RNN: An educational implementation

Auren, Trym S.
Institute for Informatics,
University of Oslo
tsauren@uio.no

Bjerkan, Mons E.
Institute for Informatics,
University of Oslo
monseb@uio.no

Ringkjøb, Elias L.
Institute for Informatics,
University of Oslo
eliaslr@uio.no

June 7^{th} 2024

Abstract

This report covers the details of how we implemented a vanilla Recurrent neural network (RNN) in Python without the use of machine learning libraries. The project is inspired by recent advances in natural language processing. We show how our model performs on text-generation (prediction) tasks, as well as benchmarking it against PyTorch on sine wave prediction tasks. The model is able to predict text and sine waves despite its simple architecture.

1 Curtain-raiser

Anno 2024, Large language models (LLM)s are heavily used by many university students and in a wide variety of professions. LLMs leverages the huge amount of natural language data available, to learn complicated semantic relations. LLMs as we know them today, are results of years of research built on RNN models. The first appearance of RNN models in research dates back to the 1980s with papers such as 'Learning internal representations by error propagating' by Rumhelhart, et al. [1]. What separates RNNs from more conventional models such as Feed forward neural network (FFNN)s, is their ability to take advantage of the dependency between elements in a sequence.

The reason for not utilising traditional FFNNs for the sequential tasks such as text generation, is because the data given to such models is assumed to be independent and identically distributed (**i.i.d.**), and the model cannot handle varying input length. RNNs overcomes this issue through parameter sharing. This means that all the data in a sequence will share the same parameters. There are some advantages of parameter sharing. For example:

- If we were to teach a model to extract what a person eats from a sequence of words, for instance a sentence, we want the item the person ate to be important, not where the item is positioned within a sequence. For example, for the sequences *I eat cake* and *I like to eat cake when I am alone*, the extracted information should in both cases be *cake*. If a FFNN was to learn the same thing, it would not share the same parameters for each word in the sequence, and therefore it would be necessary that it learns to extract *cake* from all possible positions in the sequence.
- The model can handle any finite length of sequences, also sequences with lengths not seen during training.

In addition, RNNs are able to encode experience in a *hidden state*. The hidden state is a vector (or a matrix for a batch of sequences) that stores information about the sequence processed so far. It acts as the memory of the network, capturing the context from previous time steps and allowing the network to maintain information across different time steps in the input sequence. How rich the hidden state should be (how many hidden nodes to use), is generally controlled through a hyperparameter, as with the amount of hidden nodes used in a hidden layer in a FFNN.

Many different RNN architectures exist. In this project, we focus on predicting the next words or characters in a sequence given the previously seen values in the sequence. To do this, an RNN architecture that produces output at each time step and has hidden-to-hidden recurrent connections, is employed. The network has only one hidden layer, and is often referred to as a *vanilla RNN* in literature. The computational graph of a such model can be seen in fig. 1.

To the best of our knowledge[1], the implementation presented in this paper is the first non-funded + object-oriented + readable + modular implementation of an RNN. The model's performance is comparable to that of PyTorch [3] when benchmarked on sine predictions on a wide variety of training configurations. It is also 28%

[1] extensive Google searches

faster on average for the sine wave prediction task. The model also performs well in the task of Natural language processing (NLP), creating cohesive sentences when trained on the first Harry Potter book with pre-trained word embeddings as text-encoding.

In the method section (section 2), we start by discussing data processing (section 2.1), before moving on to the python implementation of the RNN (section 2.2). Here, the forward and backward pass is discussed in detail, before describing the different optimisation algorithms and gradient regularisation techniques implemented. Our novel approach of implementing batch-processing of data in a Numpy-only RNN is also described. We then present the experiments conducted in section 3, both for natural language, but also for different hyperparameter configurations on sine wave prediction tasks and training speed tests - with some comments on the results. The report ends with a discussion of the challenges and suggestions to further work, in (section 4). Pseudocode can be found in the appendix (appendix A.

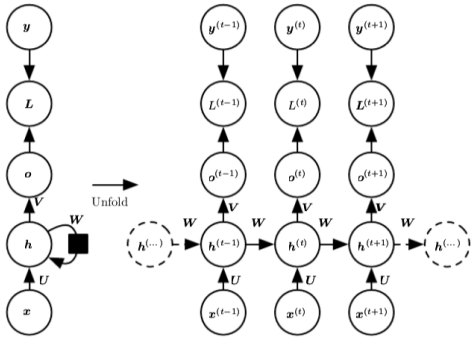


Figure 1: Vanilla RNN computational graph. $\mathbf{x^{(t)}}$ denotes the element at position (time) t in the sequence $\mathbf{X}$. $\mathbf{h^{(t)}}$ denotes the hidden state at time t. $\mathbf{o^{(t)}}$ denotes the output at time t. $\mathbf{U}, \mathbf{W}, \mathbf{V}$ denotes weights from input to hidden, hidden to hidden, and hidden to output, respectively. The left illustration is more comparable to traditional feed forward networks, while it is important to keep the time aspect in mind when looking at the unrolled illustration to the right. The figure is borrowed from [2].

2 Method

2.1 Data

In the experiments, we deploy two types of data: natural language data, and sinusoidal waves. The natural language data utilised is J.K. Rowling's Harry Potter (book 1), Shakespeare's Romeo and Juliet, and a sentence "There was a big black cat". The sinusoidal waves are generated using Numpy. Below, we discuss our implementation of natural language data processing (not part of the RNN implementation), before moving on to the model implementation.

2.1.1 Natural language data processing

There are two main methods for converting natural language into representations understandable by machine learning models: one-hot encoding and word embeddings.

One-Hot Encoding: This simpler method represents each word or character as a vector, with a length equal to the number of possible words or characters. For words, this length is typically the size of a dictionary. For characters, the vector length is equal to the number of distinct characters in the training data, or it could be the amount of ASCII characters available (256). Each vector contains all zeros except for a single one(1) at the position corresponding to the word or character index in the dictionary. Despite its simplicity, one-hot encoding can be computationally heavy if using it for word encoding, and it does not capture semantic relationships between words.

Word Embeddings: This more sophisticated method converts words into dense vectors of floating-point numbers. These vectors are usually much shorter than one-hot encoded words, making them more efficient in terms of computation and storage. The key advantage of word embeddings is their ability to capture semantic relationships between words. Words with similar meanings or contexts are placed closer together in a multidimensional vector space, allowing the model to measure semantic similarity. This approach provides richer, context-aware representation of text, able to incorporate relationships such as synonyms.

When using word embeddings in neural network based machine learning, it is normal to include an embedding layer as the first layer of the network. The function of this layer is to train and tailor the word embeddings to the training data. It should be mentioned that it is normal to start with a dataset of already pre-trained word embeddings because the amount of data and computational power needed to create and train a word embedding dataset of any substantial size from scratch is quite large. Fine-tuning a dataset of pre-trained embeddings using an embedding layer is computationally cheaper and faster. For the implementation seen in this report, it was decided to omit the embedding layer and not train the embeddings from the dataset any further. This limits

the results the model produces. However, training it on a dataset with general word embeddings should suffice since the model doesn't have any specific tasks to perform, other than general text generation/completion.

Our approach Our model tackles both one-hot and embedding representations. Spacy [4] is used for embedding text as vectors. The library provides a large set of pre-trained embeddings with an embedding size of 300. The utility code for pre-processing and post-processing natural language data is provided separately from the RNN class code, in `utils/text_processing.py`.

2.2 RNN Python implementation

The purpose of this project is to investigate the math and implementations underlying a RNN. Therefore, the RNN is implemented without traditional machine learning (ML) libraries like PyTorch or Scikit-learn. All code is written in Python, using Numpy [5] for numerical processing, such as vector and matrix multiplication. Nevertheless, the implementation handles a variety of data types and lengths and allows for configurability of optimisation algorithms and activation functions. None of the implementations exploits General processing unit (GPU)s or Tensor processing unit (TPU)s, which has been found to increase computation speed for some RNN models [6]. The implementation may therefore be slower than its GPU or TPU optimised counterparts.

The remainder of the method section is devoted to a detailed explanation of the implemented code, with an emphasis on the mathematical components. The code is object oriented, and has a user-interface that somewhat resembles Scikit-learn [7]. For examples on how to build, train, and use your own model, see README files in the public code repository at `https://github.com/trymauren/FYS5429`. Although not all parts of the explanation can be directly tied to code, links to the public repository are provided wherever applicable. Additionally, pseudocode for the various parts is included in appendix A. For each of the parts, we first explain the general concept, such as what the forward pass does, and then describe our specific method.

Model architecture

In general The number of input nodes N is usually determined based on the size of each sequence *value*, which corresponds to the number of features in standard FFNNs. The number of hidden nodes H is usually determined by the user, and essentially decides how rich the hidden state of the network should be, as in FFNNs. The number of output nodes K is determined by the size of the values in the target sequence [2]. This means that the input to hidden weights $\mathbf{U}$, the hidden-to-hidden (recurrent) weights $\mathbf{W}$ and the hidden-to-output weights $\mathbf{V}$ have the following dimensions:

$$\mathbf{U} \in \mathbb{R}^{H \times N}$$
$$\mathbf{W} \in \mathbb{R}^{H \times H}$$
$$\mathbf{V} \in \mathbb{R}^{K \times H}$$

The bias terms, which are added to the hidden- and output layers, $\mathbf{b}$ and $\mathbf{c}$ respectively, have the following dimensions:

$$\mathbf{b} \in \mathbb{R}^{H}$$
$$\mathbf{c} \in \mathbb{R}^{K}$$

Note: The input/hidden/output nodes described above should not be confused with the nodes in the unrolled computational graph in fig. 1. The nodes seen horisontally in the hidden layer in the unrolled graph are referred to as hidden states in RNN literature, and the amount of them depends on the length of the sequence processed. For a sequence length of S and batchsize of B, the dimension of the hidden state $\mathbf{h}$ is

$$\mathbf{h} \in \mathbb{R}^{S \times B \times H}$$

Our approach The parameters described above corresponds to our implementation. A difference from more advanced libraries, is that our model only implements one hidden layer - for simplicity. In multi-layer implementations, there would be several hidden weight matrices - not only one, as in our implementation.

Input data

In general The input from time $t = 0$ to time τ flows through the network sequentially. The amount of time steps can vary. All input samples are therefore assumed to have a (possibly varying length) τ. There is a huge variety in how RNNs are trained, based on the task and data. If the task is to learn to write Twitter posts, the model may be trained on a dataset where each sample is a Twitter post. Each data sample will then have

a sequence length corresponding to the number of words (embeddings) in the post, each entry in the sequence with a size equal to the representation of the data at hand. If the task is instead to generate monologue from Romeo (Shakespeare), the model can be trained on *one* sample: the very long sequence corresponding to the full manuscript of Romeo and Juliet.

Our approach Training data is passed to the network as a matrix

$$\mathbf{X} \in \mathbb{R}^{M \times S \times B \times N},$$

where m = number of samples, S = length of sequence, B = batchsize, N = features. The model assumes that *all samples are independent* of the others. In other words, the RNN will not be able to learn connections between sentences if sentences from a paragraph is passed as individual samples. The same goes for batched input: all members of the batch are assumed by the model to be independent.

