM(64))(input_tensor)
            )


def get_differentiable_neural_computer_output(output_size,
   input_tensor):
    return tf.keras.layers.RNN(dnc.DNC(output_size, 64, 16, 8,
       2))(input_tensor)


def get_unitary_rnn_output(output_size, input_tensor):
    return tf.keras.layers.Dense(output_size)(
        tf.math.real(tf.keras.layers.RNN(urnn.EUNNCell(128, 16)
            )(input_tensor)))


def get_unitary_ncp_output(output_size, input_tensor):
    return uncp.UnitaryNCP(32, 8, output_size)(input_tensor)


def get_matrix_exponential_unitary_rnn_output(output_size,
   input_tensor):
    return tf.keras.layers.RNN(meurnn.
       MatrixExponentialUnitaryRNN(128, output_size))(
       input_tensor)


def get_lstm_output(output_size, input_tensor):
```

```
        return tf.keras.layers.Dense(output_size)(
            tf.keras.layers.LSTM(64)(get_concat_inputs(input_tensor
                )))


    def get_gru_output(output_size, input_tensor):
        return tf.keras.layers.Dense(output_size)(
            tf.keras.layers.GRU(80)(get_concat_inputs(input_tensor)
                ))


    def get_transformer_output(output_size, input_tensor):
        return transformer.Transformer(token_amount=1, token_size=
            output_size, d_model=16, num_heads=2, d_ff=64,
                                       num_layers=2, dropout_rate
                                           =0, attention='mha')(
                                           input_tensor)


    def get_recurrent_network_attention_transformer_output(
        output_size, input_tensor):
        return transformer.Transformer(token_amount=1, token_size=
            output_size, d_model=8, num_heads=1, d_ff=32,
                                       num_layers=1, dropout_rate
                                           =0, attention='rna')(
                                           input_tensor)


    def get_recurrent_network_augmented_transformer_output(
        output_size, input_tensor):
        return transformer.Transformer(token_amount=1, token_size=
            output_size, d_model=8, num_heads=1, d_ff=32,
                                       num_layers=1, dropout_rate
                                           =0, attention='rnat')(
                                           input_tensor)


    def get_neural_circuit_policies_output(output_size,
        input_tensor):
        return ncp.NeuralCircuitPolicies(16, output_size)(
            input_tensor)
```

```
def get_memory_augmented_transformer_output(output_size,
    input_tensor):
    return tf.keras.layers.RNN(mat.
        MemoryAugmentedTransformerCell(output_size=output_size))
        (input_tensor)


def get_memory_cell_output(output_size, input_tensor):
    assert output_size == 2
    return tf.keras.layers.RNN(memory_cell.MemoryCell(),
        return_sequences=True)(input_tensor)


def get_model_output_by_name(model_name, output_size,
    input_tensor):
    return eval(f'get_{model_name}_output')(output_size,
        input_tensor if len(input_tensor) > 1 else input_tensor
        [0])
```

Listing 6.8: model_fatory.py

6.3.2 ct_rnn.py

```
"""
code taken from https://github.com/mlech26l/ode-lstms/blob/
    master/node_cell.py (slightly modified)
"""

import tensorflow as tf
import tensorflow_probability as tfp


@tf.keras.utils.register_keras_serializable()
class CTRNNCell(tf.keras.layers.AbstractRNNCell):
    def __init__(self, units, method, num_unfolds=None, tau=1,
        **kwargs):
        super().__init__(**kwargs)
        self.fixed_step_methods = {
            "euler": self.euler,
            "heun": self.heun,
            "rk4": self.rk4,
        }
        allowed_methods = ["euler", "heun", "rk4", "dopri5"]
        if method not in allowed_methods:
            raise ValueError(
```

```
                "Unknown␣ODE␣solver␣'{}',␣expected␣one␣of␣'{}'"
                    .format(
                     method, allowed_methods
                )
            )
        if method in self.fixed_step_methods.keys() and
            num_unfolds is None:
             raise ValueError(
                 "Fixed-step␣ODE␣solver␣requires␣argument␣'
                    num_unfolds'␣to␣be␣specified!"
             )
        self.units = units
        self.state_size_value = units
        self.num_unfolds = num_unfolds
        self.method = method
        self.tau = tau
        self.kernel, self.recurrent_kernel, self.bias, self.
            scale, self.solver = (None,) * 5

    def build(self, input_shape):
        input_dim = input_shape[-1]
        if isinstance(input_shape[0], tuple):
            input_dim = input_shape[0][-1]

        self.kernel = self.add_weight(
            shape=(input_dim, self.units), initializer="
                glorot_uniform", name="kernel"
        )
        self.recurrent_kernel = self.add_weight(
            shape=(self.units, self.units),
            initializer="orthogonal",
            name="recurrent_kernel",
        )
        self.bias = self.add_weight(
            shape=(self.units,), initializer=tf.keras.
                initializers.Zeros(), name="bias"
        )
        self.scale = self.add_weight(
            shape=(self.units,),
            initializer=tf.keras.initializers.Constant(1.0),
            name="scale",
        )
        if self.method == "dopri5":
```

```
            self.solver = tfp.math.ode.DormandPrince(
                rtol=0.01,
                atol=1e-04,
                first_step_size=0.01,
                safety_factor=0.8,
                min_step_size_factor=0.1,
                max_step_size_factor=10.0,
                max_num_steps=None,
                make_adjoint_solver_fn=None,
                validate_args=False,
                name="dormand_prince",
            )
        self.built = True

    def call(self, inputs, states):
        hidden_state = states[0]
        elapsed = 1.0
        if (isinstance(inputs, tuple) or isinstance(inputs,
           list)) and len(inputs) > 1:
            elapsed = inputs[1]
            inputs = inputs[0]

        if self.method == "dopri5":
            idx = None
            batch_dim = None
            if not type(elapsed) == float:
                batch_dim = tf.shape(elapsed)[0]
                elapsed = tf.reshape(elapsed, [batch_dim])

                idx = tf.argsort(elapsed)
                solution_times = tf.gather(elapsed, idx)
            else:
                solution_times = elapsed
            hidden_state = states[0]
            res = self.solver.solve(
                ode_fn=self.dfdt_wrapped,
                initial_time=0,
                initial_state=hidden_state,
                solution_times=solution_times,
                constants={"input": inputs},
            )
            if not type(elapsed) == float:
                i2 = tf.stack([idx, tf.range(batch_dim)], axis
```

```
                =1)
            hidden_state = tf.gather_nd(res.states, i2)
        else:
            hidden_state = res.states[-1]
    else:
        delta_t = elapsed / self.num_unfolds
        method = self.fixed_step_methods[self.method]
        for i in range(self.num_unfolds):
            hidden_state = method(inputs, hidden_state,
                delta_t)
    return hidden_state, [hidden_state]

def dfdt_wrapped(self, t, y, **constants):
    assert t is not None
    inputs = constants["input"]
    hidden_state = y
    return self.dfdt(inputs, hidden_state)

def dfdt(self, inputs, hidden_state):
    h_in = tf.matmul(inputs, self.kernel)
    h_rec = tf.matmul(hidden_state, self.recurrent_kernel)
    dh_in = self.scale * tf.nn.tanh(h_in + h_rec + self.
        bias)
    if self.tau > 0:
        dh = dh_in - hidden_state * self.tau
    else:
        dh = dh_in
    return dh

def euler(self, inputs, hidden_state, delta_t):
    dy = self.dfdt(inputs, hidden_state)
    return hidden_state + delta_t * dy

def heun(self, inputs, hidden_state, delta_t):
    k1 = self.dfdt(inputs, hidden_state)
    k2 = self.dfdt(inputs, hidden_state + delta_t * k1)
    return hidden_state + delta_t * 0.5 * (k1 + k2)

def rk4(self, inputs, hidden_state, delta_t):
    k1 = self.dfdt(inputs, hidden_state)
    k2 = self.dfdt(inputs, hidden_state + k1 * delta_t *
        0.5)
    k3 = self.dfdt(inputs, hidden_state + k2 * delta_t *
```

```
            0.5)
        k4 = self.dfdt(inputs, hidden_state + k3 * delta_t)
        return hidden_state + delta_t * (k1 + 2 * k2 + 2 * k3 +
            k4) / 6.0

    @property
    def state_size(self):
        return self.state_size_value

    @property
    def output_size(self):
        return self.state_size_value

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'units': self.units,
            'method': self.method,
            'num_unfolds': self.num_unfolds,
            'tau': self.tau
        })
        return config
```

Listing 6.9: ct_rnn.py

6.3.3 ct_gru.py

```
"""
code taken from https://github.com/mlech26l/ode-lstms/blob/
    master/node_cell.py (slightly modified)
"""

import numpy as np
import tensorflow as tf


@tf.keras.utils.register_keras_serializable()
class CTGRU(tf.keras.layers.AbstractRNNCell):
    def __init__(self, units, M=8, **kwargs):
        super(CTGRU, self).__init__(**kwargs)
        self.units = units
        self.M = M
        self.state_size_value = units * self.M
        self.ln_tau_table = np.empty(self.M)
        self.tau_table = np.empty(self.M)
```

```
        tau = 1.0
        for i in range(self.M):
            self.ln_tau_table[i] = np.log(tau)
            self.tau_table[i] = tau
            tau = tau * (10.0 ** 0.5)
        self.retrieval_layer, self.detect_layer, self.
            update_layer = (None,) * 3

    def build(self, input_shape):
        self.retrieval_layer = tf.keras.layers.Dense(
            self.units * self.M, activation=None
        )
        self.detect_layer = tf.keras.layers.Dense(self.units,
            activation="tanh")
        self.update_layer = tf.keras.layers.Dense(self.units *
            self.M, activation=None)
        self.built = True

    def call(self, inputs, states):
        elapsed = 1.0
        if (isinstance(inputs, tuple) or isinstance(inputs,
            list)) and len(inputs) > 1:
            elapsed = inputs[1]
            inputs = inputs[0]

        batch_dim = tf.shape(inputs)[0]

        h_hat = tf.reshape(states[0], [batch_dim, self.units,
            self.M])
        h = tf.reduce_sum(h_hat, axis=2)

        fused_input = tf.concat([inputs, h], axis=-1)
        ln_tau_r = self.retrieval_layer(fused_input)
        ln_tau_r = tf.reshape(ln_tau_r, shape=[batch_dim, self.
            units, self.M])
        sf_input_r = -tf.square(ln_tau_r - self.ln_tau_table)
        rki = tf.nn.softmax(logits=sf_input_r, axis=2)

        q_input = tf.reduce_sum(rki * h_hat, axis=2)
        reset_value = tf.concat([inputs, q_input], axis=1)
        qk = self.detect_layer(reset_value)
        qk = tf.reshape(qk, [batch_dim, self.units, 1])
```

```
        ln_tau_s = self.update_layer(fused_input)
        ln_tau_s = tf.reshape(ln_tau_s, shape=[batch_dim, self.
            units, self.M])
        sf_input_s = -tf.square(ln_tau_s - self.ln_tau_table)
        ski = tf.nn.softmax(logits=sf_input_s, axis=2)

        base_term = (1 - ski) * h_hat + ski * qk
        exp_term = tf.exp(-elapsed / self.tau_table)
        exp_term = tf.cast(tf.reshape(exp_term, [-1, 1, self.M
            ]), tf.float32)
        h_hat_next = base_term * exp_term

        h_next = tf.reduce_sum(h_hat_next, axis=2)
        h_hat_next_flat = tf.reshape(h_hat_next, shape=[
            batch_dim, self.units * self.M])
        return h_next, [h_hat_next_flat]

