Effects of an LSTM Composite Prefetcher

Joseph Rogers
Abstract

Effects of an LSTM Composite Prefetcher

Joseph Rogers

Recent work in computer architecture and machine learning has seen various groups begin exploring the viability of using neural networks to augment conventional processor designs. Of particular interest is using the predictive capabilities of techniques in natural language processing to assist traditional CPU memory prefetching methods. This work demonstrates one of these proposed techniques, and examines some of the challenges associated with producing satisfactory and consistently reproducible results. Special attention is given to data acquisition and preprocessing as different methods. This is important since the handling training data can enormously influence on the final prediction accuracy of the network. Finally, a number of changes to improve these methods are suggested. These include ways to raise accuracy, reduce network overhead, and to improve the consistency of results. This work shows that augmenting an LSTM prefetcher with a simple stream prefetcher leads to moderate improvements in prediction accuracy. This could be a way to start reducing the size of neural networks so they are usable in real hardware.

Handledare: David Black-Scheffer
Ämnesgranskare: Justin Pearson
Examinator: Mats Daniels
IT 19 047
Tryckt av: Reprocentralen ITC
Glossary

Address Space Layout Randomization – ASLR
Long Short-Term Memory – LSTM
Natural Language Processing – NLP
Program Counter – PC
Recurrent Neural Network – RNN
Standard Performance Evaluation Corporation – SPEC
1 Introduction

In many CPUs produced today, memory prefetchers are used to speed up execution by fetching data from slower distant memory into faster local memory, before the data is actually needed by the program. Over time there have been steady gains in the compute capabilities of CPUs, but comparatively slow gains in both memory speed and latency [1]. It is therefore increasingly desirable to develop new methods that can prefetch as accurately and efficiently as possible.

On the 6th of March 2018, members of a research team at Google published a paper studying the abilities Long Short-Term Memory (LSTM) neural networks to make predictions similar to those that are required of CPU memory prefetchers [2]. This research established two main principles. Firstly, it was shown conclusively that LSTMs can indeed make similar predictions to those of a typical CPU prefetcher, and in many cases at a higher level of accuracy. Secondly however, it was shown that the sizes of the neural networks necessary to make these improved predictions were far too large to be considered for hardware implementation.

In this project we have created an LSTM prefetcher similar in design to one presented by the Google research team. We have also investigated the effects of augmenting the neural network with a simple stream prefetcher to create a two part composite design. This was done so as to assist the prefetcher in the case of relatively basic predictions. The hope was that this assistance could eventually allow for smaller neural networks to be used. This was important as overly large network sizes are one of the major issues preventing hardware implementations in the case of prefetchers.

The LSTM based prefetcher for this project was created in software. The LSTM model was trained in an offline scenario using data from a cache simulator built with Intel’s PIN instrumentation software [3]. We have examined the accuracy first of the base prefetcher, and then of the composite design. Prediction accuracy was measured on a series of microbenchmarks, as well as on a set of select programs from the popular Standard Performance Evaluation Corporation (SPEC) 2017 CPU benchmark suite. This project did not investigate actually shrinking the network size of the LSTM.

Results show that the addition of a simple stream prefetcher to the design allows for moderate improvements in overall prediction accuracy. These improvements occur because the stream prefetcher allows the neural network to focus on learning a smaller number of access patterns. This work was carried out under the supervision of Professor David Black-Schaffer, and with assistance from Hassan Muhammad, both of Uppsala University. Additional input was provided by Professor Rakesh Kumar of The Norwegian University of Science and Technology.
2 Background

2.1 CPU Memory Prefetchers

There are numerous different types of designs for CPU memory prefetchers. Many rely on some kind of heuristics or history, such as temporary information about past access activity, which is then stored in a table. This project aims to further expand research that seeks to move beyond table based approaches, and to start examining the possibilities of building a prefetcher based on neural networks. Other work has already been done showing that different kinds of LSTM based prefetchers can have extremely high prediction accuracies on a variety of standard CPU performance benchmarks [2]. However, more work needs to be done on how to reduce the memory footprint of these networks, before they can be implemented in hardware.

2.2 Neural Networks in Computer Architecture

Neural networks being used in computer architecture is not a new idea. Researchers Daniel A. Jimenez and Calvin Lin from The University of Texas at Austin proposed the idea for using simple perceptrons, the basic building blocks for all neural networks, as branch predictors in January of 2001 [4]. This idea has taken root in industry as well, with processors such as the Samsung Exynos M1 using a simple neural network for its branch predictor [5].

2.3 Recurrent Neural Networks (RNNs)

Many of the simplest neural network are feed-forward networks. These networks are those in which data flows in only one direction. Normally this would be from a set of input nodes, which then connect to a set of what are called hidden layer nodes. These hidden nodes then connect to a set of output nodes. A basic network architecture demonstrating this setup is shown in Figure 1.

![Figure 1: A simple feed-forward neural network. Data only flows in one direction. Data goes from the input layer, to the hidden layer, to the output layer.](image)

Often, feed-forward networks are used for tasks where the ordering of data that is input to the network is not important. A good example of this would be image classification. Typically, for these types of problems, one item in the database of images does not have any relevant temporal...
relationship to any other item in the database. This is one reason why image classification networks often randomize the order of training data fed to the network each epoch. This is done so as refrain from inferring a non-existent relationship based on the ordering of images.

By contrast, recurrent neural networks (RNNs) are typically used for problems where different items in a dataset do have an important temporal relationship. With RNNs, nodes in the hidden layer receive information not only from the input layer, but also from themselves. RNNs, save information about what they have seen in the past, and use it to make contextual judgments about what they are seeing in the present. Information can be shared across many time-steps. Visualizing how information is passed time-step to time-step is called unfolding the RNN. An example of this is shown in Figure 2.