Going forwards

We have based our implementation of the forward pass on [2], chapter 10. Note that although the implementation handles batches, we have not included the batch dimension in the equations for forwards-and-backwards-propagation for simplicity.

Our approach All data samples share the same afore-mentioned set of parameters which we can denote as $\Theta = \{\mathbf{U}, \mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}\}$. A forward pass of one data sample can be described by the following equations, which are applied to all time steps of that sample:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(\mathbf{t-1})} + \mathbf{U}\mathbf{x}^{(\mathbf{t})} \tag{1}$$

$$\mathbf{h}^{(t)} = \text{ACTIVATION}(\mathbf{a}^{(t)}) \tag{2}$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \tag{3}$$

$$\hat{\mathbf{y}}^{(t)} = \text{POSTPROCESSING}(\mathbf{o}^{(t)}) \tag{4}$$

Here, $\mathbf{a}^{(t)}$ and $\mathbf{h}^{(t)}$ are the in- and output values from the hidden state at timestep t, and $\mathbf{o}^{(t)}$ and $\hat{\mathbf{y}}^{(t)}$ are the in- and outputs from the output layer at timestep t, respectively. Both $\mathbf{h}$ and $\mathbf{x}$ are stored for use in backpropagation.

`rnn/rnn.py.forward()` calls `rnn/rnn.py.forward_unit()`, which executes the equations above for all elements in the sequence to be forward-propagated.

The ACTIVATION function employed in the experiments is the hyperbolic tangent function

$$f(z) = \tanh(z). \tag{5}$$

tanh is closely related to the sigmoid activation function but resembles the identity function more than the sigmoid, as $\tanh(0) = 0$, while $\sigma(0) = \frac{1}{2}$. The trait of resembling the identity function makes optimisation easier, because optimising the network resembles optimising a linear model. [2]. The Rectified linear unit (ReLU) could also be a candidate as an activation function. However, due to the number of times the computational graph is unrolled, the property of keeping activation outputs in the range -1,1 is a favorable feature of tanh [2]. The ReLU, identity, and sigmoid activation functions are also included in the code for completeness but have not been thoroughly experimented with and verified, because of the advantage of employing the tanh function. The user is free to choose between these activations when initialising the model.

The POSTPROCESSING function $g(z)$, is either the Softmax function

$$g(z) = \frac{\exp(z)}{\sum \exp(z)}, \tag{6}$$

when doing classification, or the identity function,

$$g(z) = z, \tag{7}$$

when doing regression. The softmax function is used to convert the outputs to normalised probabilities to evaluate which label is the most probable. We employ softmax in NLP experiments, and the identity function in the sine wave experiments. The user is free to choose activation functions for the hidden layer and postprocessing functions freely from the available activation functions. See `utils/activations.py` for activation functions.

Going backwards

As with the forward section, this section is influenced by [2], chapter 10.

Our approach In addition to what is provided by [2], chapter 10, we provide a conversion of the variable names in the code to the different gradients computed below. In the case of classification, the goal is to estimate some output conditioned on input to the model. For neural networks and other parameterised models, this is achieved by estimating parameters Θ, typically under the maximum likelihood framework. We apply this framework and therefore use LOGLOSS for the classification case. MEAN SQUARE ERROR LOSS is used when doing regression, where the error is assumed to follow a normal distribution.

To adjust parameters, we apply a modified version of the general backpropagation algorithm for shared parameters called Backpropagation through time (BPTT), as described in [2], chapter 10. The contribution of each parameter to the error of the model is calculated through this algorithm, by differentiating from the error at output, backward through time to each parameter.

The gradient of the loss function must propagate backward through time, from time $t = \tau$ down to $t = 0$, but also from output to the parameters at the input. Below, we derive all the equations involved in this process. The first move is to calculate the loss with respect to the current time t.

$$\frac{\delta C}{\delta C^{(t)}} = 1$$

There is a need to distinguish the second move, which is to calculate the gradient of the loss function at the output nodes, between classification (LOGLOSS) and regression (MSE).

Assuming classification with LOGLOSS and SOFTMAX as a postprocessing step, the gradient of the cost function at the output, $\mathbf{o}$, at time t is

$$\nabla_{\mathbf{o}^{(t)}} C = \frac{\delta C}{\delta o^{(t)}} = \frac{\delta C}{\delta C^{(t)}} \frac{\delta C^{(t)}}{\delta o^{(t)}} = \hat{y}^{(t)} - \mathbf{1}_{i=y^{(t)}} \tag{8}$$

Assuming regression with MSE and IDENTITY as postprocessing step, the gradient of the cost function at the output, $\mathbf{o}$, at time t is

$$\nabla_{\mathbf{o}^{(t)}} C = \frac{\delta C}{\delta o^{(t)}} = \frac{\delta C}{\delta C^{(t)}} \frac{\delta C^{(t)}}{\delta o^{(t)}} = \frac{2}{\texttt{size}(\hat{y})} (\hat{y}^{(t)} - y^{(t)}) \tag{9}$$

The next step is to find an expression for the gradient of the final hidden (computational) node, at time τ: $\mathbf{h}^{(\tau)}$. Its only descendant is $\mathbf{o}^{(\tau)}$, which means its gradient calculation is solely dependent on this node in the computational graph, making it a good starting point for the later gradient calculations.

$$\nabla_{\mathbf{h}^{(\tau)}} C = (\nabla_{\mathbf{o}^{(\tau)}} C) \frac{\delta \mathbf{o}^{(\tau)}}{\delta \mathbf{h}^{(\tau)}} \tag{10}$$

$$= (\nabla_{\mathbf{o}^{(\tau)}} C) \mathbf{V} \tag{11}$$

$$\nabla_{\mathbf{h}^{(\tau)}} C = \mathbf{V}^\top (\nabla_{\mathbf{o}^{(\tau)}} C) \tag{12}$$

Where all the right-hand side terms are known from before.

The nodes which need gradient computation now, are all the hidden states preceding the last. I.e., for $\mathbf{h}^{(t)}$, where $t = \tau - 1, \tau - 2, \ldots, 0$. For these time steps, the gradient is influenced by both the gradient w.r.t. $\mathbf{o}^{(t)}$, as well as all the preceding hidden state gradients. Remember that the preceding hidden state of $\mathbf{h}^{(t)}$ is $\mathbf{h}^{(t+1)}$, which has preceding hidden state $\mathbf{h}^{(t+2)}$, and so on. We are calculating the gradient starting from $t = \tau - 1$, working our way down to $t = 0$:

$$\nabla_{\mathbf{h}^{(\tau-1)}} C = \nabla_{\mathbf{o}^{(\tau-1)}} C \frac{\delta \mathbf{o}^{(\tau-1)}}{\delta \mathbf{h}^{(\tau-1)}} + \nabla_{\mathbf{h}^{(\tau)}} C \frac{\delta \mathbf{h}^{(\tau)}}{\delta \mathbf{h}^{(\tau-1)}} \tag{13}$$

$$\nabla_{\mathbf{h}^{(\tau-2)}} C = \nabla_{\mathbf{o}^{(\tau-2)}} C \frac{\delta \mathbf{o}^{(\tau-2)}}{\delta \mathbf{h}^{(\tau-2)}} + \nabla_{\mathbf{h}^{(\tau-1)}} C \frac{\delta \mathbf{h}^{(\tau-1)}}{\delta \mathbf{h}^{(\tau-2)}} \tag{14}$$

$$\nabla_{\mathbf{h}^{(\tau-3)}} C = \nabla_{\mathbf{o}^{(\tau-3)}} C \frac{\delta \mathbf{o}^{(\tau-3)}}{\delta \mathbf{h}^{(\tau-3)}} + \nabla_{\mathbf{h}^{(\tau-2)}} C \frac{\delta \mathbf{h}^{(\tau-2)}}{\delta \mathbf{h}^{(\tau-3)}} \tag{15}$$

$$= \nabla_{\mathbf{o}^{(\tau-3)}} C \frac{\delta \mathbf{o}^{(\tau-3)}}{\delta \mathbf{h}^{(\tau-3)}} + \left(\nabla_{\mathbf{o}^{(\tau-2)}} C \frac{\delta \mathbf{o}^{(\tau-2)}}{\delta \mathbf{h}^{(\tau-2)}} + \nabla_{\mathbf{h}^{(\tau-1)}} C \frac{\delta \mathbf{h}^{(\tau-1)}}{\delta \mathbf{h}^{(\tau-2)}} \right) \frac{\delta \mathbf{h}^{(\tau-2)}}{\delta \mathbf{h}^{(\tau-3)}} \tag{16}$$

$$= \nabla_{\mathbf{o}^{(\tau-3)}} C \frac{\delta \mathbf{o}^{(\tau-3)}}{\delta \mathbf{h}^{(\tau-3)}} + \tag{17}$$

$$\left(\nabla_{\mathbf{o}^{(\tau-2)}} C \frac{\delta \mathbf{o}^{(\tau-2)}}{\delta \mathbf{h}^{(\tau-2)}} + \left(\nabla_{\mathbf{o}^{(\tau-1)}} C \frac{\delta \mathbf{o}^{(\tau-1)}}{\delta \mathbf{h}^{(\tau-1)}} + \nabla_{\mathbf{h}^{(\tau)}} C \frac{\delta \mathbf{h}^{(\tau)}}{\delta \mathbf{h}^{(\tau-1)}} \right) \frac{\delta \mathbf{h}^{(\tau-1)}}{\delta \mathbf{h}^{(\tau-2)}} \right) \frac{\delta \mathbf{h}^{(\tau-2)}}{\delta \mathbf{h}^{(\tau-3)}} \tag{18}$$

Generalising this to

$$\nabla_{\mathbf{h}^{(t)}} C = (\nabla_{\mathbf{o}^{(t)}} C) \frac{\delta \mathbf{o}^{(t)}}{\delta \mathbf{h}^{(t)}} + (\nabla_{\mathbf{h}^{(t+1)}} C) \frac{\delta \mathbf{h}^{(t+1)}}{\delta \mathbf{h}^{(t)}} \tag{19}$$

$$\nabla_{\mathbf{h}^{(t)}} C = (\nabla_{\mathbf{o}^{(t)}} C)\ \mathbf{V^{(t)}} + (\nabla_{\mathbf{h}^{(t+1)}} C)\ \mathbf{W^{(t+1)}} \tag{20}$$

Some parts of the equations above have been colored to emphasize the parts that we take with us from one gradient calculation to the next.

It is observable that the gradient at time t is indeed dependent on all later time steps, as well as the current output. The influence from current output can be seen to the left of the (+) in the equation right above , while the influence from previous states can be observed to the right of the (+).