    @property
    def state_size(self):
        return self.state_size_value

    @property
    def output_size(self):
        return self.units

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'units': self.units,
            'M': self.M
        })
        return config
```

Listing 6.10: ct_gru.py

6.3.4 ode_lstm.py

```
"""
code taken from https://github.com/mlech26l/ode-lstms/blob/
    master/node_cell.py (slightly modified)
"""

import tensorflow as tf

import experiments.models.ct_rnn as ct_rnn
```

```
@tf.keras.utils.register_keras_serializable()
class ODELSTM(tf.keras.layers.AbstractRNNCell):
    def __init__(self, units, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.state_size_value = (units, units)
        self.initializer = "glorot_uniform"
        self.recurrent_initializer = "orthogonal"
        self.ctrnn = ct_rnn.CTRNNCell(self.units, num_unfolds
            =4, method="euler")
        self.input_kernel, self.recurrent_kernel, self.bias = (
            None,) * 3

    def get_initial_state(self, inputs=None, batch_size=None,
        dtype=None):
        return (
            tf.zeros([batch_size, self.units], dtype=tf.float32
                ),
            tf.zeros([batch_size, self.units], dtype=tf.float32
                ),
        )

    def build(self, input_shape):
        input_dim = input_shape[-1]
        if isinstance(input_shape[0], tuple):
            input_dim = input_shape[0][-1]
        self.ctrnn.build([self.units])
        self.input_kernel = self.add_weight(
            shape=(input_dim, 4 * self.units),
            initializer=self.initializer,
            name="input_kernel",
        )
        self.recurrent_kernel = self.add_weight(
            shape=(self.units, 4 * self.units),
            initializer=self.recurrent_initializer,
            name="recurrent_kernel",
        )
        self.bias = self.add_weight(
            shape=(4 * self.units),
            initializer=tf.keras.initializers.Zeros(),
            name="bias",
```

```
        )
        self.built = True

    def call(self, inputs, states):
        cell_state, ode_state = states
        elapsed = 1.0
        if (isinstance(inputs, tuple) or isinstance(inputs,
            list)) and len(inputs) > 1:
            elapsed = inputs[1]
            inputs = inputs[0]
        z = (
                tf.matmul(inputs, self.input_kernel)
                + tf.matmul(ode_state, self.recurrent_kernel)
                + self.bias
        )
        i, ig, fg, og = tf.split(z, 4, axis=-1)
        input_activation = tf.nn.tanh(i)
        input_gate = tf.nn.sigmoid(ig)
        forget_gate = tf.nn.sigmoid(fg + 3.0)
        output_gate = tf.nn.sigmoid(og)
        new_cell = cell_state * forget_gate + input_activation
            * input_gate
        ode_input = tf.nn.tanh(new_cell) * output_gate
        ode_output, new_ode_state = self.ctrnn.call([ode_input,
            elapsed], [ode_state])
        return ode_output, [new_cell, new_ode_state[0]]

    @property
    def state_size(self):
        return self.state_size_value

    @property
    def output_size(self):
        return self.state_size_value[0]

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'units': self.units
        })
        return config
```

Listing 6.11: ode_lstm.py

6.3.5 neural_circuit_policies.py

```
import kerasncp as ncp
import tensorflow as tf


@tf.keras.utils.register_keras_serializable()
class NeuralCircuitPolicies(tf.keras.layers.Layer):
    def __init__(self, units, output_size,
       inter_neuron_percentage=0.6, sensory_fanout=2,
       inter_fanout=2,
                 recurrent_command_synapses=None, motor_fanin
                    =2, return_sequences=False, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.inter_neuron_percentage = inter_neuron_percentage
        self.inter_neurons = int(0.6 * self.units)
        self.command_neurons = self.units - self.inter_neurons
        self.motor_neurons = output_size
        self.sensory_fanout = sensory_fanout
        self.inter_fanout = inter_fanout
        self.recurrent_command_synapses = 2 * self.
           command_neurons if recurrent_command_synapses is
           None else recurrent_command_synapses
        self.motor_fanin = motor_fanin
        self.return_sequences = return_sequences
        self.rnn = tf.keras.layers.RNN(
            ncp.LTCCell(
                ncp.wirings.NCP(self.inter_neurons, self.
                   command_neurons, self.motor_neurons, self.
                   sensory_fanout, self.inter_fanout, self.
                   recurrent_command_synapses, self.motor_fanin
                   )),
            return_sequences=self.return_sequences)

    def call(self, inputs, **kwargs):
        return self.rnn(inputs)

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'units': self.units,
            'output_size': self.motor_neurons,
            'inter_neuron_percentage': self.
```

```
            inter_neuron_percentage,
            'sensory_fanout': self.sensory_fanout,
            'inter_fanout': self.inter_fanout,
            'recurrent_command_synapses': self.
                recurrent_command_synapses,
            'motor_fanin': self.motor_fanin,
            'return_sequences': self.return_sequences
        })
        return config
```

Listing 6.12: neural_circuit_policies.py

6.3.6 unitary_rnn.py

```
"""
code taken from https://github.com/jingli9111/EUNN-tensorflow/
    blob/master/eunn.py (heavily modified)
"""

import math

import numpy as np
import tensorflow as tf

import experiments.models.model_factory as model_factory


def modrelu(inputs, bias, cplex=True):
    if cplex:
        norm = tf.abs(inputs) + 0.01
        biased_norm = norm + bias
        magnitude = tf.cast(tf.nn.relu(biased_norm), tf.
            complex64)
        phase = inputs / tf.cast(norm, tf.complex64)
    else:
        norm = tf.abs(inputs) + 0.01
        biased_norm = norm + bias
        magnitude = tf.nn.relu(biased_norm)
        phase = tf.sign(inputs)
    return phase * magnitude


def generate_index_tunable(s, L):
    ind1 = list(range(s))
    ind2 = list(range(s))
```

```
        for i in range(s):
            if i % 2 == 1:
                ind1[i] = ind1[i] - 1
                if i == s - 1:
                    continue
                else:
                    ind2[i] = ind2[i] + 1
            else:
                ind1[i] = ind1[i] + 1
                if i == 0:
                    continue
                else:
                    ind2[i] = ind2[i] - 1
        ind_exe = [ind1, ind2] * int(L / 2)
        ind3 = []
        ind4 = []
        for i in range(int(s / 2)):
            ind3.append(i)
            ind3.append(i + int(s / 2))
        ind4.append(0)
        for i in range(int(s / 2) - 1):
            ind4.append(i + 1)
            ind4.append(i + int(s / 2))
        ind4.append(s - 1)
        ind_param = [ind3, ind4]
        return ind_exe, ind_param


    def ind_s(k):
        if k == 0:
            return np.array([[1, 0]])
        else:
            temp = np.array(range(2 ** k))
            list0 = [np.append(temp + 2 ** k, temp)]
            list1 = ind_s(k - 1)
            for index in range(k):
                list0.append(np.append(list1[index], list1[index] +
                    2 ** k))
            return list0


    def generate_index_fft(s):
        t = ind_s(int(math.log(s / 2, 2)))
```

```
    ind_exe = []
    for i in range(int(math.log(s, 2))):
        ind_exe.append(tf.constant(t[i]))
    ind_param = []
    for i in range(int(math.log(s, 2))):
        ind = np.array([])
        for j in range(2 ** i):
            ind = np.append(ind, np.array(range(0, s, 2 ** i))
                + j).astype(np.int32)
        ind_param.append(tf.constant(ind))
    return ind_exe, ind_param


@tf.keras.utils.register_keras_serializable()
class EUNNCell(tf.keras.layers.AbstractRNNCell):
    def __init__(self,
                 num_units,
                 capacity=4,
                 fft=False,
                 cplex=True,
                 **kwargs):
        super().__init__(**kwargs)
        self._num_units = num_units
        self._activation = modrelu
        self._capacity = capacity
        self._fft = fft
        self._cplex = cplex
        if self._capacity > self._num_units:
            raise ValueError("Do not set capacity larger than
                hidden size, it is redundant")
        if self._fft:
            if math.log(self._num_units, 2) % 1 != 0:
                raise ValueError("FFT style only supports power
                    of 2 of hidden size")
        else:
            if self._num_units % 2 != 0:
                raise ValueError("Tunable style only supports
                    even number of hidden size")
            if self._capacity % 2 != 0:
                raise ValueError("Tunable style only supports
                    even number of capacity")
        self.phase_init = tf.random_uniform_initializer(-3.14,
            3.14)
```

```
        if self._fft:
            self._capacity = int(math.log(self._num_units, 2))
            self.theta, self.phi, self.omega = self.
                create_fft_weights()
        else:
            self.capacity_A, self.capacity_B, self.theta_A,
                self.theta_B, self.phi_A, self.phi_B, self.omega
                 = self.create_tunable_weights()
        self.bias = self.add_weight("bias", [self._num_units],
            initializer=tf.constant_initializer())
        if self._cplex:
            self.U_re, self.U_im = None, None
        else:
            self.U = None

    def build(self, input_shape):
        inputs_size = model_factory.get_concat_input_shape(
            input_shape)
        input_matrix_init = tf.random_uniform_initializer
            (-0.01, 0.01)
        if self._cplex:
            self.U_re = self.add_weight("U_re", [inputs_size,
                self._num_units], initializer=input_matrix_init)
            self.U_im = self.add_weight("U_im", [inputs_size,
                self._num_units], initializer=input_matrix_init)
        else:
            self.U = self.add_weight("U", [inputs_size, self.
                _num_units], initializer=input_matrix_init)

    @property
    def state_size(self):
        return self._num_units

    @property
    def output_size(self):
        return self._num_units

    def get_initial_state(self, inputs=None, batch_size=None,
        dtype=None):
        if self._cplex:
            dtype = tf.complex64
        else:
            dtype = tf.float32
```

```
        return tf.zeros((batch_size, self.state_size), dtype)

    def create_fft_weights(self):
        theta = self.add_weight("theta", [self._capacity, self.
            _num_units // 2], initializer=self.phase_init)
        if self._cplex:
            phi = self.add_weight("phi", [self._capacity, self.
                _num_units // 2], initializer=self.phase_init)
            omega = self.add_weight("omega", [self._num_units],
                initializer=self.phase_init)
        else:
            phi, omega = None, None
        return theta, phi, omega

    def create_fft_matrices(self):
        cos_theta = tf.cos(self.theta)
        sin_theta = tf.sin(self.theta)
        if self._cplex:
            cos_phi = tf.cos(self.phi)
            sin_phi = tf.sin(self.phi)
            cos_list_re = tf.concat([cos_theta, cos_theta *
                cos_phi], axis=1)
            cos_list_im = tf.concat([tf.zeros_like(self.theta),
                cos_theta * sin_phi], axis=1)
            sin_list_re = tf.concat([sin_theta, - sin_theta *
                cos_phi], axis=1)
            sin_list_im = tf.concat([tf.zeros_like(self.theta),
                - sin_theta * sin_phi], axis=1)
            cos_list = tf.complex(cos_list_re, cos_list_im)
            sin_list = tf.complex(sin_list_re, sin_list_im)
        else:
            cos_list = tf.concat([cos_theta, cos_theta], axis
                =1)
            sin_list = tf.concat([sin_theta, -sin_theta], axis
                =1)
        ind_exe, index_fft = generate_index_fft(self._num_units
            )
        v1 = tf.stack([tf.gather(cos_list[i, :], index_fft[i])
            for i in range(self._capacity)])
        v2 = tf.stack([tf.gather(sin_list[i, :], index_fft[i])
            for i in range(self._capacity)])
        if self._cplex:
            D = tf.complex(tf.cos(self.omega), tf.sin(self.
```