![Unfolded RNN Diagram](image)

Figure 2: A simple recurrent neural network unfolded through time. $x$ represents input data being sent to a hidden layer $h$. On a new time-step $h_t$, information $V$ about the past is received from the $h_{t-1}$ layer on the previous time-step.

While the ability to share information across time-steps is very useful for being able to contextualize data being seen in the present, it has practical limitations. Neural networks are typically trained through backpropagation, where the errors in the weights of each node in the output layer are propagated backwards through the network towards the input layer. This process allows the weight values in each node to be tweaked so that the network can make better predictions in the future \[6\]. For RNNs, backpropagation also happens backwards through time. After the node weights are adjusted for $h_t$, the calculated errors in those weights will then be used to adjust $h_{t-1}$. The errors found in $h_{t-1}$ will then be used to adjust $h_{t-2}$, and so on.

The problem is that this can lead to an increasingly long chain of floating point multiplications, with values between 0 and 1. This series of multiplications will eventually drive calculated error values close enough to 0 that typical floating point resolution will no longer be sufficient to accurately track the numbers. This puts a practical limit on how many time-steps can be kept track of when using RNNs. This issue of shrinking errors is known as the vanishing gradient problem \[7\]. Problems with vanishing gradients can be seen with as few as 100 time-steps being used. Many problems require a much large number of time-steps, often many hundreds or even thousands of steps long. For this to be possible a way to address the vanishing gradient is needed.
2.4 Long Short–Term Memory (LSTM)

In 1997, Long Short–Term Memory was proposed by Hochreiter and Schmidhuber as a direct way to address the vanishing gradient problem faced by RNNs [8]. LSTMs are a direct evolution of RNNs, and are used in many networks today.

LSTMs cells have what is referred to as gated memory. Whereas before the memory contained in RNNs flowed unregulated time–step to time–step, LSTMs have series of functions, also called gates, that the data must go through in order to regulate its importance. Modern LSTMs often have three such gates. These gates are referred to as the forget gate $F$, the input gate $I$, and the output gate $O$. Each of these gates is itself a simple neural network, and automatically gets tuned as part of the overall training. The LSTM memory is referred to as the cell state $C$.

The forget gate decides what information to throw away from $C$ at the current time–step. The input gate decides what new information to add to $C$. The output gate simply decides what information output from the LSTM cell at each time–step. A detailed overview of the LSTM cell is shown in Figure 3.

![LSTM unit diagram](image)

Figure 3: Overview of an LSTM cell. Data from both the input layer $x_t$ and previous time–step $h_{t-1}$ pass through the forget gate $F_t$, the input gate $I_t$, and the output gate $O_t$. The cell states, $C_{t-1}$ and $C_t$, flow along the top of the LSTM cell, and act as its memory.

The most important part of LSTMs is the cell state $C$. The cell state allows information to flow freely between LSTM cells for thousands or even millions of time–steps. Over time, information is changed only by removal via the forget gate $F$, or addition via the input $I$. This allows calculated errors to be carried back much further through time, removing the problem of the vanishing gradient.

The benefits that LSTMs have over traditional RNNs has made them extremely popular in fields such as Natural Language Processing (NLP), where long sequences of data are used. LSTMs have proven to be very effective for tasks such as speech recognition [9], and text to text language translation [10]. This make them a ideal choice for creating a prefetcher, since prefecchers often also need to operate on large time scales, while handling many types of memory accesses.
2.5 LSTMs as Prefetchers

Predicting cache misses can be seen as a sequence based problem that could be accurately solved using LSTMs. Common programming constructs such as loops lead to repeated access patterns across various data structures. Function calls create dependencies between memory locations that can span many thousands of instruction cycles. Being able to recognize these repeated patterns, and to keep track of long-term dependencies are one set of reasons why LSTMs can effectively model cache misses.

It has been shown in [2] and [11] that LSTMs can have great success when it comes to prediction accuracy, but that much work still needs to be done about the size of the networks themselves. LSTM networks of the kind used in large scale NLP problems can comprise of millions of nodes, such as in [2]. Each of these nodes has associated weight values that must be stored. The memory footprint of these values is one of the major problems that needs to be addressed in order for neural networks to become more practical when being implemented as part of CPU hardware.
3 Methods

3.1 Cache Simulation

In order to train the neural network, we first had to acquire suitable training data. We wanted our cache misses to model LLC misses. To that end, we used a single-level in-order cache simulator to model the different activities that would happen in a real hardware scenario. The cache simulator was written by Gregory Vaumourin at Uppsala University using Intel PIN. Modifications to the cache simulator for the purposes of this project were done by Hassan Muhammad.

The cache simulator allowed us to profile which instructions in each of our programs caused misses to occur, as well as the virtual addresses of each miss. Using a cache simulator allowed us to directly control the size, associativity, and replacement policy of the cache. We only started gathering data after a certain number of instructions had already occurred. This was an important detail because we wanted to be sure that we were gathering data on misses that were not from the very start of execution. Profiling from the very start would gather unimportant misses related to initial program startup patterns, as opposed to its main content. Furthermore, starting part way into execution ensured that the cache was full when we started gathering data.

We gathered data for each of our programs using the same cache settings. Our chosen parameters are listed in Table 1.

<table>
<thead>
<tr>
<th>Cache Simulator Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>2MB</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>64B</td>
</tr>
<tr>
<td>Associativity</td>
<td>8</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>Initial Instructions Skipped</td>
<td>10 Billion</td>
</tr>
<tr>
<td>Instruction Window</td>
<td>100 Million</td>
</tr>
</tbody>
</table>

Table 1: Cache settings used for the cache simulator.

All programs were simulated over a 100 million instruction window. However, only the misses that occurred during this window were output as data to be used for training. For each cache miss the simulator wrote out a two element vector containing the program counter (PC) and virtual address at which the cache miss occurred. An example of this output is shown in Table 2.