The final step is to calculate the gradient of the parameters $\mathbf{U}, \mathbf{W}, \mathbf{b}, \mathbf{V}, \mathbf{c}$. To find these gradients, we must differentiate $\mathbf{h}^{(t)}$. Calculating the derivative of $\mathbf{h}^{(t)}$ involves differentiating the activation function using the chain rule. In the equations below, the chain rule is applied to differentiate the activations with respect to different parameters, but the activation function itself is not differentiated because it depends on which activation function is in use. It is instead denoted as $\nabla_{\text{ACTIVATION}}$.

$$\begin{aligned}
\nabla_{\mathbf{c}} C &= \sum_t (\nabla_{\mathbf{o}^{(t)}} C) \frac{\delta \mathbf{o}^{(t)}}{\delta \mathbf{c}^{(t)}} &&= \sum_t (\nabla_{\mathbf{o}^{(t)}} C)\, 1 \\
\nabla_{\mathbf{V}} C &= \sum_t (\nabla_{\mathbf{o}^{(t)}} C) \frac{\delta \mathbf{o}^{(t)}}{\delta \mathbf{V}^{(t)}} &&= \sum_t (\nabla_{\mathbf{o}^{(t)}} C)\, \mathbf{h^{(t)}} \\
\nabla_{\mathbf{b}} C &= \sum_t (\nabla_{\mathbf{h}^{(t)}} C) \frac{\delta \mathbf{h}^{(t)}}{\delta \mathbf{b}^{(t)}} &&= \sum_t (\nabla_{\mathbf{h}^{(t)}} C \cdot \nabla_{\text{ACTIVATION}})\, 1 \\
\nabla_{\mathbf{W}} C &= \sum_t (\nabla_{\mathbf{h}^{(t)}} C) \frac{\delta \mathbf{h}^{(t)}}{\delta \mathbf{W}^{(t)}} &&= \sum_t (\nabla_{\mathbf{h}^{(t)}} C \cdot \nabla_{\text{ACTIVATION}})\, \mathbf{h}^{(t-1)} \\
\nabla_{\mathbf{U}} C &= \sum_t (\nabla_{\mathbf{h}^{(t)}} C) \frac{\delta \mathbf{h}^{(t)}}{\delta \mathbf{U}^{(t)}} &&= \sum_t (\nabla_{\mathbf{h}^{(t)}} C \cdot \nabla_{\text{ACTIVATION}})\, \mathbf{x}^{(t)}
\end{aligned}$$

Below are conversions from code to math for gradient calculations executed in `rnn/rnn.py._backward()`:

`loss_grad`	$\Rightarrow \nabla_{\mathbf{o}} C$
`grad_o_Cost`	$\Rightarrow \nabla_{\mathbf{o}^{(t)}} C$
`d_act`	$\Rightarrow \nabla_{\text{ACTIVATION}}$
`grad_h_Cost_raw`	$\Rightarrow$ `d_act` @ `W`
`prev_grad_h_Cost`	$\Rightarrow$ `grad_h_Cost_raw` @ `W`
`grad_h_Cost`	$\Rightarrow$ `grad_o_Cost` @ `V` + `prev_grad_h_Cost`

Important note, which makes this code work: `prev_grad_h_Cost` is initialised to $\mathbf{0}$, and is *lagging* one time step. This means that `grad_h_Cost` is not influenced by anything else than the gradient at the final output node, before moving from time τ to $\tau - 1$.

Gradient issues

Using techniques to avoid exploding or vanishing gradients is a necessity in RNNs [8], [9], [10].

Our approach To mitigate the potentially extreme values, we employ two methods:

1. The number of time steps to forward/backpropagate can be determined by the user (through parameter `unrolling_steps`, and should be kept lower than the sequence length for long sequences. Specifying `unrolling_steps` = 10 when the sequence has length 20, will lead to the following: First the model propagates 10 timesteps forward. Then, backpropagation is performed for the 10 timesteps and the parameters are updated. The last hidden states is retained, and the rest of the sequence (10-20) is then processed forwards and backwards. This procedure also lead to less memory consumption, since only a maximum of `unrolling_steps` number of hidden states must be stored at a time. Limiting the number of bacpropagation steps is generally referred to as truncated BPTT (TBPTT). See this phd thesis for more information on TBPTT.

2. Gradient clipping is performed by normalising the gradient of each parameter, as suggested in [8]. We use the `Numpy.linalg.norm` with the Frobenius norm, as described in Numpy.linalg.norm.

Gradient checking

Training machine learning models with gradient descent is very dependent on the gradients in BPTT being correct [8]. However, it can be hard to know if this is entirely correct for a couple reasons; (1) the objective objective function may be complex; (2) incorrect gradients may not necessarily lead to large problems in training.

To combat the issues described above, one way to do so is to implement *gradient checking*. Gradient checking compares the analytical gradient calculated by backpropagation to an estimated numerical gradient.

Say we have the objective function $f(\theta)$ where $f : \Re^n \to \Re$ and the gradient $g_i(\theta) = \frac{d}{d\theta_i} f(\theta)$.The numerical derivative of f is

$$\frac{d}{d\theta} f(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \tag{21}$$

Our approach By the definition above, we can estimate the derivative for a given value of a parameter-entry θ_{ik} with the following property:

$$\frac{f(\theta_{ik} + \epsilon^*) - f(\theta_{ik} - \epsilon^*)}{2\epsilon^*} \tag{22}$$

I.e., each parameter entry in the weights matrices and bias vectors is perturbed to find a numerical gradient, before checking this gradient against the one returned from the implementation of BPTT. Note that ϵ^* represents the implementation of ϵ in equation (21) as a very small constant [2]. We expect the result of equation (22) to be equal to equation (21) to at least a few significant digits, and for it to be more accurate the lower the value of ϵ^*.

To perform gradient checking, a keyword-parameter `gradcheck=i` can be passed when fitting the model, where `i` specifies which epoch to run gradient check at. Gradcheck is slow and there is no reason to do grad check unless for testing purposes. See here for gradient check code. For examples on how to run gradient check, see `run-gradcheck` directory. Note that gradient checking should not be done in the start of training, but rather after things have stabilised.

Optimisation

Four algorithms for minimising the loss functions are included: Stochastic Gradient Descent (SGD), SGD with momentum, Adaptive Gradient Algorithm (AdaGrad), and Adam. Pseudocode for the optimisers can be viewed in appendix A. See `utils/optimisers.py` for optimiser code.

Stochastic Gradient Descent (SGD) SGD, described in algorithm 2, is a prevalently used optimization algorithm in machine learning (ML) for updating parameters. The idea of this algorithm is that you move along the parameter space to find the values of each parameter which minimises the loss function.

Say we have a parameter $\theta \in \Theta$. This parameter is minimized by finding the gradient $\mathbf{g}$ of Θ with respect to θ, and adding the scaled (w.r.t. the learning rate ϵ) gradient to the parameter. Eventually, after optimising each $\theta \in \Theta$, we will have optimised Θ w.r.t. all $\theta \in \Theta$. [2]

SGD w/ momentum The term *momentum* comes from Newtonian physics, in the sense that it takes inspiration from the Newtonian physics of a moving particle through an N-dimensional space. where $N = |\Theta|$. [2]

Essentially, when adding momentum to the SGD optimisation algorithm, we are keeping track of past gradients by accumulating an exponentially decaying moving average of the gradient w.r.t. θ. This ensures that the algorithm doesn't get as easily stuck in local minima.

Given a parameter $\theta \in \Theta$, a momentum α and a learning rate ϵ, we can calculate a velocity as such:

$$\nu \leftarrow \alpha\nu - \epsilon\mathbf{g}, \tag{23}$$

where $\mathbf{g}$ is the gradient of Θ w.r.t. θ. We then update the parameter: $\theta \leftarrow \theta + \nu$. See algorithm 3 for a pseudocode describing the *SGD with momentum* algorithm.

[2] Generally, $\epsilon^* \in [1e^{-4}, 1e^{-10}]$, but there is no good guide to choosing its value

AdaGrad The AdaGrad algorithm is described in algorithm 4. The algorithm adapts the learning rate to the gradient of each hyperparameter by accumulating the sum of the gradient over time and computing the parameter update value as the inverse of the square of the parameter gradient accumulate. [2]

This ensures the parameters which have greater partial derivatives end up with a greater decrease in the learning rate, and vice versa. The idea of AdaGrad is for it to result in rapid convergence in the case of a convex parameter space.

Adam The objective of the Adam [11] algorithm is, just like other optimisation algorithms, to minimise the expected output from an objective function f w.r.t. parameters $\theta \in \Theta$.

The algorithm updates a moving average (1^{st} moment) of the gradient, $\mathbf{s}$, as well as a moving average of the uncentered variance (2^{nd} moment) of the gradient, $\mathbf{r}$ (see also algorithm 5) [2], [11]:

$$\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g} \tag{24}$$

$$\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g} \tag{25}$$

Since $\mathbf{s}$ and $\mathbf{r}$ are instantialized to $\mathbf{0}$- with decay rates ρ_1, ρ_2 close to 1 (small decay rate)- the moment estimates become biased towards $\mathbf{0}$. [11] This is corrected as follows:

$$\hat{\mathbf{s}} = \frac{\mathbf{s}}{1 - \rho_1^t} \tag{26}$$

$$\hat{\mathbf{r}} = \frac{\mathbf{r}}{1 - \rho_2^t} \tag{27}$$

Here follows a resoning for this bias correction (refer to [11] for a more detailed reasoning):
The update of the uncentered variance at time t can be denoted as:

$$v_t = (1 - \rho_2) \sum_{i=0}^{t} \rho_2^{t-i} \cdot g_i^2 \tag{28}$$

We want $E[v_t]$ to describe $E[g_t^2]$ as best as possible. If we solve for $E[v_t]$, we get the following expression:

$$E[v_t] = E[g_t^2] \cdot (1 - \rho_2^t) + \zeta \tag{29}$$

where $\zeta = 0$ when $E[g_t^2]$ is stationary, and is a very small constant otherwise. Lastly, we see that if we divide equation (29) by $(1 - \rho_2^t)$ we have that the resulting bias-corrected term approximately equals $E[g_t^2]$. [11]

Our Approach When calling the optimiser in the `rnn/rnn.py._backward` pass method; rather than giving it the parameters directly; we give the gradients of the parameters instead. The update values are then returned as outputs from the method, rather than updating the parameters directly. The update values are then added to the parameter values inside of the `rnn/rnn.py._fit` method.

Random control

The model is by default seeded, to ensure reproducible results. The seed can be changed through a seed parameter when building it. Initialization values of all parameters (not hyperparameters) are drawn from the uniform distribution, within the range -0.3, 0.3. The hidden state is always initialised to 0.

Batch training

In general Processing more than one input sample at a time is called (mini) batch-processing. Usually, when training time series models on one long text, this long sequence is split into smaller sub-sequences. The smaller sub-sequences can then processed as individual samples in a (mini) batch, if one is willing to throw away the very-long-term dependencies.

Splitting long sequences into multiple smaller, is one way to do batch-processing in RNNs, resulting in reduced training time at the cost of a less accurate gradient approximation. However, this process is only valid for datasets where the data is one long sequence. When training time series models on for instance twitter or reddit posts, one would instead batch several posts. How batch-processing is used in RNN training is therefore very data-dependent.

Our approach We implement batch-processing only on the sequence level, by splitting long sequences into batch-size number of sub-sequences. This process is done before passing data to the RNN. Implementing batch-processing for a Numpy-only vanilla RNN is a novel approach. However, it is necessary to avoid a single-core dependent implementation.

3 Experiments

This section is divided into three: The first part will describe regression experiments: sine wave prediction, while the second will describe NLP experiments. Lastly, there is a short part about the training speed. It should be noted that any attempts to recreate our results using the provided Python scripts may have some deviations compared to the results shown in the paper. This is due to variations in versioning and platform, causing differences in the pseudo-randomness provided by Numpy.