```
                omega))
        else:
            D = None
        diag = D
        return v1, v2, ind_exe, diag

    def create_tunable_weights(self):
        capacity_A = int(self._capacity // 2)
        capacity_B = self._capacity - capacity_A
        theta_A = self.add_weight("theta_A", [capacity_A, self.
            _num_units // 2], initializer=self.phase_init)
        theta_B = self.add_weight("theta_B", [capacity_B, self.
            _num_units // 2 - 1], initializer=self.phase_init)
        if self._cplex:
            phi_A = self.add_weight("phi_A", [capacity_A, self.
                _num_units // 2], initializer=self.phase_init)
            phi_B = self.add_weight("phi_B", [capacity_B, self.
                _num_units // 2 - 1], initializer=self.
                phase_init)
            omega = self.add_weight("omega", [self._num_units],
                 initializer=self.phase_init)
        else:
            phi_A, phi_B, omega = None, None, None
        return capacity_A, capacity_B, theta_A, theta_B, phi_A,
            phi_B, omega

    def create_tunable_matrices(self):
        cos_theta_A = tf.cos(self.theta_A)
        sin_theta_A = tf.sin(self.theta_A)
        if self._cplex:
            cos_phi_A = tf.cos(self.phi_A)
            sin_phi_A = tf.sin(self.phi_A)
            cos_list_A_re = tf.concat([cos_theta_A, cos_theta_A
                * cos_phi_A], axis=1)
            cos_list_A_im = tf.concat([tf.zeros_like(self.
                theta_A), cos_theta_A * sin_phi_A], axis=1)
            sin_list_A_re = tf.concat([sin_theta_A, -
                sin_theta_A * cos_phi_A], axis=1)
            sin_list_A_im = tf.concat([tf.zeros_like(self.
                theta_A), - sin_theta_A * sin_phi_A], axis=1)
            cos_list_A = tf.complex(cos_list_A_re,
                cos_list_A_im)
            sin_list_A = tf.complex(sin_list_A_re,
```

```
        sin_list_A_im)
else:
    cos_list_A = tf.concat([cos_theta_A, cos_theta_A],
        axis=1)
    sin_list_A = tf.concat([sin_theta_A, -sin_theta_A],
         axis=1)
cos_theta_B = tf.cos(self.theta_B)
sin_theta_B = tf.sin(self.theta_B)
if self._cplex:
    cos_phi_B = tf.cos(self.phi_B)
    sin_phi_B = tf.sin(self.phi_B)
    cos_list_B_re = tf.concat([tf.ones([self.capacity_B
        , 1]), cos_theta_B, cos_theta_B * cos_phi_B, tf.
        ones([self.capacity_B, 1])], axis=1)
    cos_list_B_im = tf.concat([tf.zeros([self.
        capacity_B, 1]), tf.zeros_like(self.theta_B),
        cos_theta_B * sin_phi_B, tf.zeros([self.
        capacity_B, 1])], axis=1)
    sin_list_B_re = tf.concat([tf.zeros([self.
        capacity_B, 1]), sin_theta_B, -sin_theta_B *
        cos_phi_B, tf.zeros([self.capacity_B, 1])], axis
        =1)
    sin_list_B_im = tf.concat([tf.zeros([self.
        capacity_B, 1]), tf.zeros_like(self.theta_B), -
        sin_theta_B * sin_phi_B, tf.zeros([self.
        capacity_B, 1])], axis=1)
    cos_list_B = tf.complex(cos_list_B_re,
        cos_list_B_im)
    sin_list_B = tf.complex(sin_list_B_re,
        sin_list_B_im)
else:
    cos_list_B = tf.concat([tf.ones([self.capacity_B,
        1]), cos_theta_B, cos_theta_B, tf.ones([self.
        capacity_B, 1])], axis=1)
    sin_list_B = tf.concat([tf.zeros([self.capacity_B,
        1]), sin_theta_B, -sin_theta_B, tf.zeros([self.
        capacity_B, 1])], axis=1)
ind_exe, [index_A, index_B] = generate_index_tunable(
   self._num_units, self._capacity)
diag_list_A = tf.gather(cos_list_A, index_A, axis=1)
off_list_A = tf.gather(sin_list_A, index_A, axis=1)
diag_list_B = tf.gather(cos_list_B, index_B, axis=1)
off_list_B = tf.gather(sin_list_B, index_B, axis=1)
```

```
        v1 = tf.reshape(tf.concat([diag_list_A, diag_list_B],
            axis=1), [self._capacity, self._num_units])
        v2 = tf.reshape(tf.concat([off_list_A, off_list_B],
            axis=1), [self._capacity, self._num_units])
        if self._cplex:
            D = tf.complex(tf.cos(self.omega), tf.sin(self.
                omega))
        else:
            D = None
        diag = D
        return v1, v2, ind_exe, diag

    def loop(self, h, v1, v2, ind, _diag):
        for i in range(self._capacity):
            diag = h * v1[i, :]
            off = h * v2[i, :]
            h = diag + tf.gather(off, ind[i], axis=1)
        if _diag is not None:
            h = h * _diag
        return h

    def call(self, inputs, state):
        inputs = model_factory.get_concat_inputs(inputs)
        if self._cplex:
            inputs_re = tf.matmul(inputs, self.U_re)
            inputs_im = tf.matmul(inputs, self.U_im)
            inputs = tf.complex(inputs_re, inputs_im)
        else:
            inputs = tf.matmul(inputs, self.U)
        if self._fft:
            v1, v2, ind, diag = self.create_fft_matrices()
        else:
            v1, v2, ind, diag = self.create_tunable_matrices()
        state = self.loop(state[0], v1, v2, ind, diag)
        output = self._activation((inputs + state), self.bias,
            self._cplex)
        return output, (output,)

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'num_units': self._num_units,
            'capacity': self._capacity,
```

```
            'fft': self._fft,
            'cplex': self._cplex
        })
        return config
```

Listing 6.13: unitary_rnn.py

6.3.7 matrix_exponential_unitary_rnn.py

```
import tensorflow as tf
import tensorflow_probability as tfp

import experiments.models.model_factory as model_factory
import experiments.models.unitary_rnn as urnn


def get_unitary_matrix(vector):
    triangular_matrix = tfp.math.fill_triangular(vector)
    skew_hermitian_matrix = triangular_matrix - tf.linalg.
        adjoint(triangular_matrix)
    unitary_matrix = tf.linalg.expm(skew_hermitian_matrix)
    return unitary_matrix


@tf.keras.utils.register_keras_serializable()
class MatrixExponentialUnitaryRNN(tf.keras.layers.
    AbstractRNNCell):
    def __init__(self, state_size, output_size,
        capacity_measure=1, use_fft=False,
        trainable_initial_state=False, **kwargs):
        super().__init__(**kwargs)
        self.state_size_value = state_size
        self.output_size_value = output_size
        assert 0 <= capacity_measure <= 1
        self.full_capacity = self.state_size * (self.state_size
            + 1) // 2
        self.capacity = int(capacity_measure * self.
            full_capacity)
        self.remaining_capacity = self.full_capacity - self.
            capacity
        self.use_fft = use_fft
        self.trainable_initial_state = trainable_initial_state
        self.real_state_vector = self.add_weight('
            real_state_vector', (self.capacity,), tf.float32, tf
            .keras.initializers.Constant())
```

```
        self.imag_state_vector = self.add_weight('
            imag_state_vector', (self.capacity,), tf.float32, tf
            .keras.initializers.Constant())
        self.real_initial_state = self.add_weight('
            real_initial_state', (self.state_size,), tf.float32,
             tf.keras.initializers.Constant(), trainable=self.
            trainable_initial_state)
        self.imag_initial_state = self.add_weight('
            imag_initial_state', (self.state_size,), tf.float32,
             tf.keras.initializers.Constant(), trainable=self.
            trainable_initial_state)
        self.bias = self.add_weight('bias', (self.state_size,),
             tf.float32, tf.keras.initializers.Constant())
        self.output_layer = tf.keras.layers.Dense(self.
            output_size)
        self.real_input_matrix = None
        self.imag_input_matrix = None

    @property
    def state_size(self):
        return self.state_size_value

    @property
    def output_size(self):
        return self.output_size_value

    def get_initial_state(self, inputs=None, batch_size=None,
        dtype=None):
        return tf.repeat(tf.complex(self.real_initial_state,
            self.imag_initial_state)[tf.newaxis, ...],
            batch_size, 0)

    def build(self, input_shape):
        inputs_size = model_factory.get_concat_input_shape(
            input_shape)
        factor = 2 if self.use_fft else 1
        self.real_input_matrix = self.add_weight('
            real_input_matrix', (self.state_size, factor *
            inputs_size), tf.float32, tf.keras.initializers.
            GlorotUniform())
        self.imag_input_matrix = self.add_weight('
            imag_input_matrix', (self.state_size, factor *
            inputs_size), tf.float32, tf.keras.initializers.
```

```
            GlorotUniform())

    def call(self, inputs, states):
        inputs = model_factory.get_concat_inputs(inputs)
        state_matrix = get_unitary_matrix(tf.concat((tf.complex
            (self.real_state_vector, self.imag_state_vector), tf
            .zeros((self.remaining_capacity,), tf.complex64)),
            -1))
        input_matrix = tf.complex(self.real_input_matrix, self.
            imag_input_matrix)
        time_domain_inputs = tf.cast(inputs, tf.complex64)
        if self.use_fft:
            frequency_domain_inputs = tf.signal.fft(
                time_domain_inputs)
            augmented_inputs = tf.concat((time_domain_inputs,
                frequency_domain_inputs), -1)
        else:
            augmented_inputs = time_domain_inputs
        input_parts = tf.matmul(input_matrix, augmented_inputs
            [..., tf.newaxis])
        state_parts = tf.matmul(state_matrix, states[0][..., tf
            .newaxis])
        next_states = tf.squeeze(urnn.modrelu(state_parts +
            input_parts, self.bias[..., tf.newaxis]), -1)
        outputs = self.output_layer(tf.concat((tf.math.real(
            next_states), tf.math.imag(next_states)), -1))
        return outputs, (next_states,)

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'state_size': self.state_size,
            'output_size': self.output_size,
            'use_fft': self.use_fft,
            'trainable_initial_state': self.
                trainable_initial_state
        })
        return config
```

Listing 6.14: matrix_exponential_unitary_rnn.py

6.3.8 unitary_ncp.py

```
import tensorflow as tf
```

```
import experiments.models.neural_circuit_policies as ncp
import experiments.models.unitary_rnn as urnn


@tf.keras.utils.register_keras_serializable()
class UnitaryNCP(tf.keras.layers.Layer):
    def __init__(self, units_urnn, units_ncp, output_size,
       return_sequences=False, **kwargs):
        super().__init__(**kwargs)
        self.units_urnn = units_urnn
        self.units_ncp = units_ncp
        self.output_size = output_size
        self.return_sequences = return_sequences
        self.urnn = tf.keras.layers.RNN(urnn.EUNNCell(
           units_urnn), return_sequences=True)
        self.ncp = ncp.NeuralCircuitPolicies(units_ncp, self.
           output_size, recurrent_command_synapses=0,
           return_sequences=self.return_sequences)

    def call(self, inputs, **kwargs):
        urnn_output = self.urnn(inputs)
        ncp_output = self.ncp(tf.math.real(urnn_output))
        return ncp_output

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'units_urnn': self.units_urnn,
            'units_ncp': self.units_ncp,
            'output_size': self.output_size,
            'return_sequences': self.return_sequences,
        })
        return config
```

Listing 6.15: unitary_ncp.py

6.3.9 transformer.py

```
import tensorflow as tf

import experiments.models.recurrent_network_attention as rna
import experiments.models.
   recurrent_network_augmented_transformer as rnat
```