While in actuality many programs can execute for billions, or even trillions of instructions, gathering data on this scale was frequently found to take as long as a week to simulate, and generated output files with sizes in the hundreds of gigabytes. As a result, an instruction window of 100 million was chosen in order to gather sufficient amounts of training data, while not being too resource intensive to generate.

3.2 Simulated Programs

We gathered traces from seven different programs during the course of this project. The first three were basic microbenchmarks. The final four were from the SPEC 2017 benchmark suite.
### Table 2: Example output from the cache simulator. Output is a list of PCs and virtual addresses of each cache miss that occurs during program execution. The PC is unchanging because the program is currently in a loop.

<table>
<thead>
<tr>
<th>PC</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400619</td>
<td>0x7EFEE09AC4A8</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFEE09AC4C0</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFEE09AC500</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFEE09AC540</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFEE09AC580</td>
</tr>
</tbody>
</table>

The three microbenchmarks we wrote were made to verify that our cache simulator and neural network were both working as intended. These microbenchmarks were not meant to be representative of the prefetcher’s expected performance in a real world scenario.

#### 3.2.1 Microbenchmark 1

Our first microbenchmark was designed to create an extremely simple load pattern. This program stepped through an array many times at a constant stride, and added up all the values. We wanted to have one program with the simplest possible load pattern, something that the LSTM should have no trouble at all predicting. This would enable us to verify that the basics of the LSTM prefetcher were actually setup and working correctly.

```c
// Sum the values from an array.
for (int i = 0; i < 128; ++i){
    for (int j = 0; j < data_size/sizeof(int); ++j){
        result += data[j];
    }
}
```

#### 3.2.2 Microbenchmark 2

Our second microbenchmark is nearly the same as the first, but with two arrays instead of one. While both arrays were still being read at a constant stride, the two reading strides were not the same. It was expected that the LSTM prefetcher should still be able to perform very well on this program, but perhaps not quite as well as on microbenchmark 1.

```c
// Sum the values from two arrays with different strides.
for (int i = 0; i < 128; ++i){
    for (int j = 0; j < data_size/sizeof(int); ++j){
        result += data1[j] + data2[j*4];
    }
}
```

#### 3.2.3 Microbenchmark 3

The third and final of our microbenchmarks was designed to be substantially more challenging for the network to learn. Unlike the other two microbenchmarks, one of the arrays in this program had its stride change on every single load. The stride was set to be the square of the loop iteration. It was then modulated by the size of the array, so as to not access out of bounds.
```c
// Sum the values from two arrays, but with a constantly changing stride.
for (int i = 0; i < 512; ++i){
    for (int j = 0; j < data_size/sizeof(int); ++j){
        result += data1[j] + data2[((j*j) % data_size/sizeof(int))];
    }
}
```

3.2.4 Standard Performance Evaluation Corporation (SPEC) 2017 Benchmarks

In addition to training and measuring the performance of our prefetcher on simple self-written microbenchmarks, we also wanted to experiment with more complicated programs. We chose to use a number of programs from the SPEC CPU® 2017 benchmark suite. Of the forty-three benchmarks offered by SPEC 2017, we chose four. Using our cache simulator we gathered data from the 600.perlbench_s, 603.bwaves_s, 605.mcf_s, and 607.cactuBSSN_s benchmarks. These programs were chosen because they represent a mixture of both integer and floating point arithmetic. Additionally, some of these benchmarks had already been used to test other LSTM-based prefetchers, such as in [2]. Knowing this, we wanted to be able to directly compare our results with theirs.

3.3 Data Preprocessing

After data was gathered using the cache simulator, it was necessary to preprocess it before using it for training with the neural network. On high level, our preprocessing can be broken into three key steps.

1. Create dataset labels.
2. Throw out irrelevant data.
3. Normalize inputs.

Each preprocessing step involved a number of challenges. It was initially unclear what types of features and labels would yield the best results when training the network. This led to variety of different techniques being tried.

3.3.1 Step 1: Using Address Deltas to Create Dataset Labels

We chose to treat predicting the addresses of cache misses as a classification problem, similar to word prediction in NLP. There is a problem with this however. A modern system has \(2^{64}\) possible memory locations, and classification networks typically output vectors of length equal to that of the class vocabulary size. Building a neural network with a vocabulary size as large as \(2^{64}\) is currently completely impracticable because of limited memory resources. A vector of this length would take approximately 18 exabytes to store. Due to this, we chose to predict address deltas, as opposed to the raw addresses themselves. An address delta at a timestep \(N\) is defined as the address at timestep \(N + 1\) minus the address at timestep \(N\).

\[
\text{Delta}_N = \text{Addr}_{N+1} - \text{Addr}_N
\]

Using address deltas significantly cuts down on the number of possible classes that our network must predict. Some of these differences are shown in Table 3. The previous equation can be used to convert virtual addresses output by the cache simulator into address deltas. An example of this is shown in Table 4.

10
### Dataset

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Unique Addresses Misses</th>
<th># Unique Deltas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microbenchmark 1</td>
<td>262K</td>
<td>3</td>
</tr>
<tr>
<td>Microbenchmark 2</td>
<td>900K</td>
<td>360K</td>
</tr>
<tr>
<td>Microbenchmark 3</td>
<td>840K</td>
<td>604K</td>
</tr>
<tr>
<td>600.perlbench_s</td>
<td>355K</td>
<td>251K</td>
</tr>
<tr>
<td>603.bwaves_s</td>
<td>898K</td>
<td>164K</td>
</tr>
<tr>
<td>605.mcf_s</td>
<td>125K</td>
<td>29K</td>
</tr>
<tr>
<td>607.cactuBSSN_s</td>
<td>300K</td>
<td>250</td>
</tr>
</tbody>
</table>

Table 3: Number of unique raw addresses locations where misses occurred in the cache vs their address deltas. Data is from a 100 million instruction window for each of the benchmarks.