3.1 Periodic waves

Using RNNs for prediction of periodic waves, such as sine waves, is one of the basic tasks an RNN should be able to handle. We compare our model to a PyTorch RNN model over a range of different data configurations and hyperparameters, such as hidden size (number of hidden nodes), learning rate and type of optimisers. The optimisers used in the experiments were SGD, SGD w/ momentum, Adagrad, and Adam, and for each of these, the models are tested using 2,10,20,30,40,50 and 60 hidden nodes with learning rates 0.001, 0.003, 0.005, 0.007 and 0.009 on each hidden layer size. The configurations yield varying results, but under three different tests both models are able to generate clear periodic-like waves. The tests consist of:

- Predict a sine with random amplitude and phase when trained on a dataset of sine-waves with random amplitude and phase.
- Predict a smooth sine from a noisy sine-wave when trained on a dataset of noisy sine-waves.
- Predict a sine when trained only on a single sine-wave.

3.1.1 Random sine prediction

The data used to train the models for this comparison consists of 10 randomly generated sine-waves, each wave consisting of 200 timesteps. The waves are generated with an amplitude drawn from a uniform distribution in range (-1, 1) and a phase shift drawn from a uniform distribution in range $(-\pi, \pi)$. The same dataset and waves are used for all tests and predictions, to be able to compare the models. Both our model and the PyTorch model were trained for 500 epochs on each of the different hyperparameter configurations.

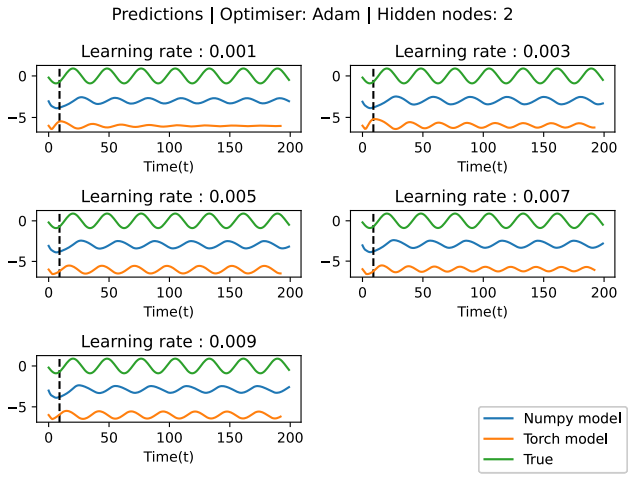
For the predictions, we seed each model with the first 10 samples of our validation wave, letting the model generate the remaining 190 samples.

Using AdaGrad, the predicted waves tend to converge toward a flat line further into the predictions for most learning rates and hidden nodes. With more hidden nodes; 40-60, we saw pronounced changes in the waves predicted by our Numpy model. The best result (visually validated) achieved by the Numpy model was achieved when using 50 hidden nodes and a learning rate of 0.009, while the pytorch model did best using 10 hidden nodes and a learning rate of 0.007, see fig. 6. Not achieving better results than what we did using AdaGrad indicates that we could have tested a wider range of parameters for this optimiser.

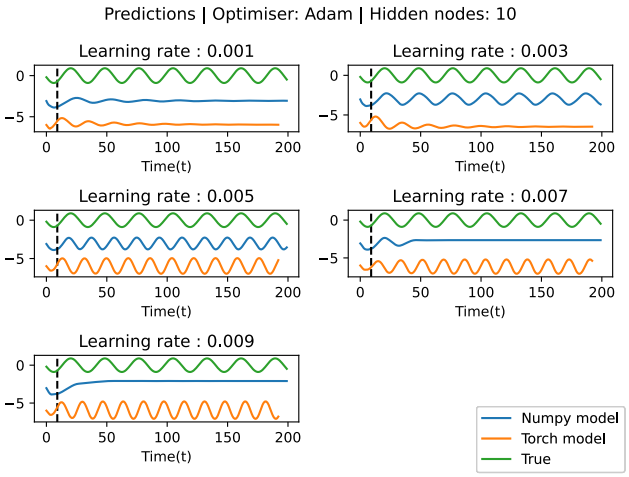
Using SGD, the models displayed varying results. However, both models achieved best performance with 10 hidden nodes, producing periodic-like waves at learning rates of 0.007 and 0.009. The PyTorch model performed well at both learning rates, while the Numpy model achieved its best result only at a learning rate of 0.009. The PyTorch model provided the best prediction out of the two. However, when using more hidden nodes, the Numpy model also provides good predictions at 50 hidden nodes using learning rates of 0.007 and 0.009, which the Pytorch model did not reciprocate. See fig. 7.

Adding momentum to SGD yielded improved results for both models, and is the first optimiser where both models at several different configurations predict waves closely resembling the validation wave. The Numpy model slightly out-performs the PyTorch model, generating a nearly identical copy of the validation wave when using 50 hidden nodes and a learning rate of 0.007. In contrast, at the same configuration the PyTorch model converged to a flat line, and the best result the PyTorch model produced was using only 2 hidden nodes and a learning rate of 0.009, see fig. 8.

Lastly, the optimiser yielding the best results was Adam for both models. At least one of the two models manage to generate waves at most configurations. The PyTorch model performed good across the spectrum, with good results at 2, 10, 20, and 60 hidden nodes and varying learning rates, generating non-converging waves at all tried hidden layer sizes. The best results the PyTorch model produced were using 2 and 10 hidden nodes, with the single best ones using 2 hidden nodes and learning rates of 0.005 and 0.009. The Numpy model has, in contrast to the PyTorch model, very varying results depending on the learning rate on some hidden layer sizes, but yield at least one non-converging periodic wave for each hidden layer size. Still, the Numpy model manages to generate the best result of both models across all configurations with its best prediction here being almost visually identical to the validation wave when using 10 hidden nodes and a learning rate of 0.003. See fig. 2 (fig. 9 for predictions with corresponding loss).



(a) The best result produced by the PyTorch model can be seen in this plot using 2 hidden nodes and a learning rate of 0.009



(b) The best result produced by the NumPy model can be seen in this plot using 10 hidden nodes and a learning rate of 0.005

Figure 2: The Adam optimiser achieved the best prediction result for both the PyTorch- and NymPy models when predicting sine waves after being trained on a set of random sine waves for 500 epochs.

3.1.2 Noisy sine predictions

To assess the model's performance in a potentially more real-world scenario, we opted to train the models on a dataset consisting of 10 sine waves with added Gaussian (normally distributed random) noise, rather than solely relying on training them to predict sine waves from random waves. The goal is that when they're passed a noisy sine wave, they'll predict the corresponding non-noisy wave. The experiments were conducted using Adam, as it is the optimiser that has performed best on average. For this test, a trend where the Numpy model and the PyTorch model provided similar predictions can be observed, but the PyTorch model out-perform the Numpy model slightly, as the Numpy model seemed to struggle with getting the phase shift right. The PyTorch model performed best using 10 and 50 hidden nodes, generating clear waves with corresponding frequency and phase shift to the noisy validation wave, while the Numpy model never managed to both remove all noise and have the correct phase at the same time. fig. 10 shows the results for different learning rates when using 2,10,20,30,40 and 50 hidden nodes.

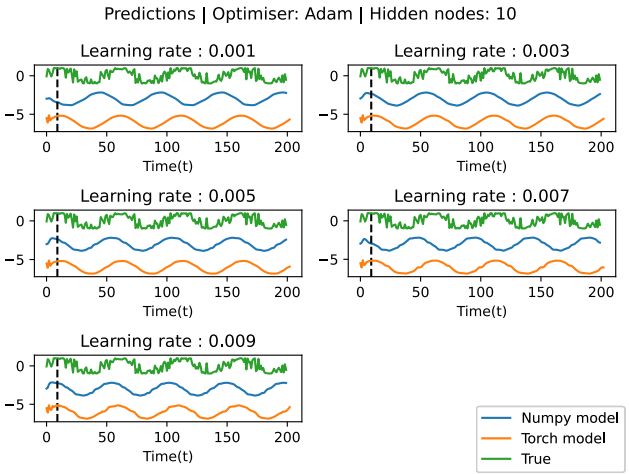


Figure 3: Predictions made by the PyTorch and NumPy models after training on noisy sine waves, using 10 hidden nodes, for 500 epochs

3.1.3 Plain sine prediction

The last sinusoidal wave prediction test was with the models only trained on one single sine wave. Both models are able to generate clear wave predictions on most configurations. Here, same as for noisy waves, we also only used Adam as an optimiser, and attempted to predict the same wave we trained on. Due to the simplicity of this task compared to the previous two, the models were only trained for 200 epochs and were only given 3 seed values for predictions. The sequence length is still 200, and the same range of hidden nodes and learning rates as in the previous tests are used. All hidden layer sizes show good results, except when using 2 hidden nodes, where the predictions flatlined immediately for both models. This could probably be counteracted by running more epochs when using 2 hidden nodes. When using 10 or more hidden nodes most learning rates proved efficient for predicting the wave, with some outliers, and from 30 hidden nodes and up most learning rates provide arguably good results. It is hard to tell which model provided the single best prediction. The PyTorch model has some very good predictions using 10 hidden nodes and a learning rate of 0.007 and using 20 hidden nodes and learning rates of 0.003 - 0.007. The Numpy model provided its best result at 30 nodes and

up, with 30 hidden nodes and a learning rate of 0.003 being very good, and 50 hidden nodes and learning rates 0.001 - 0.007 all being nearly visually identical to the validation wave. See fig. 11.

3.1.4 Summary

In summary, for sine predictions, the Numpy model performs exceptionally well, at the same levels as PyTorch. However, there is room for improvement in the tests, as it is likely that neither models's optimal parameters have been tested. In addition, each optimiser has different parameter sensitivities, which has not compensated for beyond testing a range of parameters for all optimisers. We did not manage to run any hyperparameter search on the models, because we do not have any reliable way to consistently measure the similarity between the predicted waves and the validation wave. This is also the reason why comparisons and validation of the predicted waves were done visually in the tests. See section 4.2. However, given the results, we would argue that based on the tested configurations the Numpy model performs at a level comparable to that of PyTorch, which is a feat of its own. See `FYS5429/run-sine/run_pretrained_sine.py` and its accompanying `README` in the GitHub repo for more information on running pre-trained models for sine predictions.

3.2 Natural Language Processing (NLP)

It should be possible to train an RNN on loads of text such that it can automatically complete a sequence of words (sentence). This has been our goal. To make the RNN able to complete sequences, the idea is to train it by feeding it sequences of text, where it should always predict the next word at any step in the sequence. Ideally, after training, the RNN should be able to predict which values should be output next given a primer string. As an example, given *RNNs are very good at predicting*, it should be able to output *word*. When *word* is then passed as input, it should output a punctuation mark - for instance.

The completion task means that the RNN can be trained on general text data without needing specialised datasets. Other tasks (not experimented with) could for instance be to answer questions, which would also require a dataset consisting of questions and answers. The datasets we employ in these experiments are

- (parts of) The first Harry Potter book, by J.K. Rowling
- (parts of) Romeo and Juliet, by William Shakespeare
- A self-made dataset consisting only of the string *There was a big black cat*

We begin by describing in detail how textual data is pre-processed before fed to the RNN. There are two types of experiments conducted for NLP: experiments on character-level, and experiments on word-level.