```
def compute_padding_mask(signals):
    # mask the input away if all vector entries are zero
    padding_mask = tf.reduce_max(tf.cast(signals != 0, dtype=tf
        .float32), axis=2)
    # adjust dimension to enable batch operation during dot
        product attention
    return padding_mask[:, tf.newaxis, :]


def compute_look_ahead_mask(signals):
    # make sure that no attention to future positions is
        possible
    look_ahead_mask = tf.linalg.band_part(tf.ones((signals.
        shape[1], signals.shape[1])), -1, 0)
    # adjust dimension to enable batch operation during dot
        product attention
    return look_ahead_mask[tf.newaxis, ...]


def positional_encoding(positions, d_model):
    # compute factors for all dimensions
    positional_encoding_factors = 1 / tf.pow(1E4, tf.cast(2 * (
        tf.range(d_model) // 2) / d_model, dtype=tf.float32))
    # multiply each factor with the corresponding position to
        get the argument for the trigonometric functions
    positional_encoding_matrix = tf.cast(positions, dtype=tf.
        float32) * positional_encoding_factors
    # apply a sine to the even dimensions
    pem_sine = tf.sin(positional_encoding_matrix[:, :, 0::2])
    # apply a cosine to the odd dimensions
    pem_cosine = tf.cos(positional_encoding_matrix[:, :, 1::2])
    # cast the result and return a tensor
    return tf.reshape(tf.concat([pem_sine[..., tf.newaxis],
        pem_cosine[..., tf.newaxis]], axis=-1), (-1,
        positional_encoding_matrix.shape[1],
        positional_encoding_matrix.shape[2]))


def feed_forward_network(d_model, d_ff):
    # return the feed forward network structure used in the
        transformer layers
    return tf.keras.Sequential([
        # a dense layer with relu activation
```

```
            tf.keras.layers.Dense(d_ff, activation='relu'),
            # a dense layer without an activation function
            tf.keras.layers.Dense(d_model)
        ])


    def split_heads(qkv, num_heads, d_qkv):
        # split queries, key or values into num_heads - permutation
            necessary to compute right dot product
        return tf.transpose(tf.reshape(qkv, (-1, qkv.shape[1],
           num_heads, d_qkv)), perm=[0, 2, 1, 3])


    class MultiHeadAttention(tf.keras.layers.Layer):
        def __init__(self, d_model, num_heads):
            super().__init__()
            # parameters
            self.d_model = d_model
            self.num_heads = num_heads
            # used layers
            self.query_generator_network = tf.keras.layers.Dense(
               self.num_heads * self.d_model)
            self.key_generator_network = tf.keras.layers.Dense(self
               .num_heads * self.d_model)
            self.value_generator_network = tf.keras.layers.Dense(
               self.num_heads * self.d_model)
            self.mha_output_generator_network = tf.keras.layers.
               Dense(self.d_model)

        def call(self, inputs, **kwargs):
            # split inputs tuple to the arguments
            query_gen_input, key_gen_input, value_gen_input, mask =
                inputs
            # bring mask to format where 1 denotes no attention and
                insert head dimension
            if mask is not None:
                mask = tf.expand_dims(1 - mask, 1)
            # generate queries, keys and values
            queries = self.query_generator_network(query_gen_input)
            keys = self.key_generator_network(key_gen_input)
            values = self.value_generator_network(value_gen_input)
            # split queries, keys and values to the right amount of
                heads
```

```
        queries_heads = split_heads(queries, self.num_heads,
            self.d_model)
        keys_heads = split_heads(keys, self.num_heads, self.
            d_model)
        values_heads = split_heads(values, self.num_heads, self
            .d_model)
        # compute the attention logits from each query to each
            key
        attention_logits = tf.matmul(queries_heads, keys_heads,
             transpose_b=True)
        # scale the attention logits
        scaled_attention_logits = attention_logits / tf.math.
            sqrt(tf.cast(self.d_model, dtype=tf.float32))
        # set attention logits to very small value for input
            positions in mask (if present)
        if mask is not None:
             scaled_attention_logits -= tf.where(mask == 1, tf.
                 ones_like(mask) * float('inf'), mask)
        # compute the attention weight to each value per query
        attention_weights = tf.nn.softmax(
            scaled_attention_logits)
        # compute dot product attention
        dpa = tf.matmul(attention_weights, values_heads)
        # transpose dpa matrix such that the heads dimension is
             behind input dimension
        reshaped_dpa = tf.transpose(dpa, perm=[0, 2, 1, 3])
        # merge heads to single value dimension
        concatenated_dpa = tf.reshape(reshaped_dpa, (-1,
            reshaped_dpa.shape[1], self.num_heads * self.d_model
            ))
        # transform concatenated dpa to vectors of size d_model
        return self.mha_output_generator_network(
            concatenated_dpa), attention_weights


class EncoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, d_ff, dropout_rate,
        attention):
        super().__init__()
        # parameters
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_ff = d_ff
```

```
        self.dropout_rate = dropout_rate
        self.attention = attention
        # used layers
        self.att = self.attention(self.d_model, self.num_heads)
        self.ffn = feed_forward_network(self.d_model, self.d_ff
            )
        self.att_layer_norm = tf.keras.layers.
            LayerNormalization(epsilon=1e-6)
        self.ffn_layer_norm = tf.keras.layers.
            LayerNormalization(epsilon=1e-6)
        self.att_dropout = tf.keras.layers.Dropout(self.
            dropout_rate)
        self.ffn_dropout = tf.keras.layers.Dropout(self.
            dropout_rate)

    def call(self, inputs, **kwargs):
        # split inputs tuple to the arguments
        signals, encoder_zero_input_mask = inputs
        # compute self attention output values
        att_output, attention_weights = self.att((signals,
            signals, signals, encoder_zero_input_mask))
        # use a dropout layer to prevent overfitting
        att_output = self.att_dropout(att_output)
        # normalize att output with residual connection
        att_layer_norm_output = self.att_layer_norm(signals +
            att_output)
        # compute feed forward network output values
        ffn_output = self.ffn(att_layer_norm_output)
        # use a dropout layer to prevent overfitting
        ffn_output = self.ffn_dropout(ffn_output)
        # normalize ffn output with residual connection
        ffn_layer_norm_output = self.ffn_layer_norm(
            att_layer_norm_output + ffn_output)
        # the output of the second normalization layer is the
            output of the encoder layer
        return ffn_layer_norm_output


class Encoder(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, d_ff, num_layers,
        mask_zero_inputs, dropout_rate, attention):
        super().__init__()
        # parameters
```

```
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_ff = d_ff
        self.num_layers = num_layers
        self.mask_zero_inputs = mask_zero_inputs
        self.dropout_rate = dropout_rate
        self.attention = attention
        # used layers
        self.embedding = tf.keras.layers.Dense(self.d_model)
        self.encoder_layers = [EncoderLayer(self.d_model, self.
            num_heads, self.d_ff, self.dropout_rate, self.
            attention) for _ in range(self.num_layers)]
        self.dropout_layer = tf.keras.layers.Dropout(self.
            dropout_rate)

    def call(self, inputs, **kwargs):
        # this function computes the output of the encoder
        if isinstance(inputs, tuple):
            encoder_input, times = inputs
            positional_encoding_matrix = positional_encoding(tf
                .cumsum(times, axis=1), self.d_model)
        else:
            encoder_input = inputs
            positional_encoding_matrix = positional_encoding(tf
                .range(encoder_input.shape[1])[tf.newaxis, :, tf
                .newaxis], self.d_model)
        # compute mask to not attend to zero inputs if enabled
        if self.mask_zero_inputs:
            encoder_zero_input_mask = compute_padding_mask(
                encoder_input)
        else:
            encoder_zero_input_mask = None
        # embed the signal vectors into vectors of size d_model
        embedded_signals = self.embedding(encoder_input)
        # scale with with factor
        embedded_signals *= tf.math.sqrt(tf.cast(self.d_model,
            dtype=tf.float32))
        # add positional information to the embedded signals
            using times
        positional_embedded_signals = embedded_signals +
            positional_encoding_matrix
        # use a dropout layer to prevent overfitting
        positional_embedded_signals = self.dropout_layer(
```

```
            positional_embedded_signals)
        # create variable that is updated by each encoder layer
        encoder_layer_inout = positional_embedded_signals
        for i in range(self.num_layers):
            # compute output of each encoder layer
            encoder_layer_inout = self.encoder_layers[i]((
                encoder_layer_inout, encoder_zero_input_mask))
        # the output of the last encoder layer is the output of
            the encoder
        return encoder_layer_inout, encoder_zero_input_mask


class DecoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, d_ff, dropout_rate,
        attention):
        super().__init__()
        # parameters
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_ff = d_ff
        self.dropout_rate = dropout_rate
        self.attention = attention
        # used layers
        self.self_att = self.attention(self.d_model, self.
            num_heads)
        self.enc_dec_att = self.attention(self.d_model, self.
            num_heads)
        self.ffn = feed_forward_network(self.d_model, self.d_ff
            )
        self.self_att_layer_norm = tf.keras.layers.
            LayerNormalization(epsilon=1e-6)
        self.enc_dec_att_layer_norm = tf.keras.layers.
            LayerNormalization(epsilon=1e-6)
        self.ffn_layer_norm = tf.keras.layers.
            LayerNormalization(epsilon=1e-6)
        self.self_att_dropout = tf.keras.layers.Dropout(self.
            dropout_rate)
        self.enc_dec_att_dropout = tf.keras.layers.Dropout(self
            .dropout_rate)
        self.ffn_dropout = tf.keras.layers.Dropout(self.
            dropout_rate)

    def call(self, inputs, **kwargs):
```

```
        # split inputs tuple to the arguments
        signals, encoder_output, decoder_zero_input_mask,
            look_ahead_mask = inputs
        # compute self attention output values
        self_att_output, attention_weights = self.self_att((
            signals, signals, signals, look_ahead_mask))
        # use a dropout layer to prevent overfitting
        self_att_output = self.self_att_dropout(self_att_output
            )
        # normalize self att output with residual connection
        self_att_layer_norm_output = self.self_att_layer_norm(
            signals + self_att_output)
        # compute encoder decoder att output values
        enc_dec_att_output, attention_weights = self.
            enc_dec_att((self_att_layer_norm_output,
            encoder_output, encoder_output,
            decoder_zero_input_mask))
        # use a dropout layer to prevent overfitting
        enc_dec_att_output = self.enc_dec_att_dropout(
            enc_dec_att_output)
        # normalize encoder decoder att output with residual
            connection
        enc_dec_att_layer_norm_output = self.
            enc_dec_att_layer_norm(self_att_layer_norm_output +
            enc_dec_att_output)
        # compute feed forward network output values
        ffn_output = self.ffn(enc_dec_att_layer_norm_output)
        # use a dropout layer to prevent overfitting
        ffn_output = self.ffn_dropout(ffn_output)
        # normalize ffn output with residual connection
        ffn_layer_norm_output = self.ffn_layer_norm(
            enc_dec_att_layer_norm_output + ffn_output)
        # the output of the third normalization layer is the
            output of the decoder layer
        return ffn_layer_norm_output


class Decoder(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, d_ff, num_layers,
        token_amount, token_size, mask_zero_inputs, dropout_rate
        , attention):
        super().__init__()
        # parameters
```