<table>
<thead>
<tr>
<th>PC</th>
<th>Virtual Address</th>
<th>Address Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400619</td>
<td>0x7EFE09AC4A8</td>
<td>0x18</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFE09AC4C0</td>
<td>0x40</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFE09AC500</td>
<td>0x40</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFE09AC540</td>
<td>0x40</td>
</tr>
<tr>
<td>0x400619</td>
<td>0x7EFE09AC580</td>
<td>0x40</td>
</tr>
</tbody>
</table>

Table 4: Virtual addresses and their converted address deltas. Repeated address deltas shows that the program is accessing data at a strided interval.

#### 3.3.2 Step 2: Throwing Out Irrelevant Data via Filtering by Deltas

As shown in Table 3, even when choosing to use address deltas, the number of different unique deltas present in our programs was sometimes only slightly smaller than the number of unique addresses. Most of the time this meant that class size was still too large to be easily worked with.

Though individual programs may have a sizable number of unique address deltas, it was very often the case that only a small subset of those deltas were reused many times, while most only occurred very rarely. This was due to regular strided access patterns accounting for the vast majority of loads in most of the programs. An example of this preference for certain deltas can be seen in Figure 4.

![Distribution of Address Deltas, 603.bwaves_s](image)

Figure 4: Distribution of address deltas for a 100 million instruction window from the 603.bwaves_s program. This figure shows that of all the deltas present in the dataset, a subset of them occur several orders of magnitude more often than all the others.
Recognizing this program behavior meant that it was possible to greatly reduce the class vocabulary size, while still accounting for the vast majority of cache misses. We chose to use a max class vocabulary size of 10,000. Using a size of 10,000 allowed us to create a network that was small enough to be worked with, while still accounting for more than 90 percent of all cache misses in each of our programs. A class size as small as only 100 would still have been able to account for at least 50 percent of the cache misses in each program.

With these ideas in mind, filtering of deltas can now be summarized in two main steps. First, for each dataset generate a list of the top 10,000 most often occurring deltas. Second, for each element in the dataset, check if its label is in the top 10,000 list. If it is not in the list then delete that entry from the dataset. Filtering dataset delta yields massive memory savings, while only sacrificing a small amount of accuracy.

3.3.3 Step 3: Normalize Inputs, The Problem with Normalizing Addresses

It is often beneficial when working with neural networks to normalize the different input features to be on the same scale [12]. This is done primarily so that each of the different input features is given equal preference when training the network. If different input features are on vastly different scales from each other, then the network may end up ignoring one of them during training, and will generally under-perform as a result.

Currently, the input features in for the neural network are [PC, Virtual Address] vectors. The PCs in each program are typically densely clustered. This means that normalizing the PCs is straightforward, and not an issue. Challenges arise however when attempting to normalize the virtual addresses. As mentioned previously, a typical computer will have a 64-bit address space. There are therefore $2^{64}$ different possible address values. When trying to normalize values on a scale as large as this there are problems with floating point resolution. Normalized values for neural networks are often stored as 32-bit single precision floats. This level of precision is inadequate for trying to accurately normalize specific addresses across such a large region. More importantly however is the sparsity of address space itself. When analyzing our simulation data it was often observed that different areas of activity within the address space would be separated by large regions of empty space. This can be observed in Figure 5.

Consider the address space of the cactuBSSN_s dataset as shown in Figure 5. If normalization happens across the whole address space with a 32-bit float, then there will effectively be only three values. Even if a sufficiently high resolution float were used, one that can differentiate individual addresses at such a large scale, it would not lead to a better trained network. This is due to the fact the neural networks are fairly invariant to noisy data. As a result, even if an extremely precise float was used, tiny changes in floating point numbers would just look like noise to the network. These issues mean that normalizing addresses in the standard way is not ideal.

3.3.4 Address Clustering

To deal with the problem of normalizing addresses in the method described above, we opted to cluster the different addresses into discrete regions. Clustering was done using the popular K-Means clustering algorithm [13]. The number of clusters was chosen manually based on inspecting the distribution of addresses histograms, on the whole address space. As an example, the 607.cactuBSSN_s dataset as shown in Figure 5 was clustered into three regions.
Figure 5: Distribution of addresses misses for a 100 million instruction window on the 607.cactuBSSN_s dataset. When viewing the whole address space, misses are very sparse. However, when zooming in on one of the bars it can be seen that it is very rich in information. Due to floating point resolution this information will be lost if normalization happens across the whole address space.

Once clustering was complete, addresses were then normalized locally on a cluster by cluster basis. This allowed the normalized address values within each cluster to properly span the normalized range 0 to 1. Clustering locally however leads to the issue of addresses in one cluster having similar normalized values to those of addresses in completely different clusters. To properly differentiate these addresses we created a new input feature known as the clusterID value. Each feature vector had its respective clusterID appended as a third element, to be input to the network during training. Doing this was the last step of our preprocessing.

Figure 6: Clustering of addresses misses for a 100 million instruction window on the 607.cactuBSSN_s dataset. Address space is clustered into 3 regions as shown by the colors. Normalization of address values then occurs locally within each region.
3.4 LSTM Neural Network Prefetcher Architecture

With preprocessing complete, data could now be sent to the network for training. Several different types of LSTM designs were considered. We briefly considered the idea of an LSTM network that used an embedding layer. Embedding layers are common when solving Word2Vec problems in NLP. Since the long term hope is to move towards having a smaller network however, we decided that using an embedding layer was not ideal. Embedding layers encode information about how the different classes, in our case deltas, relate to each other [14]. While this is very useful for prediction accuracy, it is very costly in terms of memory. Embedding layers store class relationships as a matrix of size $N \times N$ where $N$ is the number of classes. With a class size as large as ours this memory footprint becomes far too large to be acceptable.