3.2.1 Text pre-processing

To train a model on NLP data, text must be converted to numbers. Two different methods for converting numbers to text are tested in the experiments: tokenisation + word-embedding, and tokenisation + one-hot encoded characters. In the former, the python library Spacy [4] is used to deal with tokenisation from string to words, and conversion from words to word embeddings. In the one-hot method, the text is split on a character level, before each character is converted to a one-hot representation.

All input data is read as a long string. Before the string is split into input and target values, it must first be split on character/word level. Whether it is split by character or word, is determined by the experiment conducted. More on these two methods for splitting strings are described below. After the string has been split, it is processed in the following way: A window of size `s` is sliding over the split string, picking out elements from 0 to `s` to be used as input and elements 1 to `s` + 1 as target, creating input and output sequences of length `s`. The window is then moved to pick out elements 1 to `s+1` for input and 2 to `s+2`. Note that *elements* are used to describe both characters *and* words, and is dependent on whether the string was split on character or word-level.

Moving the window 10 steps, generates 10 input-target sequence pairs. A visualisation of how data is created from a string representation can be seen in fig. 4. All generated input/target sequence pairs can be processed independently (in a big batch), as none of the sequences are assumed to continue another. This should be clear from thinking of what happens when the window is slid one step to the right in fig. 4. The length of the window/sequence determines how much the model should emphasize long-term relationships, where a shorter sequence length will throw away longer-term relationships (within that sequence).

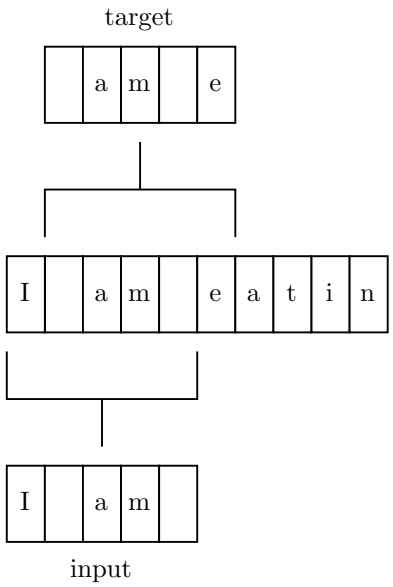


Figure 4: The visualisation shows how a window is sliding over a string, picking out sequences to be used as input and target. Note: This is for strings split on a character level. When conducting word embedding experiments, the string is rather split on a word (token) level!

Word Embedding Representation In the embedding experiments, the string is both tokenized and embedded before the windowing-process described above. The library Spacy [4] efficiently handles both tokenization and embedding of the tokens. Tokenization refers to the process of splitting a sequence of text into individual word-like parts. For example, the word *the* will be identified as a single token, whereas the contraction *We're* may be split into two tokens: *We,* and *'re*.

After tokenization, Spacy converts *input tokens* into pretrained word embeddings of length 300. For the model to output probabilities for the next word in the sequence over all possible words in the dataset, we utilize a lookup table, referred to as the *vocabulary*. This vocabulary maintains a mapping from each distinct token to

its corresponding index, and vice versa. The *target tokens* are converted into one-hot encoded vectors, where each vector has a length equal to the number of unique tokens in the dictionary. This one-hot encoding provides a way to compute a probability distribution over the possible next words. The embedded input and one-hot output approach is similar to how more advanced language models is trained.

One-hot representation For the character-level experiments, the text is split into characters. This means that the string *We're* will be split into (W , e , ' , r , e).

Also here, a *vocabulary* is created, mapping each distinct character to an index, and vice versa. The character-level representation of the string is then converted to one-hot encoded vectors, with size equal to the number of distinct characters in the dataset (often no more than 70 when considering alphabet + special signs). Both the input and target sequences are one-hot encoded.

3.2.2 NLP experiments - character level

We show two of the experiments conducted at the character level. The first experiment is to train the model to continue anywhere in the sentence "There was a big black cat.". The second is to generate lines from the characters Romeo and Juliet in the play "Romeo and Juliet" by Shakespeare.

"There was a big black cat." Input data is made from the string "There was a big black cat.", by using the sliding window technique, with a window size of 6 characters. 20 movements of the window is conducted, leaving 20 sequences, each 6 timesteps long, to train on.

Various testing on hyperparameters, such as optimiser and learning rate, found that the `Adam` optimiser using a learning rate of `0.001` performed best on average. The model was trained with a batch consisting of all 20 sequences.

To validate performance, we evaluate the generated text by discretion. It is clear that the model has learnt to generate characters, and has some notion of which characters should come after others.

The trained model performs significantly better when primed with the token *There* compared to the token *black*. This improved performance could be attributed to the fact that the characters in "There" only occur in that specific order in "There was a big black cat", while most of the characters in "black" occur several times in different orders.

The size of the memory (the hidden state) heavily influences how good the predictions are. Too many hidden nodes make the model very good at generating " was a big black cat (...)" when given "There" as a primer, but also very bad at continuing the sequence when primed with a character found in the middle of the sequence, such as "black", with target prediction "black cat (...)". Few hidden nodes only generate gibberish, while a hidden size in the range (10,50) perform mediocre for both the "There" and the "black" primer.

See **??** for a figure of loss on the different hidden size configurations, and table 1 for the text completions generated. To recreate experiments, see `simple_cat.py`.

Hidden size	**[There] as primer**	**[black] as primer**
1	[There] btalae aasgckca e atalal	[black]cai bitbiahawaha si a wkl
2	[There] wac. wie allkgc a wa bl	[black] carecacigclc tkgs la ba
3	[There] was aacg ca big aas clac	[black] a black ca bg bigre wacc
4	[There] black cack catherere was	[black]k big g blacwas clas blaa
5	[There] cack cat.re was blkwas b	[black] casig a big a big black
10	[There] was a big black cat.ack	[black] cat.a big black cat.ack
20	[There] was a big black cat.s a	[black] cat.s a big black cat.s
50	[There] was a big black cat. bb	[black] cat.a black cat.a black
100	[There] was a big black ck ck.Th	[black] cat.a black cat.rere was
200	[There] was a big blaca ca big a	[black] ca backkcackiclc ackrc c

Table 1: Predictions for experiment "There was a big black cat". Primer sequence is in square brackets, the following characters are generated.

Romeo and Juliet Input data is made from a string representation of the play Romeo and Juliet, by using the sliding window technique, with a window size of 24 characters. 5000 movements of window is conducted, leaving 5000 sequences, each 24 timesteps long.

Also here, `Adam` was found to work best, using a learning rate of `0.001`. Like in the experiment above, the model was trained with a batch consisting of all 5000 sequences. See fig. 5 for a plot of loss for different hidden size configurations. After training, The primers "Romeo" and "Juliet" are used to generate lines for

Hidden size	[ROMEO] as primer	[JULIET] as primer
1	[ROMEO],rtpyaienanfeoaczecwuu s:	[JULIET]ntp l sC.dit'seym?hpsfeo
2	[ROMEO]Qtlhiihwr aeyknvsWami auo	[JULIET] n,witrwih nWaln i Qhens
3	[ROMEO]Sinpforenn l,lhohr d lSi	[JULIET]lot atsvtea n n ade?auitc
4	[ROMEO]ccnelri wuhen: bau drpt e	[JULIET]hidewesiwo skte we, heees
5	[ROMEO]ny al alo reesa toan pyi	[JULIET]awtay uut,ire:se tige aa.
10	[ROMEO]re amy sarekalit?' and Ce	[JULIET]he yor co ore nusry ig he
20	[ROMEO]'l divene you? doer.r ert	[JULIET] guvid meaf as, wh you co
50	[ROMEO]cisufferest Ciand thed cu	[JULIET] Se the falu, well you. F
100	[ROMEO]cs emonters nom heamy,emy	[JULIET]unar ou'trell knoe to Din
200	[ROMEO]ENIUS: she haar blyy ? Wh	[JULIET]hso chars gaydirsly, mpea
300	[ROMEO]cist is ro strts, pitise	[JULIET]hea' yom, You, melb in th
400	[ROMEO]red thithey we'lr-eodeera	[JULIET]ing ourtbeor Apast cithes
500	[ROMEO], ta ghecounoufe ty utfir	[JULIET]ob esukiss crtizend. rest
1000	[ROMEO],d Fot sfomhed ynou.entor	[JULIET] tain adise wen asmy-unon

Table 2: Predictions for experiment "Romeo and Juliet". Primer sequence is in square brackets, the following characters are generated.

the two characters. The generated text can be viewed in table 2. None of the predictions made by the model are outstanding. Since there are examples of good performance on character-level for the *Romeo and Juliet* play using more advanced character-level models, see tensorflow example, it may be due to the simplicity of the model. To recreate the plots, see `romeo_and_juliet_mp.py`. To use a trained model for inference, see `romeo_and_juliet.py`.

3.2.3 NLP experiments - word level

We show one of the experiments conducted at word level, involving the first Harry Potter book. The target is to generate some text given various primers.

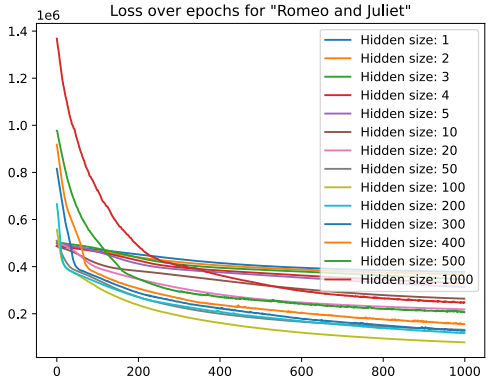


Figure 5: Loss plot for Romeo and Juliet experiment with various number of hidden nodes

Harry potter Input data is made from a string representation of the first Harry Potter book, by using the sliding window technique. 500 movements of window is conducted, leaving 500 sequences to train on. Input words are converted to word embeddings, and target words are converted to one-hot vectors. Two sequence length (window length) configurations are used: 24 and 96. Also here, `Adam` was found to work best, using a learning rate of `0.001`. The model was trained with a batch consisting of all 500 sequences. After training, several primers were tested.

For the model trained on sequences of length 24, the following output was achieved for a hidden size of 400 (primer sequence in square brackets):

[Harry was] no finer boy anywhere .The Dursleys had everything they wanted but they also had a secret and their greatest fear was that somebody would discover

[They were] the last people you d expect to be involved in anything strange or mysterious because they just did nt hold with such nonsense Lahouaiej Dursley

[Harry did not do] d expect to be involved in anything strange or mysterious because they just did nt hold with such nonsense Lahouaiej Dursley was the director of

[never] even seen him .This boy was another good reason for keeping the Potters away they did nt want Dudley mixing with a child like tha

The output for sequence length 96 is not included in the report, as they were pretty equal to the output above. Clearly, the model has learnt some intra-word relationships, as it is able to produce sentences that are

mostly grammatically correct. For few hidden nodes (1-3), the model does not produce any sensible sentences, while for many hidden nodes (20-1000), one can find exactly the sentences found in the book, indicating that

is has learnt to "remember" sentences, essentially overfitting to the training data. Hidden nodes in the range (4,10) seem to act against the sentence remembering, but does not produce grammatically sound sentences. Even though the model was trained on sequences of length 24 (and 96), the model does not seem to incorporate longer-term relationships, other than "remembering" sequences of length 24/96. Giving it words it has never seen, leads to prediction of nonsense, or at best sentences not corresponding to the primer at all.