```
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_ff = d_ff
        self.num_layers = num_layers
        self.token_amount = token_amount
        self.token_size = token_size
        self.mask_zero_inputs = mask_zero_inputs
        self.dropout_rate = dropout_rate
        self.attention = attention
        # used layers
        self.embedding = tf.keras.layers.Dense(self.d_model)
        self.token_output_layer = tf.keras.layers.Dense(self.
            token_size)
        self.decoder_layers = [DecoderLayer(self.d_model, self.
            num_heads, self.d_ff, self.dropout_rate, self.
            attention) for _ in range(self.num_layers)]
        self.dropout_layer = tf.keras.layers.Dropout(self.
            dropout_rate)

    def call(self, inputs, **kwargs):
        # split inputs tuple to the arguments
        encoder_output, decoder_zero_input_mask = inputs
        # create a start token
        tokens = tf.repeat(tf.ones_like(encoder_output)[:, :1,
            :1], self.token_size, axis=-1)
        # create the right amount of tokens
        for _ in range(self.token_amount):
            # create a look ahead mask such that tokens can
                only attend to previous positions
            look_ahead_mask = compute_look_ahead_mask(tokens)
            # embed the current tokens
            embedded_tokens = self.embedding(tokens)
            # scale with with factor
            embedded_tokens *= tf.math.sqrt(tf.cast(self.
                d_model, dtype=tf.float32))
            # build the position matrix
            positions = tf.range(embedded_tokens.shape[1])[tf.
                newaxis, :, tf.newaxis]
            # add positional information to the embedded tokens
            positional_embedded_tokens = embedded_tokens +
                positional_encoding(positions, self.d_model)
            # use a dropout layer to prevent overfitting
            positional_embedded_tokens = self.dropout_layer(
```

```
                positional_embedded_tokens)
            # create variable that is updated by each decoder
                layer
            decoder_layer_inout = positional_embedded_tokens
            for i in range(self.num_layers):
                # compute output of each decoder layer
                decoder_layer_inout = self.decoder_layers[i]((
                    decoder_layer_inout, encoder_output,
                    decoder_zero_input_mask, look_ahead_mask))
            # the output of the last decoder layer must be fed
                to the output dense layer
            # only the output token corresponding to the last
                input token is used
            next_token = self.token_output_layer(
                decoder_layer_inout[:, -1:, :])
            # add the new token to the token matrix
            tokens = tf.concat([tokens, next_token], axis=1)
        # return all produced tokens except the start token
        return tokens[:, 1:, :]


@tf.keras.utils.register_keras_serializable()
class Transformer(tf.keras.layers.Layer):
    def __init__(self, token_amount, token_size, d_model,
        num_heads, d_ff, num_layers, dropout_rate, attention,
        flatten_output=True, mask_zero_inputs=False, **kwargs):
        super().__init__(**kwargs)
        # parameters
        self.token_amount = token_amount
        self.token_size = token_size
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_ff = d_ff
        self.num_layers = num_layers
        self.flatten_output = flatten_output
        self.mask_zero_inputs = mask_zero_inputs
        self.dropout_rate = dropout_rate
        self.attention_type = attention
        if self.attention_type == 'mha':
            self.attention = MultiHeadAttention
        elif self.attention_type == 'rna':
            self.attention = rna.RecurrentNetworkAttention
        elif self.attention_type == 'rnat':
```

```
            self.attention = rnat.MultiHeadRecurrentAttention
        else:
            raise NotImplementedError
        # used layers
        self.encoder = Encoder(self.d_model, self.num_heads,
            self.d_ff, self.num_layers, self.mask_zero_inputs,
            self.dropout_rate, self.attention)
        self.decoder = Decoder(self.d_model, self.num_heads,
            self.d_ff, self.num_layers, self.token_amount, self.
            token_size, self.mask_zero_inputs, self.dropout_rate
            , self.attention)
        self.flatten = tf.keras.layers.Flatten()

    def call(self, inputs, **kwargs):
        # build the encoder output
        encoder_output = self.encoder(inputs)
        # build the decoder output
        decoder_output = self.decoder(encoder_output)
        # the output of the transformer is the (flattened)
            output of the decoder
        if self.flatten_output:
            return self.flatten(decoder_output)
        else:
            return decoder_output

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'token_amount': self.token_amount,
            'token_size': self.token_size,
            'd_model': self.d_model,
            'num_heads': self.num_heads,
            'd_ff': self.d_ff,
            'num_layers': self.num_layers,
            'dropout_rate': self.dropout_rate,
            'attention': self.attention_type,
            'flatten_output': self.flatten_output,
            'mask_zero_inputs': self.mask_zero_inputs
        })
        return config
```

Listing 6.16: transformer.py

6.3.10 recurrent_network_augmented_transformer.py

```
import tensorflow as tf

import experiments.models.transformer as transformer


def recurrent_dot_product_attention(queries, keys, values,
    d_qkv, recurrent_network_layers, mask):
    # compute the attention logits from each query to each key
    attention_logits = tf.matmul(queries, keys, transpose_b=
        True)
    # scale the attention logits
    scaled_attention_logits = attention_logits / tf.math.sqrt(
        tf.cast(d_qkv, dtype=tf.float32))
    # set attention logits to very small value for input
        positions in mask (if present)
    if mask is not None:
        scaled_attention_logits -= tf.where(mask == 1, tf.
            ones_like(mask) * float('inf'), mask)
    # compute the attention weight to each value per query
    attention_weights = tf.nn.softmax(scaled_attention_logits)
    # duplicate all value vectors for each input and weight
        them accordingly
    weighted_value_vectors = tf.expand_dims(attention_weights,
        axis=-1) * tf.repeat(tf.expand_dims(values, axis=2),
        values.shape[2], axis=2)
    shape = (-1, weighted_value_vectors.shape[3],
        weighted_value_vectors.shape[4])
    # this variable holds the concatenated rnn output at end
    concatenated_rnn_output = tf.ones_like(values)[:, :0, :, :]
    for head_index, layer in enumerate(recurrent_network_layers
        ):
        # aggregate all weighted value vectors for each input
            via an rnn for each head instead of a simple
            summation
        rnn_input = tf.reshape(weighted_value_vectors[:,
            head_index, :, :, :], shape)
        rnn_output = tf.reshape(recurrent_network_layers[
            head_index](rnn_input), shape)
        # concatenate the real part of the output for this head
            to the running variable
        concatenated_rnn_output = tf.concat([
            concatenated_rnn_output, tf.expand_dims(rnn_output,
            axis=1)], axis=1)
```

```
        # return the concatenated result of all heads
        return concatenated_rnn_output, attention_weights


    class MultiHeadRecurrentAttention(tf.keras.layers.Layer):
        def __init__(self, d_model, num_heads):
            super().__init__()
            # parameters
            self.d_model = d_model
            self.num_heads = num_heads
            # used layers
            self.query_generator_network = tf.keras.layers.Dense(
                self.num_heads * self.d_model)
            self.key_generator_network = tf.keras.layers.Dense(self
                .num_heads * self.d_model)
            self.value_generator_network = tf.keras.layers.Dense(
                self.num_heads * self.d_model)
            self.mhra_output_generator_network = tf.keras.layers.
                Dense(self.d_model)
            self.recurrent_network_layers = [tf.keras.layers.LSTM(
                self.d_model) for _ in range(self.num_heads)]

        def call(self, inputs, **kwargs):
            # split inputs tuple to the arguments
            query_gen_input, key_gen_input, value_gen_input, mask =
                inputs
            # bring mask to format where 1 denotes no attention and
                insert head dimension
            if mask is not None:
                mask = tf.expand_dims(1 - mask, 1)
            # generate queries, keys and values
            queries = self.query_generator_network(query_gen_input)
            keys = self.key_generator_network(key_gen_input)
            values = self.value_generator_network(value_gen_input)
            # split queries, keys and values to the right amount of
                heads
            queries_heads = transformer.split_heads(queries, self.
                num_heads, self.d_model)
            keys_heads = transformer.split_heads(keys, self.
                num_heads, self.d_model)
            values_heads = transformer.split_heads(values, self.
                num_heads, self.d_model)
            # compute the recurrent dot product attention
```

```
        rdpa, attention_weights =
            recurrent_dot_product_attention(queries_heads,
            keys_heads, values_heads, self.d_model, self.
            recurrent_network_layers, mask)
        # transpose rdpa matrix such that the heads dimension
            is behind input dimension
        reshaped_rdpa = tf.transpose(rdpa, perm=[0, 2, 1, 3])
        # merge heads to single value dimension
        concatenated_rdpa = tf.reshape(reshaped_rdpa, (-1,
            reshaped_rdpa.shape[1], self.num_heads * self.
            d_model))
        # transform concatenated dpa to vectors of size d_model
        return self.mhra_output_generator_network(
            concatenated_rdpa), attention_weights
```

Listing 6.17: recurrent_network_augmented_transformer.py

6.3.11 recurrent_network_attention.py

```
import tensorflow as tf

import experiments.models.unitary_rnn as urnn


class RecurrentNetworkAttention(tf.keras.layers.Layer):
    def __init__(self, dim, heads):
        super().__init__()
        # save the dimension and the heads of the transformer
        self.dim = dim
        self.heads = heads
        # create the rnn layers
        self.rnn_layers = [tf.keras.layers.RNN(urnn.EUNNCell(
            self.dim)) for _ in range(self.heads)]
        self.dense_layer = tf.keras.layers.Dense(self.dim)

    def call(self, inputs, **kwargs):
        # split inputs tuple to the arguments
        queries, _, values, _ = inputs
        # bring queries and values to the same shape
        duplicated_queries = tf.repeat(tf.expand_dims(queries,
            2), values.shape[1], 2)
        duplicated_values = tf.repeat(tf.expand_dims(values, 1)
            , queries.shape[1], 1)
        # concatenate queries and values together and reshape
            it to a single batch dimension
```

```
        memory_layer_input = tf.reshape(tf.concat([
            duplicated_queries, duplicated_values], -1), (-1,
            values.shape[1], 2 * self.dim))
        # accumulate information with memory layer
        accumulated_inputs = tf.concat([tf.math.real(rnn_layer(
            memory_layer_input)) for rnn_layer in self.
            rnn_layers], -1)
        # merge outputs of multiple heads to one single
            representation
        transformed_inputs = self.dense_layer(
            accumulated_inputs)
        # reshape the output to the right batch size and the
            right query dimension
        return tf.reshape(transformed_inputs, (-1, queries.
            shape[1], self.dim)), None
```

Listing 6.18: recurrent_network_attention.py

6.3.12 memory_augmented_transformer.py

```
import tensorflow as tf

import experiments.models.model_factory as model_factory
import experiments.models.transformer as transformer


@tf.keras.utils.register_keras_serializable()
class MemoryAugmentedTransformerCell(tf.keras.layers.
    AbstractRNNCell):
    def __init__(self, memory_rows=16, memory_columns=16,
        output_size=1,
                 embedding_size=32, heads=2, feed_forward_size
                     =128, dropout_rate=0, **kwargs):
        super().__init__(**kwargs)
        self.memory_rows = memory_rows
        self.memory_columns = memory_columns
        self.output_size_value = output_size
        self.output_layer = tf.keras.layers.Dense(output_size)
        self.memory_input_layer = tf.keras.layers.Dense(1 +
            self.memory_columns)
        self.embedding_size = embedding_size
        self.input_embedding = tf.keras.layers.Dense(self.
            embedding_size)
        self.memory_embedding = tf.keras.layers.Dense(self.
            embedding_size)
```

```
        self.heads = heads
        self.attention = transformer.MultiHeadAttention(self.
            embedding_size, self.heads)
        self.feed_forward_size = feed_forward_size
        self.feed_forward_layer = transformer.
            feed_forward_network(self.embedding_size, self.
            feed_forward_size)
        self.layer_normalization = tf.keras.layers.
            LayerNormalization(epsilon=1E-6)
        self.state_size_value = ((self.memory_rows, self.
            memory_columns),)
        self.dropout_rate = dropout_rate
        self.dropout_layer = tf.keras.layers.Dropout(self.
            dropout_rate)
        self.positional_encoding = transformer.
            positional_encoding(tf.range(1 + self.memory_rows)[
            tf.newaxis, ..., tf.newaxis], self.embedding_size)