Since the network was being considered for use as a hardware prefetcher, we wanted our design to be as simple as possible, while still having a high prediction accuracy. The network we used is shown in Figure 7. The network consisted of two stacked LSTM layers connected to a dense layer with a softmax activation. The input features to the networks were a three element [PC, Virtual Address, ClusterID] vectors. The network output a softmax probability vector of length 10,000, our number of classes. Exact values for each part of the neural network are shown in Table 5. These values yield slightly over 2 million trainable parameters that needed to be stored. Additional values, such as those chosen for various hyperparameters may be found in appendix section 7.4.

**LSTM Prefetcher Architecture**

![Diagram of LSTM Prefetcher Architecture](image)

Figure 7: The neural network architecture used to create the LSTM prefetcher.

<table>
<thead>
<tr>
<th>Layer</th>
<th># Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>3</td>
</tr>
<tr>
<td>LSTM 1</td>
<td>128</td>
</tr>
<tr>
<td>LSTM 2</td>
<td>128</td>
</tr>
<tr>
<td>Dense</td>
<td>10K</td>
</tr>
</tbody>
</table>

Table 5: Values chosen for the number of neurons present in each layer of the neural network. Total number of trainable parameters was slightly over 2 million.
3.5 Design of LSTM with Stream Prefetcher

When the LSTM network was paired with the stream prefetcher it used exactly the same architecture as before, as shown in Figure 7. The only difference was in the data that the network was trained on. The stream prefetcher was run as part of the cache simulator. Since the cache simulator only outputs cache misses, any would be misses caught by the stream prefetcher were not written out. This resulted in a dataset of reduced size when training the network. The effect of this was that the network was only trained on, and made predictions for, misses not taken care of by the stream prefetcher.
4 Results

4.1 LSTM Prediction Results

Every program simulated by the cache simulator had two separate datasets. The first of these two datasets represented all of the misses that occur due to regular cache activity. The second dataset represented all misses that occur even after a simple stream prefetcher was turned on. The LSTM prefetcher was trained separately on each of these two datasets. Its performance was also measured separately. This was done to try and see if there was a difference in how much the LSTM was able to learn when it got to see all of the misses, and what it learned when being trained on just a subset of the misses.

For each dataset the accuracy LSTM prefetcher was evaluated twice under different conditions. It was run once where it issued 4 prefetches at a time, and once where it issued 10 prefetches at a time. Issuing 4 prefetches at once is a fairly reasonable amount when compared with existing hardware, while 10 is relatively aggressive. Selecting which deltas are prefetched was done by taking the top $N$ values from the softmax vector output by the neural network.

![LSTM Prefetcher Accuracy, No Stream Prefetcher](image1)

![LSTM Prefetcher Accuracy, Stream Prefetcher ENABLED](image2)

Figure 8: Accuracy of predictions for the LSTM prefetcher, LSTM+Stream prefetcher, and statistical prefetcher for each of the datasets. The top chart show the accuracies that occur when the stream prefetcher is turned off. The bottom chart shows the effects of turning the stream prefetcher on.

In addition to the results of the LSTM and LSTM+Stream prefetcher accuracies, data is also displayed for what we are calling statistical accuracy. Statistical accuracy refers to how successful a prefetcher would be if all it did was to guess the deltas that occurred most often in
the dataset. As a reminder, due to regular strided access patterns exhibited by many programs, most datasets have a small subset of deltas that occur far more often than others. Using this information, it is possible to take the top $N$ most often occurring deltas for each program, and design a prefetcher that guesses these values exclusively. In some cases surprisingly high accuracy can be achieved with this method. Results are provided for a prefetcher that guesses on the top 4, and top 10, most often occurring deltas.

The motivation for these statistical tests was to get an idea of whether or not the LSTM prefetchers were actually learning access patterns based on local behavior happening in each of the programs, or whether they were overfitting to only guessing deltas that occurred very often. Figure 8 shows the results for each of these tests.

4.2 Learning Memory Access Patterns Results

Figure 9 is a copy of the accuracy results from the Google research paper *Learning Memory Access Patterns* [2]. Their results are shown here so that they may be used as a point of reference in the discussion section.

![Figure 9: Accuracy results presented in the Google paper *Learning Memory Access Patterns* [2]. The Stream and GHB tests represent traditional prefetchers, while the Embedding, and Clustering results are accuracies achieved when using LSTMs.](image_url)
5 Discussion

5.1 LSTM Prefetcher Accuracy

Depending on the program being run, the accuracy of the LSTM prefetcher could vary substantially. Microbench1 had an accuracy of nearly 100 percent in all cases. This was because the misses in the program overwhelmingly consisted of only a single delta. Having a nearly perfect accuracy was expected, and indicates that the network functions well for simple extremely simple access patterns.

The LSTM's performance on microbench2 was substantially better than statistically guessing. For both 4 and 10 prefetches, the accuracy of the LSTM was close to 90 percent, while the statistical accuracy was closer to 35 percent. This is very encouraging evidence that some kind of access pattern was actually being learned. From section 3.2.2, it is easy from the programmer's perspective to see that microbench2 is simply two different strides for two different arrays. However, this is not as obvious when looking at the cache misses produced by the simulator. Since the memory accesses are interleaved, and at different strides, the two read points are continuously getting further away from each other. This leads to a continuously changing delta. The delta is changing in a way that is very predictable however, and it appears that the LSTM may have found this. A visual example of the effects of reading at two different strides is shown in Figure 10.