To recreate experiments, see `harry_potter_mp.py`. To use a trained model for inference, see `harry_potter.py`. Note that the trained models with sequence length 96 is not included in the public repository, because they are too large. Some of the generated text can be found in `generated_text_for_seq_length_xx`.

Due to time constraints, and issues related to PyTorch implementation, efforts were not made to thoroughly compare the performance in natural language experiments to PyTorch and other libraries.

3.3 Training speed

We observed that the NumPy RNN trained models faster than PyTorch's RNN for models of the same size. To quantify this, a test was conducted measuring the training time for each framework to train 75 models for sine wave prediction, using a learning rate of 0.001, 40 hidden nodes, and 100 epochs. The results were clear: the NumPy implementation averaged 7.517 seconds per model, while PyTorch averaged 10.524 seconds. The total training time for 75 models was 563.837 seconds for NumPy and 789.328 seconds for PyTorch. This indicates that the NumPy implementation is, on average, 28.57% faster than PyTorch's implementation. The test was conducted on a first-generation Lenovo Yoga Slim 7 laptop with an AMD Ryzen 7 4800U CPU and 16GB of memory. The script used for testing can be found in See `training_speed.py`.

4 Discussion

4.1 Batch processing

A lot of work has gone into getting the gradient correct. The gradient was assumed to be correct, until a big drop in model performance after implementation of batch-processing was observed. It turned out that the confusion on BPTT had led to an inaccurate gradient. After implementing a non-batched version corrected for gradient mistakes, attempts were made to implement batch processing. However, this was challenging, as there were no resources on how to do this in a non-autograd way. I.e., how can the batch dimension be included in the forward and backward equations? Batching of sequential data resulted in a 3-dimensional matrix to be processed at every forward-, and backward step. The implementation now handles batches of data, which leads to a significant decrease in runtime, especially when the model is trained on HPC clusters.

4.2 Hyperparameter search

The optimal way to train our model for any case would be with hyperparameter training before conducting any of the experiments. An attempt to implement random search to do this was made but quickly shut down after realizing we had no clear way of measuring the performance of the models with anything except for loss. For sine wave predictions we tried using Mean Square Error (MSE) to find the error between the validation waves and the predicted ones. However, this resulted in unintuitive error numbers and scale. Normalizing the MSE by the max amplitude of the validation wave might have been a viable way to get an intuitive scale of error, or using Dynamic Time Warping (DTW) could also be a viable option we did not explore. The same goes for validating the results in our NLP tests; when working with one-hot characters it will not work to look at the distance between indexes of characters to estimate an error, as the index each character receives is not linked to their relationship to one another. For embeddings, several ways to measure semantic meaning exists. Due to time constraints, these were not implemented.

4.3 Performance

The model is both able to predict sine waves and natural language, through an implementation that is sufficiently fast and readable. The implementation is more than comparable top PyTorch, while at the same time being much simpler built. The code provides a strong foundation for further improvements and experiments.

5 Further work

The biggest limitation of the implemented RNN model, is that the implementation only allows for one hidden layer. Since including more hidden layers is a way to increase capacity of neural networks [2], adding more hidden layer could allow for a better model. However, training RNNs are already difficult enough with one layer, and more advanced training techniques may be required in order to prevent issues with gradient descent (BPTT).

Additionally, the model implemented does not exploit GPUs (or TPUs). Efforts were made to implement GPU acceleration using jax. However, the readability and understanding of what is happening in code vanished. It was therefore decided to not use jax and similar libraries. The gradient calculations are therefore performed through symbolic derivation instead of automatic differentiation.

A Pseudocode

Algorithm 1 Gradient Clipping Function: clip_gradient()

Require: Array of gradients to be clipped ∇
Require: Threshold value v
  for i in $0 \rightarrow |\nabla| - 1$ **do**
    $a \leftarrow \text{normalise}(\nabla_i)$
    if $a > v$ **then**
      $\nabla_i \leftarrow \nabla_i \cdot \frac{v}{a}$
    end if
  end for
  return ∇

Algorithm 2 Optimiser 1: Stochastic Gradient Descent (SGD)

Require: Parameters Θ
Require: Learning rate ϵ
  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\mathbf{\Theta}} \Sigma_i L(f(\mathbf{x}^{(i)}; \mathbf{\Theta}), \mathbf{y}^{(i)})$
  for i in $0 \rightarrow |\Theta| - 1$ **do**
    $\Theta_i \leftarrow \Theta_i + \epsilon \cdot \mathbf{g}_i$
  end for
  return Θ

Algorithm 3 Optimiser 2: SGD w/ momentum

Require: Parameters Θ
Require: Learning rate ϵ
Require: Momentum rate ν
  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\mathbf{\Theta}} \Sigma_i L(f(\mathbf{x}^{(i)}; \mathbf{\Theta}), \mathbf{y}^{(i)})$
  for i in $0 \rightarrow |\Theta| - 1$ **do**
    $\Theta_i \leftarrow \nu \Theta_i + \epsilon \mathbf{g}$
  end for
  return Θ

Algorithm 4 Optimiser 3: AdaGrad

Require: Parameters Θ
Require: Learning rate ϵ
Require: Very small constant δ
  $\alpha \leftarrow \mathbf{0} \in \mathbb{R}^{|\Theta|}$
  $\mathbf{g} \leftarrow \frac{1}{m}\nabla_{\mathbf{\Theta}}\Sigma_i L(f(\mathbf{x}^{(i)}; \mathbf{\Theta}), \mathbf{y}^{(i)})$
  for i in $0 \rightarrow |\Theta| - 1$ **do**
    $\alpha_i \leftarrow \alpha_i + \mathbf{g}_i \odot \mathbf{g}_i$
    $\Delta\Theta \leftarrow \epsilon \cdot \frac{\mathbf{g}_i}{\delta + \sqrt{\alpha_i}}$
    $\Theta_i \Theta_i + \Delta\Theta_i$
  end for
  return Θ

Algorithm 5 Optimiser 4: Adam

Require: Parameters Θ
Require: Learning rate ϵ
Require: Exponensial Decay Rates $\rho_1, \rho_2 \in [0, 1)$
Require: Very small constant δ
  $\mathbf{s} = \mathbf{0}$
  $\mathbf{r} = \mathbf{0}$
  $t = 0$
  $\mathbf{g} \leftarrow \frac{1}{m}\nabla_{\mathbf{\Theta}}\Sigma_i L(f(\mathbf{x}^{(i)}; \mathbf{\Theta}), \mathbf{y}^{(i)})$
  for i in $0 \rightarrow |\Theta|$ **do**
    $t \leftarrow t + 1$
    $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$
    $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$
    $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$
    $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$
    $\Delta\mathbf{\Theta} = \epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$
    $\mathbf{\Theta} \leftarrow \mathbf{\Theta} + \mathbf{\Delta\Theta}$
  end for

Algorithm 6 Forward pass: _forward()

Require: Sequence $\mathbf{x}$
Require: Parameters $\Theta = \{\mathbf{U}, \mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}\}$
Require: Activation function $f(\cdot)$
Require: Postprocessing function $g(\cdot)$
  for t in $0 \rightarrow \tau - 1$ **do**
    $\mathbf{a}^{(t)} \leftarrow \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$
    $\mathbf{h}^{(t)} \leftarrow f(\mathbf{a}^{(t)})$
    $\mathbf{o}^{(t)} \leftarrow \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$
    $\hat{\mathbf{y}}^{(t)} \leftarrow g(\mathbf{o}^{(t)})$
  end for
  return $\hat{y}$

Algorithm 7 Backward pass funciton: _backward()

Require: Optimiser- and gradient clipping functions
Require: Input and hidden state tables
Require: Parameters $\Theta = \{\mathbf{U}, \mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}\}$
Require: Activation function $f(\cdot)$
Require: Loss function

$\nabla_{\mathbf{V}} C \leftarrow \mathbf{0} \in \mathbb{R}^{K \times M}$
$\nabla_{\mathbf{W}} C \leftarrow \mathbf{0} \in \mathbb{R}^{M \times M}$
$\nabla_{\mathbf{U}} C \leftarrow \mathbf{0} \in \mathbb{R}^{M \times N}$
$\nabla_{\mathbf{c}} C \leftarrow \mathbf{0} \in \mathbb{R}^{K}$
$\nabla_{\mathbf{b}} C \leftarrow \mathbf{0} \in \mathbb{R}^{M}$
$\nabla_{\mathbf{o}} C \leftarrow$ loss.grad()
for t in $\tau \rightarrow 0$ **do**

$\nabla_{\mathbf{h}^{(\tau)}} C \leftarrow \nabla_{\mathbf{h}^{(t)}} + \mathbf{V}^T \cdot \nabla_{\mathbf{o}^{(t)}} C$

$\nabla_f \leftarrow \frac{\delta f}{\delta \mathbf{h}^{(t)}}$

$\nabla_{\mathbf{V}^{(t)}} C \leftarrow \nabla_{\mathbf{V}^{(t+1)}} C + \mathbf{h}^{(t)} \cdot \nabla_{\mathbf{o}^{(t)}} C$

$\nabla_{\mathbf{W}^{(t)}} C \leftarrow \nabla_{\mathbf{W}^{(t+1)}} C + \nabla_f \cdot \mathbf{h}^{(t-1)} \cdot \nabla_{\mathbf{h}^{(t)}} C$

$\nabla_{\mathbf{U}^{(t)}} C \leftarrow \nabla_{\mathbf{U}^{(t+1)}} C + \nabla_f \cdot \mathbf{x}^{(t)} \cdot \nabla_{\mathbf{h}^{(\tau)}} C$

$\nabla_{\mathbf{c}^{(t)}} C \leftarrow \nabla_{\mathbf{c}^{(t+1)}} C + \nabla_{\mathbf{o}^{(t)}} C \cdot \mathbf{1}$

$\nabla_{\mathbf{b}}^{(t)} C \leftarrow \nabla_{\mathbf{b}^{(t+1)}} C + \nabla_f \cdot \nabla_{\mathbf{h}^{(\tau)}} C$

$\nabla_{\mathbf{h}^{(t)}} C \leftarrow \nabla_f \cdot \mathbf{W}^T \cdot \mathbf{h}^{(t)} C$
end for
$\nabla \leftarrow \{\nabla_{\mathbf{V}} C, \nabla_{\mathbf{W}} C, \nabla_{\mathbf{U}} C, \nabla_{\mathbf{c}} C, \nabla_{\mathbf{b}} C\}$
$\overline{\nabla} \leftarrow$ clip_gradient(∇)
$\Theta \leftarrow \Theta -$ optimiser($\overline{\nabla}$)

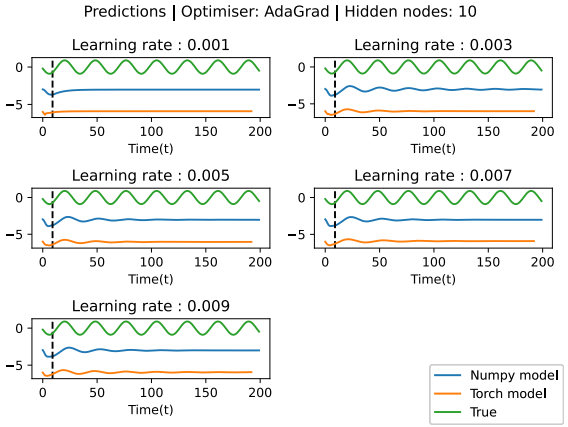
Algorithm 8 Training function: fit()

Require: Input sequence X
Require: Set of labels $\mathbf{y}$
Require: Number of training iterations N_{epochs}
Require: Initialisation functions for weights and hidden states.