    @property
    def state_size(self):
        return self.state_size_value

    @property
    def output_size(self):
        return self.output_size_value

    def get_initial_state(self, inputs=None, batch_size=None,
        dtype=None):
        return tf.fill((batch_size, self.memory_rows, self.
            memory_columns), 1E-6)

    def call(self, inputs, states):
        inputs = model_factory.get_concat_inputs(inputs)
        memory_state = states[0]
        embedded_memory_contents = self.memory_embedding(
            memory_state)
        embedded_inputs = self.input_embedding(tf.expand_dims(
            inputs, -2))
        augmented_inputs = tf.concat((embedded_inputs,
            embedded_memory_contents), -2) + self.
            positional_encoding
        augmented_inputs = self.dropout_layer(augmented_inputs)
        attention_output = self.dropout_layer(self.attention((
```

```
            augmented_inputs, augmented_inputs, augmented_inputs
                , None))[0]) + augmented_inputs
            normed_attention_output = self.layer_normalization(
                attention_output)
            feed_forward_output = self.dropout_layer(self.
                feed_forward_layer(normed_attention_output)) +
                normed_attention_output
            normed_feed_forward_output = self.layer_normalization(
                feed_forward_output)
            memory_layer_outputs = self.output_layer(
                normed_feed_forward_output[:, 0, :])
            memory_inputs = self.memory_input_layer(
                normed_feed_forward_output[:, 1:, :])
            control_signals = memory_inputs[..., :1]
            data_signals = memory_inputs[..., 1:]
            memory_state = tf.sigmoid(-control_signals) *
                memory_state + tf.sigmoid(control_signals) *
                data_signals
            return memory_layer_outputs, (memory_state,)

        def get_config(self):
            config = super().get_config().copy()
            config.update({
                'memory_rows': self.memory_rows,
                'memory_columns': self.memory_columns,
                'output_size': self.output_size,
                'embedding_size': self.embedding_size,
                'heads': self.heads,
                'feed_forward_size': self.feed_forward_size,
                'dropout_rate': self.dropout_rate
            })
            return config
```

Listing 6.19: memory_augmented_transformer.py

6.3.13 differentiable_neural_computer.py

```
"""
Differentiable Neural Computer model definition.
code taken from https://github.com/willsq/tf-DNC/tree/master/
    dnc (slightly modified)

Reference:
    http://www.nature.com/nature/journal/vaop/ncurrent/full/
        nature20101.html
```

```
Conventions:
    B - batch size
    N - number of slots in memory
    R - number of read heads
    W - size of each memory slot i.e word size
"""

import collections

import tensorflow as tf

import experiments.models.model_factory as model_factory

# -*- coding: utf-8 -*-
"""DNC memory operations and state.

Conventions:
    B - batch size
    N - number of slots in memory
    R - number of read heads
"""

EPSILON = 1e-6


class ContentAddressing:
    """
    Access memory content using cosine similarity.

        Used for: reading, writing
    """

    @staticmethod
    def weighting(memory_matrix, keys, strengths, sharpness_op=
        tf.math.softplus):
        """Get content-based weighting using cosine similarity.
            The weighting
        for each memory slot will be high if the key points in
            the same
        direction as the memory contents at that slot.

        Args:
```

```
            memory_matrix (Tensor [B, N, W]): the memory matrix
                to query
            keys (Tensor [B, W, R]): the keys to query the
               memory
            strengths (Tensor [B, R]): strengths for each
               lookup key
            sharpness_op (fn): operation to transform strengths
                before softmax


        Returns:
            Tensor [B, N, R]: lookup weightings for each key
        """
        memory_normalised = tf.math.l2_normalize(memory_matrix,
            2, epsilon=EPSILON)
        keys_normalised = tf.math.l2_normalize(keys, 1, epsilon
           =EPSILON)
        similarity = tf.matmul(memory_normalised,
           keys_normalised)
        strengths = tf.expand_dims(sharpness_op(strengths), 1)

        return tf.math.softmax(similarity * strengths, 1)


class TemporalLinkAddressing:
    """
    Access memory content by considering which interactions
       have happened
    recently in time.

        Used for: reading
    """

    @staticmethod
    def update_precedence_vector(prev_precedence_vector,
       write_weighting):
        """Return next precedence vector by taking into account
            the writing
        action that has just happened via `write_weighting`.

        The precedence vector at position `i` denotes the
           degree to which
        memory location `i` has been recently written.
```

```
        Args:
            prev_precedence_vector (Tensor [B, N]): precedence
                vector from time t-1
            write_weighting (Tensor [B, N]): final weighting
                used to write at time t

        Returns:
            Tensor [B, N]: precedence vector to use at next
                time step
        """
        write_strength = tf.reduce_sum(input_tensor=
            write_weighting, axis=1, keepdims=True)
        updated_precedence_vector = (1 - write_strength) *
            prev_precedence_vector + write_weighting

        return updated_precedence_vector

    @staticmethod
    def update_link_matrix(prev_link_matrix,
        prev_precedence_vector, write_weighting):
        """Adjust the link matrix by taking into account the
            writing
        action that has just happened and the previous
            precedence vector.

        Link matrix at `L[t,i,j]` describes the degree to which
            memory location
        `i` was written after location `j` between time `t` and
            `t+1`.

        Args:
            prev_link_matrix (Tensor [B, N, N]): link matrix
                from time t-1
            prev_precedence_vector (Tensor [B, N]): precedence
                vector from time t-1
            write_weighting (Tensor [B, N)): final weighting
                used to write at time t

        Returns:
            Tensor [B, N, N]: temporal link matrix to use at
                next time step
        """
```

```
        batch_size = prev_link_matrix.shape[0]
        if batch_size is None:
            return prev_link_matrix
        words_num = prev_link_matrix.shape[1]

        write_weighting_i = tf.expand_dims(write_weighting, 2)
            # [b x N x 1] duplicate columns
        write_weighting_j = tf.expand_dims(write_weighting, 1)
            # [b x 1 X N] duplicate rows
        prev_precedence_vector_j = tf.expand_dims(
            prev_precedence_vector, 1)  # [b x 1 X N]

        link_matrix = (
                (1 - write_weighting_i - write_weighting_j) *
                    prev_link_matrix
                + (write_weighting_i * prev_precedence_vector_j
                    )
        )
        zero_diagonal = tf.zeros([batch_size, words_num], dtype
            =link_matrix.dtype)

        return tf.linalg.set_diag(link_matrix, zero_diagonal)

    @staticmethod
    def weightings(link_matrix, prev_read_weightings):
        """Calculate weightings for each read head so they have
            a preference
        towards directionality.

        Args:
            link_matrix (Tensor [B, N, N])
            prev_read_weightings (Tensor [B, N, R]): read
                weightings from time t-1

        Returns:
            Tuple(Tensor [B, N, R], Tensor [B, N, R]): temporal
                weightings for each memory slot
        """
        forward_weighting = tf.matmul(link_matrix,
            prev_read_weightings)
        backward_weighting = tf.matmul(link_matrix,
            prev_read_weightings, adjoint_a=True)
```

```
        return forward_weighting, backward_weighting


class AllocationAddressing:
    """
    Access memory content by considering which memory slots can
        be allocated to.
    This is used to provide a differentiable form of dynamic
       memory allocation
    where slots can only be written to if they are determined
       to be free.

        Used for: writing
    """

    @staticmethod
    def update_usage_vector(free_gates, prev_read_weightings,
                            prev_write_weighting,
                               prev_usage_vector):
        """Adjust the usage vector based on reads and writes
           from previous time
        step.

        The usage vector is a helper data structure to aid in
           the calculation of
        the allocation weighting. `u[t,i]` describes the usage
           between [0,1]
        inside memory slot `i` at time `t`. Elements of usage
           vector may add up
        to a maximum of `N`.

        The free gate allows reads to happen over multiple time
            steps at the same
        location, otherwise we would always say a location is
           unused immediately
        after a read has occurred.

        Args:
            free_gates (Tensor [B, R]): current free gate
            prev_read_weightings (Tensor [B, N, R]): read
               weightings from time t-1
            prev_write_weighting (Tensor [B, N]): write
               weighting from time t-1
```

```
            prev_usage_vector (Tensor [B, N]): usage vector
                from time t-1

        Returns:
            Tensor [B, N]: new usage vector
        """
        with tf.name_scope('allocation_addressing'):
            retention_vector = tf.reduce_prod(
                input_tensor=1 - tf.expand_dims(free_gates, 1)
                    * prev_read_weightings,
                axis=2,
            )
            usage_vector = (
                    (prev_usage_vector + prev_write_weighting
                     - (prev_usage_vector *
                         prev_write_weighting))
                    * retention_vector
            )
            return usage_vector

    @staticmethod
    def batch_unsort(tensor, indices):
        """Permute each batch in a batch first tensor according
            to tensor
        of indices.
        """
        if indices.shape[0] is None:
            return tensor
        unpacked = tf.unstack(indices)
        indices_inverted = tf.stack(
            [tf.math.invert_permutation(permutation) for
                permutation in unpacked]
        )

        unpacked = zip(tf.unstack(tensor), tf.unstack(
            indices_inverted))
        return tf.stack([tf.gather(value, index) for value,
            index in unpacked])

    @staticmethod
    def weighting(usage_vector):
        """Calculate allocation weighting so we know which
            memory slots are
```

```
        free to be written to. Tells us the degree to which
           each memory location
        is "allocable".

        Args:
            usage_vector (Tensor [B, N]): newly calculated
               usage vector at time t

        Returns:
            Tensor [B, N]: allocation weighting for each memory
                slot
        """
        usage = (1 - EPSILON) * usage_vector + EPSILON
        emptiness = 1 - usage

        words_num = usage_vector.get_shape().as_list()[1]
        emptiness_sorted, free_list = tf.nn.top_k(emptiness, k=
           words_num)
        usage_sorted = 1 - emptiness_sorted
        allocation_sorted = emptiness_sorted * tf.math.cumprod(
           usage_sorted, axis=1, exclusive=True)

        return AllocationAddressing.batch_unsort(
           allocation_sorted, free_list)


class Memory:
    """Differentiable memory for the DNC.

    This module implements a recurrent module interface and
       tracks memory state
    through time. Performs a write and read operation given the
        previous state
    and an interface vector defining how to interact with the
       memory at the
    current time step.

    Note: although this layer behaves similar to an rnn, it has
        no parameters
    and is actually a deterministic operation:
        (interface, prev_memory_state) -> (read_vectors,
           new_memory_state)
```

```
    Args:
        words_num (int): number of memory slots
        word_size (int): size of each memory slot
        read_heads_num (int): number of read heads to use
            inside memory
    """

    state = collections.namedtuple(
        "memory_state", [
            'memory_matrix',
            'usage_vector',
            'link_matrix',
            'precedence_vector',
            'write_weighting',
            'read_weightings',
        ]
    )

    def __init__(self, words_num=256, word_size=64,
        read_heads_num=4):
        self._N = words_num
        self._W = word_size
        self._R = read_heads_num

    def __call__(self, interface, prev_memory_state):
        """Define op for the recurrent module.

        Args:
            interface (namedtuple): parsed interface vector
            prev_memory_state (namedtuple): object containing
                the memory plus all
                 the helper data structures used to interface
                    with the memory

        Returns:
            Tuple:
                read vectors (Tensor [N, R]): read vectors
                    taken out of the memory
                next memory state (namedtuple): new state after
                     write and read
        """
        with tf.name_scope("write"):
            usage, write_weighting, memory_matrix, link_matrix,
```

```
            precedence = Memory.write(
                prev_memory_state,
                interface,
            )

        with tf.name_scope("read"):
            read_weightings, read_vectors = Memory.read(
                memory_matrix,
                prev_memory_state.read_weightings,
                link_matrix,
                interface,
            )
        return read_vectors, Memory.state(
            memory_matrix=memory_matrix,
            usage_vector=usage,
            link_matrix=link_matrix,
            precedence_vector=precedence,
            write_weighting=write_weighting,
            read_weightings=read_weightings,
        )

    @staticmethod
    def read(memory_matrix, prev_read_weightings, link_matrix,
        interface):
        """Perform read on memory.