![Figure 10: The effects of reading from two different places in memory at the same stride vs reading at two different strides. When reading with different strides the read points can get further and further apart over time. This leads to a changing address delta.](image)

For microbench3, the accuracy was very close to 0 in all cases. This was somewhat disappointing but not altogether surprising. Microbench3 was purposefully designed to be very challenging. It is a program where one of the strides was always being changed through a combination of squaring and modulation. Though the deltas in microbench3 appear to change somewhat sporadically, they are not random. Subsequent runs of the program will always produce the same pattern of deltas. As mentioned in section 3.1, data for this project was gathered using an inorder cache simulator. This added to the complexity of the delta pattern since it automatically caused interleaving of each of the loads. It is possible that using a correct out of order cache simulator would make this program easier to learn for the LSTM, due to one of the strides being consistent.

The LSTM prefetcher on the SPEC 2017 benchmarks perlbench, bwaves, mcf, and cactuBSSN exhibited some fairly interesting behaviors. In the cases of perlbench, bwaves, and mcf, the LSTM had only marginally better accuracy than statistical guessing. This was true for both
the 4, and the 10 prefetches issued cases. This was fairly disappointing. Though in all cases the accuracies were significantly above 0, and therefore would likely assist in overall execution speed of the program, accuracies only slightly better than those of the statistical prefetcher indicate that the LSTM only learned a part of the miss pattern that occurred during the window of execution. There are several reasons why this could be occurring. The LSTM was trained in an offline scenario. This means that the weights of the network were static the accuracy was tested. Depending on how the misses were ordered when they are fed to the dataset during, the network may have overfitted to only part of the dataset. This could have affected the final prediction accuracy.

The cactuBSSN benchmark shows the only results where the statistical accuracy was higher than that of the LSTM. For both the 4 and 10 prefetch cases, the statistical accuracy was about 10 to 15 percent higher than that of the LSTM. This may be indicative of changing program behavior that the LSTM had a hard time learning. For this benchmark it could be the case that there were several different types of access patterns happening at separate times throughout the window of execution. Some of the access patterns may have used the same deltas as in other access patterns. It is possible however that the usage of these deltas may have occurred in different orders. It could be that the LSTM learned to predict one ordering of deltas better than it did the other ordering. The ordering of the deltas would not have affected the accuracy of the statistical prefetcher.

Depending on the type of work being done in a program at some point in its execution, certain strides could be being used continuously. Then, at a later point in execution, the program could be using a different set of strides continuously. When training in an offline scenario, the LSTM could potentially fit very well to first the initial set of strides exhibited by the program, and then later change its weights to fit very well to the second set of strides. Since the training was offline, the LSTM would then be evaluated using only the weights that it had at the end of training. Depending on the misses present in the test set being used to evaluate the accuracy of the LSTM, this could lead to a good accuracy, or a bad accuracy. If the data in the test set is very similar to the data that the LSTM saw closer to the end of its training, then there is a higher chance that the LSTM will perform well. However, if the data in the test set is very different, then there is a chance that the LSTM will perform poorly. If this occurs then it means that the neural network is overfitting too much to one part of the dataset, and not generalizing well.

In other domains that use neural networks, such as image classification, attempts are made to mitigate this problem by randomizing the order of the dataset, so as to deliberately remove any temporal relationship that might exist between dataset members [15]. It is not possible to do this in our case however, as the temporal relationship between dataset members is the basis for how to predict future deltas. In our case, a better way to avoid overfitting to a single part of the dataset would be to use and online training scenario. In an online training scenario, the weights of the LSTM would be continuously changed as predictions occurred. This would allow the LSTM to fit well to different parts of the dataset as they came and went. It would likely also allow the LSTM to work better on a wider variety of programs, instead of trying to find a static set of LSTM weights that generalized well to many types of program behaviors.
5.2 LSTM+Stream Prefetcher Accuracy

When looking at the accuracy results for the LSTM+Stream Prefetcher, as shown in the bottom half of Figure 8, it can be seen that in almost all cases, turning on the stream prefetcher to work along side the LSTM improved overall prediction accuracy. When the stream prefetcher was turned on, the dataset seen by the LSTM was normally only a subset of what it was trained on without the stream prefetcher. In some cases, the number of misses filtered out by the stream prefetcher was very high. In others, it filtered almost none. The differences in sizes of these datasets is shown in Table 6. In an ideal situation with the stream prefetcher turned on, the stream prefetcher should filter out the simpler patterns, and the LSTM should be able to focus on learning only the more complicated access patterns, and not risk overfitting to the simpler ones.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Misses, SP OFF</th>
<th># Misses, SP ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microbenchmark 1</td>
<td>833K</td>
<td>21</td>
</tr>
<tr>
<td>Microbenchmark 2</td>
<td>1,630K</td>
<td>26</td>
</tr>
<tr>
<td>Microbenchmark 3</td>
<td>2,878K</td>
<td>2,316K</td>
</tr>
<tr>
<td>600.perlbench_s</td>
<td>787K</td>
<td>459K</td>
</tr>
<tr>
<td>603.bwaves_s</td>
<td>1,951K</td>
<td>571K</td>
</tr>
<tr>
<td>605.mcf_s</td>
<td>1,718K</td>
<td>200K</td>
</tr>
<tr>
<td>607.cactuBSSN_s</td>
<td>1,081K</td>
<td>1,081K</td>
</tr>
</tbody>
</table>

Table 6: Number of misses present in each dataset with and without the stream prefetcher turned on in the cache simulator. Data is shown for a 100 million instruction window.

For microbenchmarks 1 and 2, the stream prefetcher had an almost 100 percent accuracy, leaving almost no misses for the LSTM to train on. This makes sense, as both of those benchmarks represent simple strides that a stream prefetcher should do an excellent job of finding.

In microbenchmark 3 the stream prefetcher had an accuracy of about 20 percent, while the LSTM remained at 0. Given the in-order nature of the cache simulator it is surprising that the stream prefetcher was able to function at all for this benchmark. Since loads were interleaved, it should have been very difficult for the stream prefetcher to find the static stride that exists from reading the first array. If the loads were separated instead of being interleaved then the stream prefetcher should be able to achieve an accuracy of closer to 50 percent.