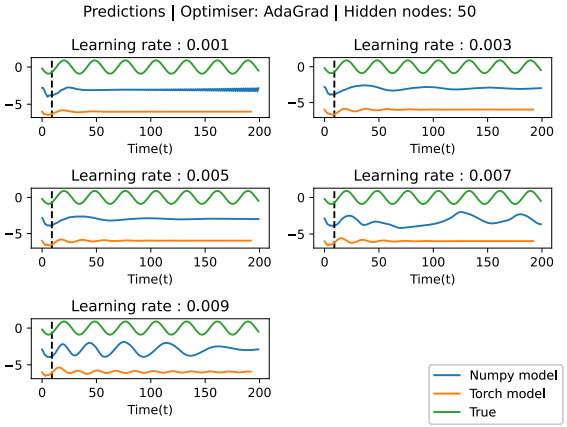
initialise_weights()
for e in $0 \rightarrow N_{\text{epochs}}$ **do**
  for every entry (x, y) in the table {X, Y} **do**
    $N_{\text{hidden\_states}} \leftarrow |x|$
    initalise_states()
    $\hat{y} \leftarrow$ forward(x)
    loss = loss(y, $\hat{y}$)
    backward()
  end for
end for

B Plots of Sine Wave Predictions When Trained on Random Sine Waves

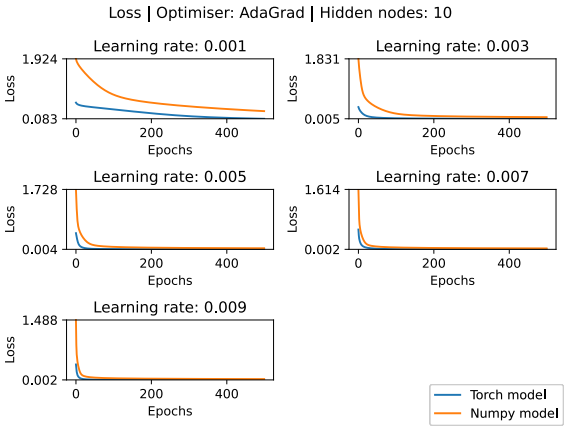
B.1 Results for AdaGrad Optimiser



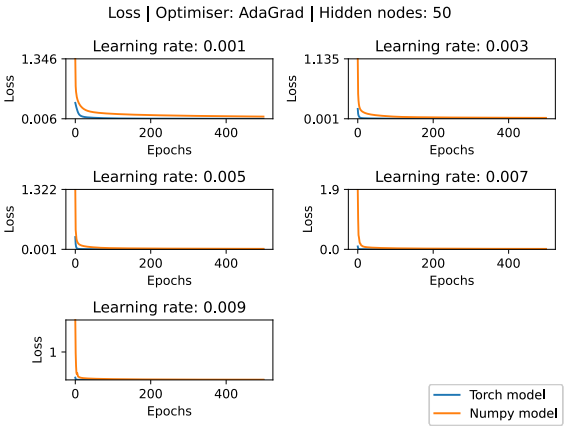
(a) The best result produced by the PyTorch model can be seen in this plot using 10 hidden nodes and a learning rate of 0.009



(b) The best result produced by the Numpy model can be seen in this plot using 50 hidden nodes and a learning rate of 0.009



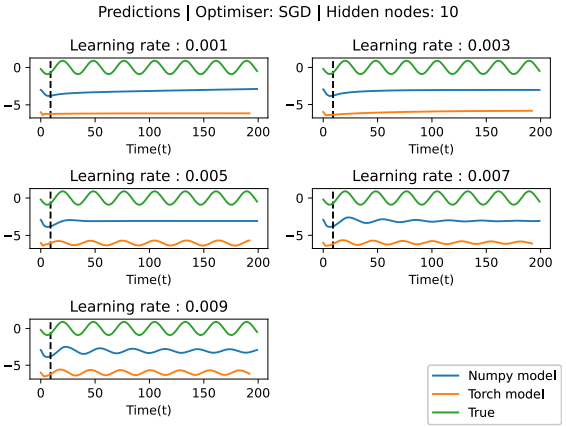
(c) The training loss for the PyTorch and Numpy models when getting the best PyTorch result



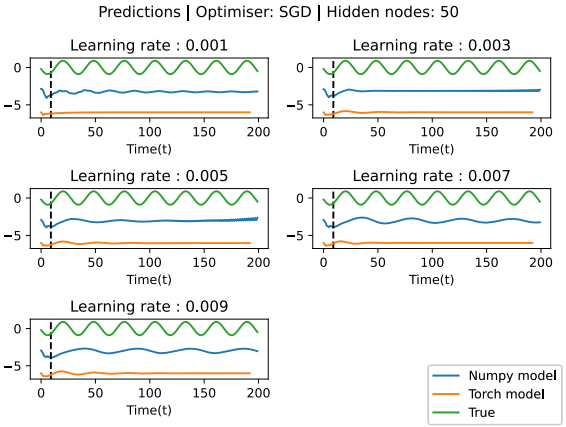
(d) The training loss for the PyTorch and Numpy models when getting the best Numpy result

Figure 6: Results of sine wave predictions using the AdaGrad optimiser with different hyperparameters

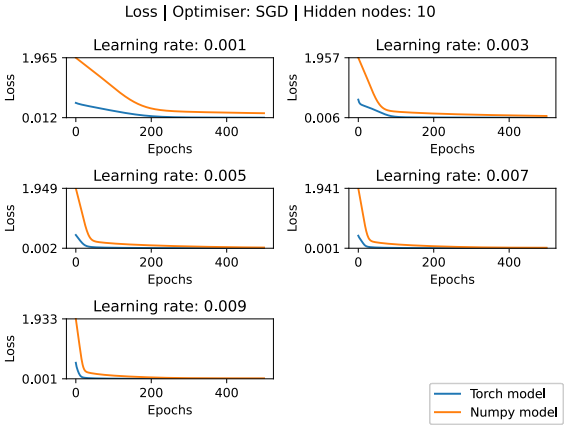
B.2 Results for Stochastic Gradient Descent (SGD) Optimiser



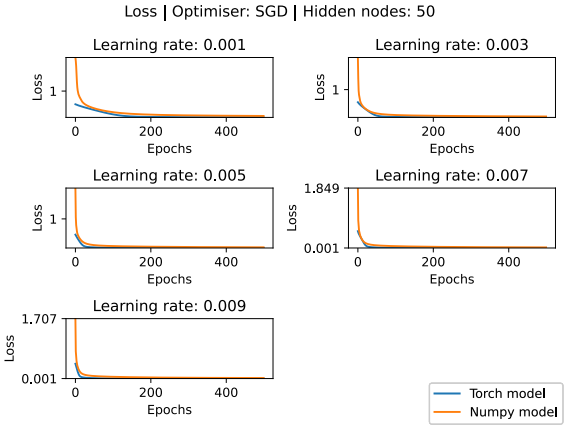
(a) The best result produced by the PyTorch model can be seen in this plot using 10 hidden nodes and a learning rate of 0.009, and the best result produced by the Numpy model can be seen in the same plot



(b) The Numpy model also produced some ok results using 50 hidden nodes, with either 0.007 or 0.009 as learning rates, however, the frequency is quite off compared to the validation wave



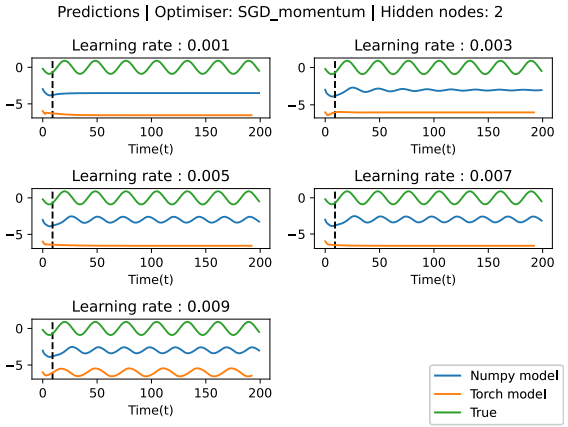
(c) The training loss for the PyTorch and Numpy models when getting the best result for both models]



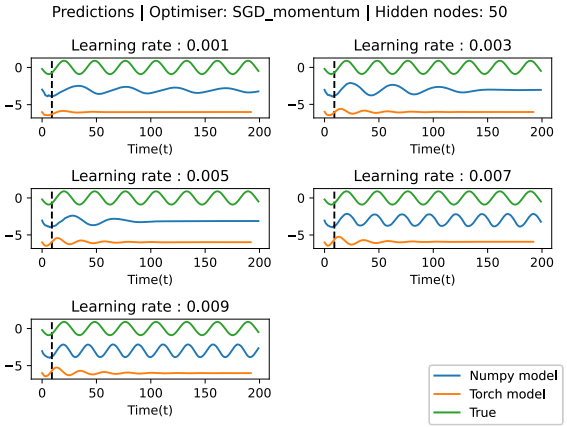
(d) The training loss for the PyTorch and Numpy models when getting good results from the Numpy model

Figure 7: Results of sine wave predictions using the SGD optimiser with different hyperparameters

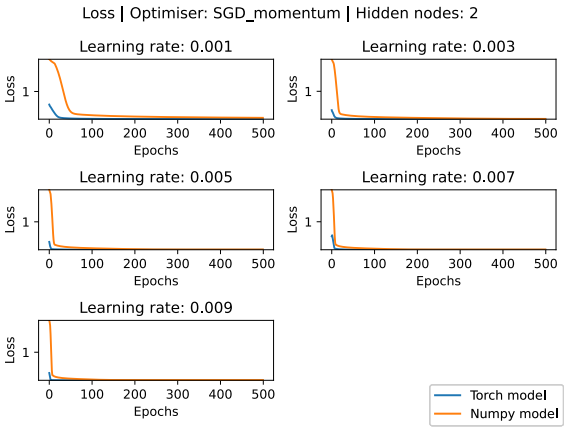
B.3 Results for SGD w/ Momentum Optimiser



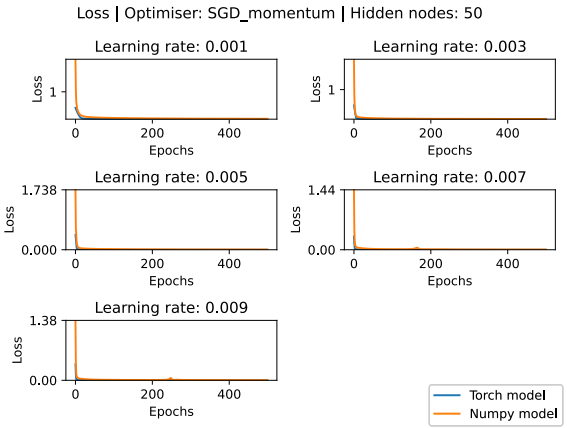
(a) The best result produced by the PyTorch model can be seen in this plot using 2 hidden nodes and a learning rate of 0.009