        Args:
            memory_matrix (Tensor [B, N, W]): memory matrix
                after recent write at time t
            prev_read_weightings (Tensor [B, N, R]): read
                weightings from time t-1
            link_matrix (Tensor [B, N, N]): link matrix after
                recent write at time t
            interface (namedtuple): parsed interface vector

        Returns:
            Tuple:
                read_weightings (Tensor [B, N, R]): read
                    vectors taken out of the memory
                read_vectors (Tensor [B, W, R]): read vectors
                    taken out of the memory

        """
```

```
        with tf.name_scope("content_addressing"):
            lookup_weighting = ContentAddressing.weighting(
                memory_matrix,
                interface.read_keys,
                interface.read_strengths
            )
        with tf.name_scope("temporal_link_addressing"):
            forward_weighting, backward_weighting =
               TemporalLinkAddressing.weightings(
                link_matrix,
                prev_read_weightings,
            )

        with tf.name_scope("blend_addressing_modes"):
            read_weightings = tf.einsum(
                "bsr,bnrs->bnr",
                interface.read_modes,
                tf.stack([backward_weighting, lookup_weighting,
                    forward_weighting], axis=3)
            )
        read_vectors = tf.matmul(memory_matrix, read_weightings
            , adjoint_a=True)

        return read_weightings, read_vectors

    @staticmethod
    def write(prev_memory_state, interface):
        """Perform write on memory.

        Args:
            prev_memory_state (namedtuple): memory state from
                time t-1
            interface (namedtuple): parsed interface vector

        Returns:
            Tuple:
                usage_vector (Tensor [B, N])
                write_weighting (Tensor [B, N])
                memory_matrix (Tensor [B, N, W])
                link_matrix (Tensor [B, N, N])
                precedence_vector (Tensor [B, N])
        """
        m = prev_memory_state
```

```
i = interface

with tf.name_scope("calculate_weighting"):
    with tf.name_scope("allocation_addressing"):
        usage_vector = AllocationAddressing.
            update_usage_vector(
             i.free_gates,
             m.read_weightings,
             m.write_weighting,
             m.usage_vector
        )
        allocation_weighting = AllocationAddressing.
            weighting(usage_vector)
    with tf.name_scope("content_addressing"):
        lookup_weighting = ContentAddressing.weighting(
             m.memory_matrix,
             i.write_key,
             i.write_strength
        )
    write_weighting = (
             i.write_gate * (i.allocation_gate *
                allocation_weighting +
                             (1 - i.allocation_gate) *
                                tf.squeeze(
                                lookup_weighting))
    )

with tf.name_scope("erase_and_write"):
    erase = m.memory_matrix * (
        (1 - tf.einsum("bn,bw->bnw", write_weighting, i
            .erase_vector)))
    write = tf.einsum("bn,bw->bnw", write_weighting, i.
        write_vector)
    memory_matrix = erase + write

with tf.name_scope("final_update"):
    link_matrix = TemporalLinkAddressing.
        update_link_matrix(
        m.link_matrix,
        m.precedence_vector,
        write_weighting
    )
    precedence_vector = TemporalLinkAddressing.
```

```
                    update_precedence_vector(
                     m.precedence_vector,
                     write_weighting
                )

            return usage_vector, write_weighting, memory_matrix, \
                    link_matrix, precedence_vector

        @property
        def state_size(self):
            return Memory.state(
                memory_matrix=tf.TensorShape([self._N, self._W]),
                usage_vector=tf.TensorShape([self._N]),
                link_matrix=tf.TensorShape([self._N, self._N]),
                precedence_vector=tf.TensorShape([self._N]),
                write_weighting=tf.TensorShape([self._N]),
                read_weightings=tf.TensorShape([self._N, self._R]),
            )

        def get_initial_state(self, batch_size, dtype=tf.float32):
            return Memory.state(
                memory_matrix=tf.fill([batch_size, self._N, self._W
                    ], EPSILON),
                usage_vector=tf.zeros([batch_size, self._N], dtype=
                    dtype),
                link_matrix=tf.zeros([batch_size, self._N, self._N
                    ], dtype=dtype),
                precedence_vector=tf.zeros([batch_size, self._N],
                    dtype=dtype),
                write_weighting=tf.fill([batch_size, self._N],
                    EPSILON),
                read_weightings=tf.fill([batch_size, self._N, self.
                    _R], EPSILON),
            )


    @tf.keras.utils.register_keras_serializable()
    class DNC(tf.keras.layers.AbstractRNNCell):
        """DNC recurrent module that connects together the
           controller and memory.

        Performs a write and read operation against memory given 1)
            the previous state
```

```
    and 2) an interface vector defining how to interact with
       the memory at the
    current time step.

    Args:
        output_size (int): size of final output dimension for
           the whole DNC cell at each time step
        controller_units (int): size of hidden state in
           controller
        memory_size (int): number of slots in external memory
        word_size (int): the width of each memory slot
        num_read_heads (int): number of memory read heads
    """

    state = collections.namedtuple("dnc_state", [
        "memory_state",
        "controller_state",
        "read_vectors",
    ])

    interface = collections.namedtuple("interface", [
        "read_keys",
        "read_strengths",
        "write_key",
        "write_strength",
        "erase_vector",
        "write_vector",
        "free_gates",
        "allocation_gate",
        "write_gate",
        "read_modes",
    ])

    def __init__(self, output_size, controller_units=256,
       memory_size=256,
                 word_size=64, num_read_heads=4, **kwargs):
        super().__init__(**kwargs)

        self._output_size = output_size
        self._N = memory_size
        self._R = num_read_heads
        self._W = word_size
        self._interface_vector_size = self._R * self._W + 3 *
```

```
            self._W + 5 * self._R + 3
        self._clip = 20.0

        self._controller = tf.keras.layers.LSTMCell(units=
            controller_units)
        self._controller_to_interface_dense = tf.keras.layers.
            Dense(
             self._interface_vector_size,
             name='controller_to_interface'
        )
        self._memory = Memory(memory_size, word_size,
            num_read_heads)
        self._final_output_dense = tf.keras.layers.Dense(self.
            _output_size)

    def _parse_interface_vector(self, interface_vector):
        r, w = self._R, self._W

        sizes = [r * w, r, w, 1, w, w, r, 1, 1, 3 * r]
        fns = collections.OrderedDict([
            ("read_keys", lambda v: tf.reshape(v, (-1, w, r))),
            ("read_strengths", lambda v: 1 + tf.nn.softplus((tf
                .reshape(v, (-1, r))))),
            ("write_key", lambda v: tf.reshape(v, (-1, w, 1))),
            ("write_strength", lambda v: 1 + tf.nn.softplus((tf
                .reshape(v, (-1, 1))))),
            ("erase_vector", lambda v: tf.nn.sigmoid(tf.reshape
                (v, (-1, w)))),
            ("write_vector", lambda v: tf.reshape(v, (-1, w))),
            ("free_gates", lambda v: tf.nn.sigmoid(tf.reshape(v
                , (-1, r)))),
            ("allocation_gate", lambda v: tf.nn.sigmoid(tf.
                reshape(v, (-1, 1)))),
            ("write_gate", lambda v: tf.nn.sigmoid(tf.reshape(v
                , (-1, 1)))),
            ("read_modes", lambda v: tf.nn.softmax(tf.reshape(v
                , (-1, 3, r)), axis=1)),
        ])
        indices = [[sum(sizes[:i]), sum(sizes[:i + 1])] for i
            in range(len(sizes))]
        zipped_items = zip(fns.keys(), fns.values(), indices)
        interface = {name: fn(interface_vector[:, i[0]:i[1]])
            for name, fn, i in zipped_items}
```

```
        return DNC.interface(**interface)

    def _flatten_read_vectors(self, x):
        return tf.reshape(x, (-1, self._W * self._R))

    def call(self, inputs, prev_dnc_state):
        inputs = model_factory.get_concat_inputs(inputs)
        prev_dnc_state = tf.nest.pack_sequence_as(self.
            state_size_nested, prev_dnc_state)
        with tf.name_scope("inputs_to_controller"):
            read_vectors_flat = self._flatten_read_vectors(
                prev_dnc_state.read_vectors)
            input_augmented = tf.concat([inputs,
                read_vectors_flat], 1)
            controller_output, controller_state = self.
                _controller(
                 input_augmented,
                 prev_dnc_state.controller_state,
            )
            controller_output = tf.clip_by_value(
                controller_output, -self._clip, self._clip)

        with tf.name_scope("parse_interface"):
            interface = self._controller_to_interface_dense(
                controller_output)
            interface = self._parse_interface_vector(interface)

        with tf.name_scope("update_memory"):
            read_vectors, memory_state = self._memory(interface
                , prev_dnc_state.memory_state)
            state = DNC.state(
                 memory_state=memory_state,
                 controller_state=controller_state,
                 read_vectors=read_vectors,
            )

        with tf.name_scope("join_outputs"):
            read_vectors_flat = self._flatten_read_vectors(
                read_vectors)
            final_output = tf.concat([controller_output,
                read_vectors_flat], 1)
            final_output = self._final_output_dense(
```

```
                final_output)
            final_output = tf.clip_by_value(final_output, -self
                ._clip, self._clip)

        return final_output, tf.nest.flatten(state)

    @property
    def state_size_nested(self):
        return DNC.state(
            memory_state=self._memory.state_size,
            controller_state=self._controller.state_size,
            read_vectors=tf.TensorShape([self._W, self._R]),
        )

    @property
    def state_size(self):
        return tf.nest.flatten(self.state_size_nested)

    def get_initial_state(self, inputs=None, batch_size=None,
       dtype=tf.float32):
        del inputs
        initial_state_nested = DNC.state(
            memory_state=self._memory.get_initial_state(
                batch_size, dtype=dtype),
            controller_state=self._controller.get_initial_state
                (batch_size=batch_size, dtype=dtype),
            read_vectors=tf.fill([batch_size, self._W, self._R
                ], EPSILON),
        )
        return tf.nest.flatten(initial_state_nested)

    @property
    def output_size(self):
        return self._output_size

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'output_size': self.output_size,
            'controller_units': self._controller.units,
            'memory_size': self._N,
            'word_size': self._W,
            'num_read_heads': self._R
```

```
        })
        return config
```

Listing 6.20: differentiable_neural_computer.py

6.3.14 memory_cell.py

```
import tensorflow as tf


@tf.keras.utils.register_keras_serializable()
class MemoryCell(tf.keras.layers.AbstractRNNCell):
    def __init__(self, discretization_steps=2, **kwargs):
        super().__init__(**kwargs)
        self.discretization_steps = discretization_steps
        self.params = {
            'step_size': self.add_weight(name='step_size',
                shape=(1,), initializer=tf.keras.initializers.
                Constant(1.5573331)),
            'capacitance': self.add_weight(name='capacitance',
                shape=(1,), initializer=tf.keras.initializers.
                Constant(1), trainable=False),
            'leakage_conductance': self.add_weight(name='
                leakage_conductance', shape=(1,), initializer=tf
                .keras.initializers.Constant(0.4505964)),
            'resting_potential': self.add_weight(name='
                resting_potential', shape=(1,), initializer=tf.
                keras.initializers.Constant(0), trainable=False)
                ,
            'recurrent_conductance': self.add_weight(name='
                recurrent_conductance', shape=(1,), initializer=
                tf.keras.initializers.Constant(1.0334609)),
            'recurrent_mean_conductance_potential': self.
                add_weight(name='
                recurrent_mean_conductance_potential', shape
                =(1,), initializer=tf.keras.initializers.
                Constant(0.07879465)),
            'recurrent_std_conductance_potential': self.
                add_weight(name='
                recurrent_std_conductance_potential', shape=(1,)
                , initializer=tf.keras.initializers.Constant
                (100), trainable=False),
            'recurrent_target_potential': self.add_weight(name=
                'recurrent_target_potential', shape=(1,),
```