The perlbench benchmark was the first time that the true effects of the LSTM and stream prefetcher working together were properly realized. For both the 4 and 10 prefetch tests, accuracy results were about 20 percent higher than they were previously when only the LSTM was used. Additionally, the performance gap between the LSTM and statistical prefetcher was substantially widened. Whereas before the LSTM was only about 5 percent more accurate than the statistical prefetcher, it is now almost 20 percent more accurate. This change in relative performance could indicate that the LSTM has learned much more about the actual miss behavior of the cache. This could be because it only needed to focus on the more complicated access patterns.

Unfortunately, this desirable performance gap between the LSTM and statistical prefetchers was not maintained in all cases. When examining the results of the bwaves benchmark, the accuracy gap shrunk. Overall, the accuracy of the LSTM+Stream prefetcher system was still much higher than the purely LSTM system. Interestingly the statical prefetcher became more
accurate than the LSTM, when it was paired with stream prefetcher. Since the stream prefetcher filtered out almost 70 percent of the data that the LSTM got to see, it appears that for this program the left over dataset was more difficult to learn.

The results from the mcf dataset, show a slightly less than 20 percent improvement in prediction accuracy. The LSTM+Stream prefetcher accuracy was close to 100 percent. The performance gap between the LSTM and statistical prefetcher was also much larger than it was with the stream prefetcher. Again, this indicates that actual miss patterns were starting to be understood by the LSTM, as opposed to reliable guessing. Overall the mcf benchmark represents the best case scenario of what can be hoped to be achieved with this design. In an ideal situation, the LSTM should be able to predict almost all of the misses that the stream prefetcher fails on. This appeared to be the case for mcf.

For the cactuBSSN dataset, the stream prefetcher has an accuracy of nearly 0. This suggests that there were no reliable strides that the stream prefetcher was able to learn. This leaves the behavior of the LSTM the same as it was before, along with the same relative performance when compared to the statistical prefetcher.

5.3 Comparison with Other Results

The programs bwaves and mcf that were tested as part of this project, were also tested as part of Google’s work, as shown in Figure 9. This provides an opportunity to directly compare accuracy results and comment on any differences. The LSTM prefetcher most similar to ours that was created as part of the Google paper is the Clustering prefetcher. Their accuracy results are based off of issuing 10 prefetches, so we will use the same amount when comparing our results.

For the bwaves benchmark, our results are close to as good as theirs, but only when using the LSTM+Stream prefetcher. When only considering the LSTMs however, our accuracy is about 30 percent lower. This could be due to several factors. Firstly, the design of the LSTMs are not exactly the same. While data acquisition and preprocessing are done in a very similar manner, Google trained their LSTM on a per-cluster basis. What this means is that they had a separate LSTM dedicated for each cluster in the dataset, instead of a single LSTM being used for all of the clusters simultaneously. This difference may have greatly assisted them with the problem of interleaved accesses that we experienced. It also would have allowed for greater specialization within their network, since a single LSTM would not have had to generalize to all clusters at once.

For the mcf benchmark, our accuracy is almost 50 percent higher. While at first this sounds like a great improvement, it is hard to say whether or not the results are very comparable. In the case of the bwaves benchmark the accuracy of the stream prefetchers used in both sets of results was fairly close. This indicates that the program behavior that was profiled was likely quite similar between both experiments. In the case of the mcf results however, they have a stream prefetcher accuracy of close to 0, while our stream prefetcher accuracy is close to 90 percent. Assuming that the two stream prefetchers worked in a relatively similar way, this suggests vastly different kinds of program behavior was being profiled on the mcf benchmark.

A possible explanation for this could be the phase of the program being simulated. For each of our simulations we executed the first 10 billion instructions before starting to gather results.
on misses for the 100 million instruction window. Depending on exactly where Google started gathering results, it is possible that the type of work being done by the program was completely different in their simulation than it was in our simulation. This makes it difficult to directly compare results between projects, and to conclusively say whether or not improvements have truly been made.

5.4 Address Space Layout Randomization (ASLR)

Late in the project it was discovered that there were unexpected differences in LSTM accuracies when trained on separate traces from the cache simulator. It was known that some of the activity of the programs being simulated was not completely deterministic, so it made sense to us that there should be some very small differences in the data that was gathered trace to trace. What we did not expect to see however was that the LSTM prediction accuracy would sometimes vary by as much as 20 percent due to these changes.

Initially we wondered whether or not our LSTM converged in an unstable way. It is certainly the case when working with neural networks that subsequent training runs sometimes yield different levels of convergence. This did not appear to be the problem however. We attempted many retrains on the network, and were always able to converge to about the same level of accuracy.

Eventually we wondered if our results were being affected by address space layout randomization (ASLR). ASLR is a security technique employed by most operating systems today that randomly offsets the starting locations in memory of many key data areas [16]. These areas will typically include the executable base, stack, heap, and libraries. This is done so that an attacker cannot reliably jump to a known location in memory where sensitive data might be stored.

The effect of this for our project is that different traces gathered from the cache simulator have different layouts in virtual memory. Two of these different layouts for the cactuBSSN benchmark are shown in Figure 11.

![Figure 11: ASLR as it affects the address distribution of the cactuBSSN dataset. The top two charts show the distribution of addresses across the global address space. The bottom two charts show how things can change on the local level.](image-url)
Having different memory layouts on subsequent traces poses a problem for being able to get consistent accuracy results when using a combination of virtual addresses and cluster IDs as inputs to the neural network. As previously mentioned in section 3.3.4, we manually chose the number of clusters for K-Means to use by inspecting the dataset as part of our preprocessing phase. However, when looking at the two top charts in Figure 11, it is no longer clear how many clusters to actually use. For one chart it appears that using only 2 clusters would be sufficient, while for another it appears that the number should be at least 3. To further complicate the problem, the ordering of data on the local level no longer appears consistent.