(b) The best result produced by the Numpy model can be seen in this plot using 50 hidden nodes and a learning rate of 0.007



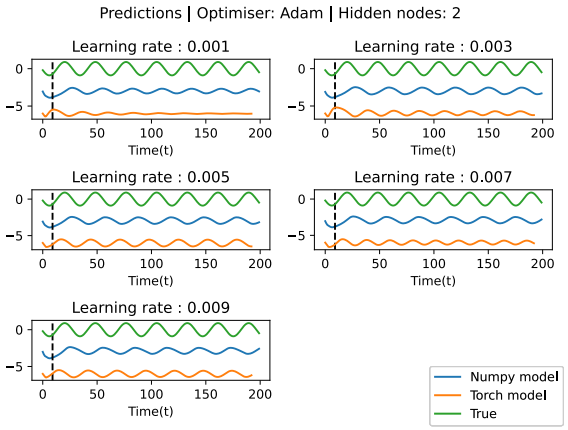
(c) The training loss for the PyTorch and Numpy models when only having 2 hidden nodes



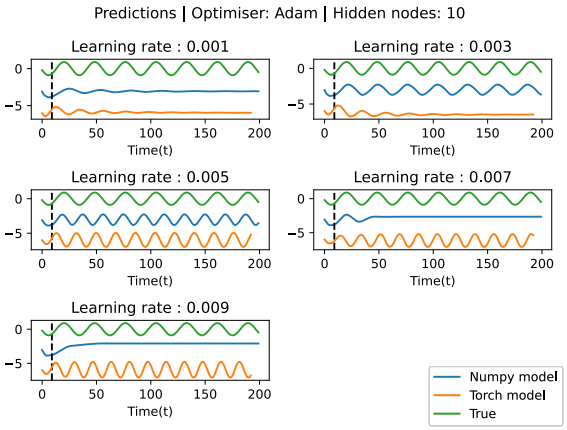
(d) The training loss for the PyTorch and Numpy models when having 50 hidden nodes

Figure 8: Results of sine wave predictions using the SGD w/ momentum optimiser.

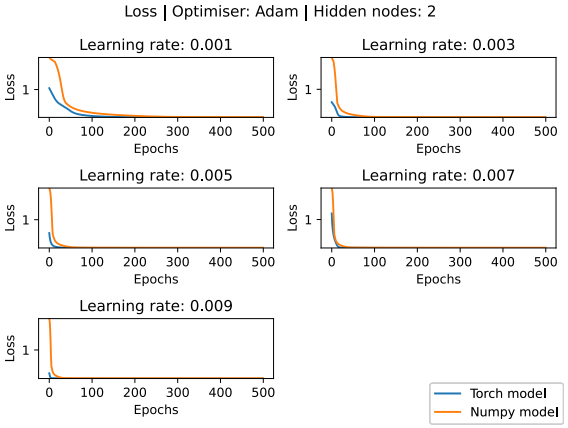
B.4 Results for Adam Optimser



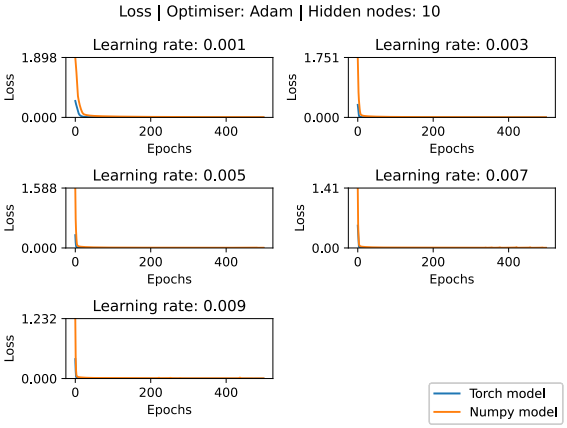
(a) The best result produced by the PyTorch model can be seen in this plot using 2 hidden nodes and a learning rate of 0.005



(b) The best result produced by the Numpy model can be seen in this plot using 10 hidden nodes and a learning rate of 0.003



(c) The training loss for the PyTorch and Numpy models when getting the best PyTorch result



(d) The training loss for the PyTorch and Numpy models when getting the best Numpy result

Figure 9: Results of sine wave predictions using Adam optimiser algorithm.

C Plots of Noise Reduction Results

Figure 10: Noise reduction of a sine-wave using different combinations of hidden layer sizes and learning rates with Adam for both the Numpy model and PyTorch models. Both models predict similar waves, but we can see the PyTorch model achieves the best results given the right configurations. The Numpy model seems to struggle with a phase shift problem in many cases.

C.1 Plots of Sine Wave Predictions When Training on the Same Sine Wave

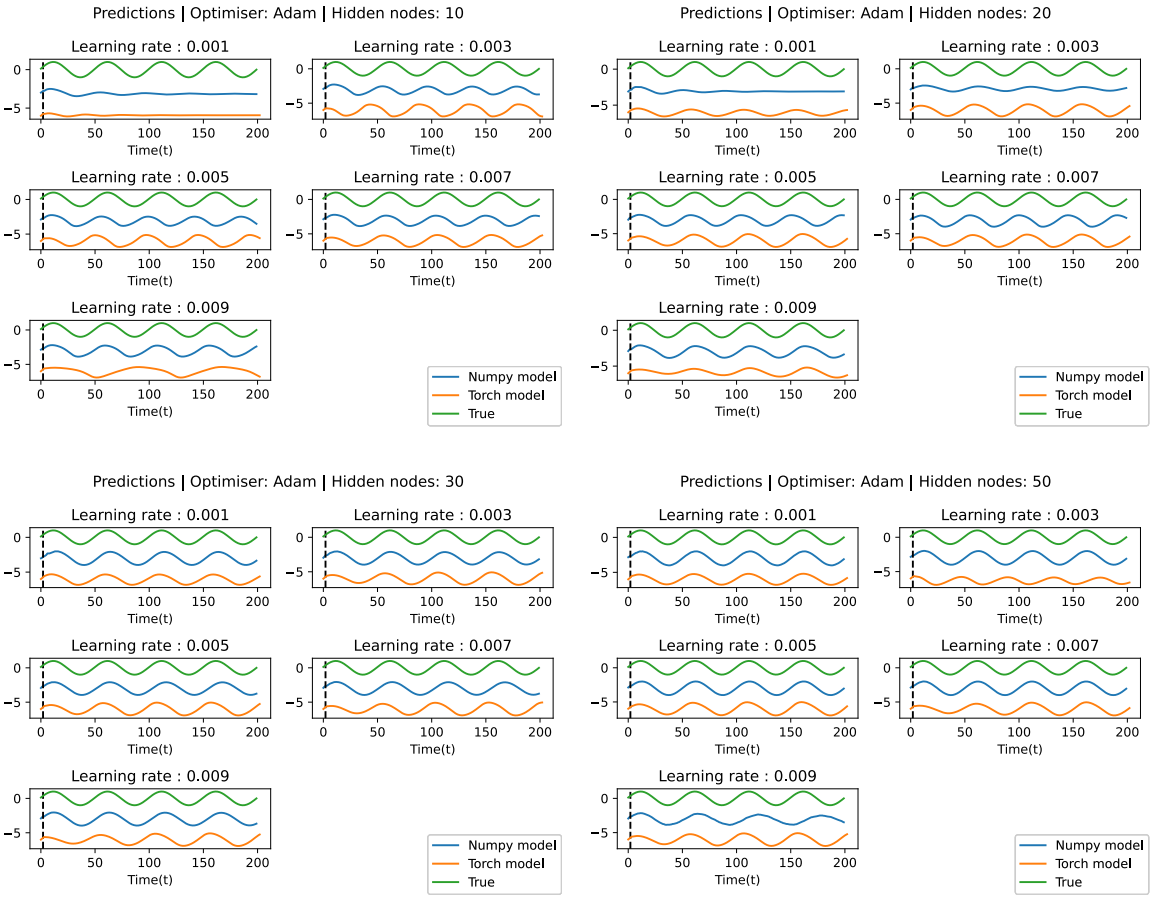


Figure 11: Predictions of a sine wave with the models only trained on the same wave for 200 epochs and 3 seed values. We can see that 10-, and 20 hidden nodes still struggle a bit with 0.001 as the learning rate, but that with more hidden nodes or a higher learning rate, we achieve predictions closely resembling the validation wave in several different configurations

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," en, *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/323533a0. [Online]. Available: https://www.nature.com/articles/323533a0 (visited on 05/31/2024).

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning* (Adaptive computation and machine learning), eng. Cambridge, Mass: The MIT press, 2016, ISBN: 978-0-262-03561-3.

[3] A. Paszke, S. Gross, F. Massa, *et al.*, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, arXiv:1912.01703 [cs, stat], Dec. 2019. DOI: 10.48550/arXiv.1912.01703. [Online]. Available: http://arxiv.org/abs/1912.01703 (visited on 05/31/2024).

[4] I. Montani, M. Honnibal, M. Honnibal, A. Boyd, S. V. Landeghem, and H. Peters, *Explosion/spaCy: V3.7.2: Fixes for APIs and requirements*, Oct. 2023. DOI: 10.5281/ZENODO.1212303. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.1212303 (visited on 03/29/2024).

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," en, *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://www.nature.com/articles/s41586-020-2649-2 (visited on 06/04/2024).

[6] Y. E. Wang, G.-Y. Wei, and D. Brooks, *Benchmarking TPU, GPU, and CPU Platforms for Deep Learning*, arXiv:1907.10701 [cs, stat], Oct. 2019. [Online]. Available: http://arxiv.org/abs/1907.10701 (visited on 03/29/2024).

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, *Scikit-learn: Machine Learning in Python*, arXiv:1201.0490 [cs], Jun. 2018. DOI: 10.48550/arXiv.1201.0490. [Online]. Available: http://arxiv.org/abs/1201.0490 (visited on 06/04/2024).

[8] R. Pascanu, T. Mikolov, and Y. Bengio, *On the difficulty of training Recurrent Neural Networks*, arXiv:1211.5063 [cs], Feb. 2013. DOI: 10.48550/arXiv.1211.5063. [Online]. Available: http://arxiv.org/abs/1211.5063 (visited on 04/01/2024).

[9] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar. 1994, Conference Name: IEEE Transactions on Neural Networks, ISSN: 1941-0093. DOI: 10.1109/72.279181. [Online]. Available: https://ieeexplore.ieee.org/document/279181 (visited on 06/07/2024).

[10] R. J. Williams and D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, Jun. 1989, Conference Name: Neural Computation, ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.2.270. [Online]. Available: https://ieeexplore.ieee.org/document/6795228?ref=floydhub.ghost.io (visited on 06/07/2024).

[11] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980 [cs], Jan. 2017. DOI: 10.48550/arXiv.1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980 (visited on 03/21/2024).