```
                initializer=tf.keras.initializers.Constant
                (1.4378392)),
            'inhibitory_conductance': self.add_weight(name='
                inhibitory_conductance', shape=(1,), initializer
                =tf.keras.initializers.Constant(1.3365093)),
            'inhibitory_mean_conductance_potential': self.
                add_weight(name='
                inhibitory_mean_conductance_potential', shape
                =(1,), initializer=tf.keras.initializers.
                Constant(0.06618887)),
            'inhibitory_std_conductance_potential': self.
                add_weight(name='
                inhibitory_std_conductance_potential', shape
                =(1,), initializer=tf.keras.initializers.
                Constant(100), trainable=False),
            'inhibitory_target_potential': self.add_weight(name
                ='inhibitory_target_potential', shape=(1,),
                initializer=tf.keras.initializers.Constant(0),
                trainable=False),
            'input_conductance': self.add_weight(name='
                input_conductance', shape=(1,), initializer=tf.
                keras.initializers.Constant(0.07915332)),
            'input_mean_conductance_potential': self.add_weight
                (name='input_mean_conductance_potential', shape
                =(1,), initializer=tf.keras.initializers.
                Constant(0.5), trainable=False),
            'input_std_conductance_potential': self.add_weight(
                name='input_std_conductance_potential', shape
                =(1,), initializer=tf.keras.initializers.
                Constant(100), trainable=False),
            'input_target_potential': self.add_weight(name='
                input_target_potential', shape=(1,), initializer
                =tf.keras.initializers.Constant(1.5931877)),
        }

    @property
    def state_size(self):
        return 2

    @property
    def output_size(self):
        return 2
```

```
def get_initial_state(self, inputs=None, batch_size=None,
    dtype=None):
    return tf.concat((tf.zeros((batch_size, 1)), tf.ones((
        batch_size, 1))), -1)

def synaptic_current(self, synapse_type, presynaptic,
    postsynaptic):
    conductance = self.params[f'{synapse_type}_conductance'
        ] * \
                  tf.math.sigmoid(self.params[f'{
                      synapse_type}
                      _std_conductance_potential'] * (
                      presynaptic - self.params[f'{
                      synapse_type}
                      _mean_conductance_potential']))
    potential_difference = self.params[f'{synapse_type}
        _target_potential'] - postsynaptic
    return conductance * potential_difference

def leakage_current(self, potential):
    return self.params['leakage_conductance'] * (self.
        params['resting_potential'] - potential)

def state_derivative(self, neuron_x_inputs,
    neuron_x_potentials, neuron_y_inputs,
    neuron_y_potentials):
    current_x = self.synaptic_current('input',
        neuron_x_inputs, neuron_x_potentials)
    current_x += self.synaptic_current('inhibitory',
        neuron_y_potentials, neuron_x_potentials)
    current_x += self.synaptic_current('recurrent',
        neuron_x_potentials, neuron_x_potentials)
    current_x += self.leakage_current(neuron_x_potentials)
    current_y = self.synaptic_current('input',
        neuron_y_inputs, neuron_y_potentials)
    current_y += self.synaptic_current('inhibitory',
        neuron_x_potentials, neuron_y_potentials)
    current_y += self.synaptic_current('recurrent',
        neuron_y_potentials, neuron_y_potentials)
    current_y += self.leakage_current(neuron_y_potentials)
    return current_x / self.params['capacitance'],
        current_y / self.params['capacitance']
```

```
    def call(self, inputs, states):
        neuron_x_inputs = inputs[:, :1]
        neuron_y_inputs = inputs[:, 1:]
        neuron_x_potentials = states[0][:, :1]
        neuron_y_potentials = states[0][:, 1:]
        partial_step_size = self.params['step_size'] / self.
            discretization_steps
        for _ in range(self.discretization_steps):
            neuron_x_potentials_derivative,
                neuron_y_potentials_derivative = self.
                state_derivative(neuron_x_inputs,
                neuron_x_potentials, neuron_y_inputs,
                neuron_y_potentials)
            neuron_x_potentials +=
                neuron_x_potentials_derivative *
                partial_step_size
            neuron_y_potentials +=
                neuron_y_potentials_derivative *
                partial_step_size
        states = tf.concat((neuron_x_potentials,
            neuron_y_potentials), axis=-1)
        return states, (states,)

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'discretization_steps': self.discretization_steps
        })
        return config
```

Listing 6.21: memory_cell.py

6.4 Other Code

6.4.1 apply_and_save_statistics.py

```
"""
each invocation of run_all_benchmarks_and_models.py generates
    these four folders in the project directory (assuming
    default arguments are used): results, saved_models,
    tensorboard and visualizations
place these folders in a new folder called benchmark_logs/run_{
    i} where i is the run index after all folders have been
    manually validated
```

```
    then execute this script to generate statistics based on all
        available runs in folder benchmark_logs/statistics
    """

import argparse
import os

import numpy as np
import pandas as pd

import experiments.benchmarks.benchmark as benchmark

parser = argparse.ArgumentParser()
parser.add_argument('--result_folder_name', default='results',
    type=str)
parser.add_argument('--decimal_places', default=3, type=int)
args = parser.parse_args()

statistics_folder_path = os.path.join('benchmark_logs', '
    statistics')
os.makedirs(statistics_folder_path, exist_ok=True)

available_runs = 0
while os.path.exists(os.path.join('benchmark_logs', f'run_{
    available_runs}')):
    available_runs += 1

for benchmark_name in benchmark.BENCHMARK_NAMES:
    result_tables = []
    header = None
    first_columns = None
    for run_index in range(available_runs):
        result_table = pd.read_csv(os.path.join('benchmark_logs
            ', f'run_{run_index}', args.result_folder_name,
            benchmark_name, 'merged_results.csv'))
        result_table.sort_values(result_table.columns[0],
            inplace=True)
        current_header = result_table.columns.values.astype(str
            )
        assert header is None or np.all(header ==
            current_header)
        header = current_header
        data = result_table.values
```

```
            current_first_columns = data[:, :2].astype(str)
            assert first_columns is None or np.all(first_columns ==
                current_first_columns)
            first_columns = current_first_columns
            result_tables.append(data[:, 2:].astype(np.float64))
        assert result_tables and header is not None and
            first_columns is not None
        merged_results = np.stack(result_tables, -1)
        if merged_results.shape[1] > 4:
            loss_column = -2
        else:
            loss_column = -1
        means = np.mean(merged_results, -1)
        sort_order = means[:, loss_column].argsort()
        sorted_means = np.round(means[sort_order], decimals=args.
            decimal_places)
        sorted_first_columns = first_columns[sort_order]
        formatted_means = np.char.mod(f'%.{args.decimal_places}f',
            sorted_means)
        stds = np.std(merged_results, -1, ddof=1)
        sorted_stds = np.round(stds[sort_order], decimals=args.
            decimal_places)
        formatted_stds = np.char.mod(f'%.{args.decimal_places}f',
            sorted_stds)
        merged_results_string = np.char.add(np.char.add(
            formatted_means, ' \u00b1 '), formatted_stds)
        result_table = pd.DataFrame(np.concatenate((
            sorted_first_columns, merged_results_string), -1),
            columns=header)
        result_table.to_csv(os.path.join(statistics_folder_path, f'
            {benchmark_name}.csv'), index=False)
```

Listing 6.22: apply_and_save_statistics.py

6.4.2 run_all_benchmarks_and_models.py

```
import argparse
import os
import subprocess

import experiments.benchmarks.benchmark as benchmark
import experiments.models.model_factory as model_factory

parser = argparse.ArgumentParser()
```

```
parser.add_argument('--cuda_visible_devices', default='', type=
    str)
parser.add_argument('--result_folder_name', default='results',
    type=str)
parser.add_argument('--python_executable_name', default='
    python3.8', type=str)
args = parser.parse_args()

os.environ['CUDA_VISIBLE_DEVICES'] = args.cuda_visible_devices

for benchmark_name in benchmark.BENCHMARK_NAMES:
    for model_argument in model_factory.MODEL_ARGUMENTS:
        if not os.path.exists(os.path.join(os.path.curdir, args
            .result_folder_name, benchmark_name, model_argument,
             'training.csv')):
            if benchmark_name == 'cell' and model_argument != '
                memory_cell':
                continue
            if benchmark_name != 'cell' and model_argument == '
                memory_cell':
                continue
            subprocess.run([f'{args.python_executable_name}', '
                -m', f'experiments.benchmarks.{benchmark_name}
                _benchmark',
                            '--model', f'{model_argument}', '--
                                result_folder_name', f'{args.
                                result_folder_name}'], check=
                                True)
```

Listing 6.23: run_all_benchmarks_and_models.py

List of Figures

List of Tables

Bibliography

[AAB+15] 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, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan 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, 2015. Software available from tensorflow.org.

[AMH09] Awad Al-Mohy and Nicholas Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM Journal on Matrix Analysis and Applications*, 31, 01 2009.

[ASB16] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks, 2016.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[C+15] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[CHM+17] William R. Clements, Peter C. Humphreys, Benjamin J. Metcalf, W. Steven Kolthammer, and Ian A. Walmsley. An optimal design for universal multiport interferometers, 2017.

[CPGK19] Avraam Chatzimichailidis, Franz-Josef Pfreundt, Nicolas R. Gauger, and Janis Keuper. Gradvis: Visualization and second order analysis of optimization surfaces during the training of deep neural networks, 2019.

[CRBD19] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[GRUG17] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations, 2017.

[GSC00] Felix Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12:2451–71, 10 2000.

[GWD14] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.

[GWR+16] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[HLA+18] Ramin M. Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid time-constant recurrent neural networks as universal approximators, 2018.

[HLA+20] Ramin Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid time-constant networks, 2020.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[iFN93] Ken ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801–806, 1993.

[JSD+17] Li Jing, Yichen Shen, Tena Dubček, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns, 2017.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[LCB10] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[LH20] Mathias Lechner and Ramin Hasani. Learning long-term dependencies in irregularly-sampled time series, 2020.

[LHA+20] Mathias Lechner, Ramin Hasani, Alexander Amini, Thomas Henzinger, Daniela Rus, and Radu Grosu. Neural circuit policies enabling auditable autonomy. *Nature Machine Intelligence*, 2:642–652, 10 2020.

[LXT+18] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets, 2018.

[Ma16] Jianqiang Ma. All of recurrent neural networks. https://medium.com/@jianqiangma/all-about-recurrent-neural-networks-9e5ae2936f6e, 2016.

[MAT20] MATLAB. *R2020b*. The MathWorks Inc., Natick, Massachusetts, 2020.

[MKL17] Michael C. Mozer, Denis Kazakov, and Robert V. Lindsey. Discrete event, continuous time rnns, 2017.

[Pon62] Lev S Pontrjagin. *The mathematical theory of optimal processes*. Wiley, New York, NY [u.a.], 1962.

[Rei14] Marek Rei. Neural networks, part 3: The network. http://www.marekrei.com/blog/neural-networks-part-3-network/, 2014.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[Smi97] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, USA, 1997.

[TET12] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.