This inconsistency in the ordering of local data means that the corresponding \([\text{Virtual Address, ClusterID}]\) relationships, established on one trace, will be completely different on another trace. It appears that these differences have the power to greatly affect the overall predictive power of the LSTM. It also means that an LSTM trained on one trace will probably be unable to reliably predict misses that came from another trace.

This behavior is strong evidence that offline training solutions are not likely to yield networks that will consistently work well on a large variety of programs. With ASLR active, it may be very difficult to find a static set of weights that work well on subsequent runs of even one program. It seems more likely that using an online training setup would be a good solution to this problem.
6 Related Work

6.1 Learning Memory Access Patterns

By far the most similar research to this project is the 6th of March, 2018 paper Google published, *Learning Memory Access Patterns* [2], that this work is directly based on. Similar to this project, the Google work demonstrated that LSTM neural networks are able to predict cache misses at an accuracy often higher than that of typical CPU prefetchers. Similar to this project they also trained their LSTM using data from a cache simulator, and did so in an offline training scenario. The most important difference between our experiments and theirs, is our incorporation of a stream prefetcher as part of the overall prefetching system.

6.2 A Neural Network Memory Prefetcher using Semantic Locality

On the 6th of July 2018, researchers at the Israel Institute of Technology published work detailing the use of LSTMs as memory prefetchers, but for an online training scenario [11]. They implemented their LSTM prefetcher as part of a more fully working CPU using the open-source gem5 simulator [17]. By doing an online simulation this way, they were able to comment on a variety of details such as the silicon footprint, and energy requirements of their design. Additionally using gem5 to simulate their design allowed modeling many more advanced CPU behaviors such as out of order execution.

On a variety of SPEC 2006 benchmarks they were able to achieve an average speedup 30%. This is a very impressive result, and gives further motivation for pursuing the idea of using neural networks as prefetchers. If a neural network with similar performance to theirs could be made with a slightly smaller architecture, then it would be another step closer to what is necessary for a true hardware implementation.

6.3 Long Short Term Based Memory Hardware Prefetcher

In 2017, Yuan Zeng at Lehigh University wrote a Masters Thesis that also investigated the use of LSTMs as memory prefetchers [18]. Similar to our work they also investigated the use of a stream prefetcher to assist the neural network. Unlike our work, their LSTM required the use of a dedicated a page table to record local address deltas. This setup is convenient because it allowed them to achieve fairly good results with a relatively small LSTM compared to ours. The downside is that in order to function it required usage of the page table. The page table is of a finite size, so particularly long or complicated patterns will eventually exceed the limitations of its memory.

One of the other things that this paper emphasized is how certain key aspects of LSTMs could actually be implemented in real hardware. An example of this is a section on look-up tables to implement LSTM activation functions such as sigmoid, and tanh. Having hardware implementations for functions like these is critical to a hardware LSTM being possible.

6.4 Samsung M1 Microarchitecture

The Samsung Exynos M1 processor is recent example of a real CPU that features a neural network as part of its design [5]. Instead of using a neural network for memory prefetching, Samsung instead uses one to do branch prediction. Though not a very complex network, the fact that this is part of a real design shows that the potential for neural networks in computer architecture is being taken seriously outside of academia.
7 Conclusions and Future Work

Like other groups, this project has shown LSTMs have strong potential for being able help create more efficient memory prefetcers. It has been shown that LSTMs augmented with a traditional stream prefetcher are able to predict cache misses with an accuracy higher than that of either method by itself. However, there still exist some difficult problems that need to be addressed, before hardware implementations can be seriously considered.

7.1 Cluster IDs and Normalization

As part of our data preprocessing, K-Means was used to cluster addresses. Values were then normalized on a per cluster basis. This process for handling input data to the network is unlikely to be easy to do in any sort of hardware context. Both normalization and address clustering suffer from the same fundamental problem. In order to perform them effectively, global information about an address’ position relative to other addresses is required.

If a program is running and a new address comes up, how can that address be correctly normalized or clustered? Addresses that may come in the future could affect the normalized values and position of clusters that the prefetcher is having to deal with in the present. Normalization and clustering was only possible the way it was in this project because we had access to the entire trace at once. This allowed us to directly see global information about how the scales of the values of the addresses related to each other.

It may be possible to do some kind of normalization or clustering in a more local context, one based on data that has only been seen in the past. As we will now discuss however, It may be more effective to move away from using addresses entirely.

7.2 Changes in Training Data

As mentioned previously in section 5.4 ASLR can affect the perceived number of clusters across the address space, as well as the ordering of data in more local contexts. This creates problems for being able to get consistent prediction results when training a neural network based on virtual addresses and clusters.

Luckily, virtual addresses are not the only real-time pieces of information that a prefetcher would have access to. In this project virtual addresses were already used to create address deltas. These same deltas could be used as training data. Delta labels from previous time-steps could be used as input features on the current time-step. Furthermore, deltas would remain consistent in-between multiple runs of the same program. Using deltas would be one way to avoid the problems that are caused by ASLR.

For our neural network, we only trained it on PCs, addresses, and clusterIDs, that were generated as a result of cache misses. However it would be possible to include cache hits too. Training on hits as well as misses might greatly assist the LSTM with learning the true memory access patterns of a program, more than just the misses would. For many of our programs, the cache was generally hitting so often that the LSTM only got to see about 1 percent of the total load activity. If the LSTM was instead able to see all of the load activity as part of its training data, then it might be possible to effectively model cache misses with a significantly smaller LSTM architecture.
7.3 Online Training

As mentioned several times previously in this report, perhaps one of the best ways that this project could be improved in the future is to move towards an online training scenario for the neural network. This will likely help for a number of different reasons.

Programs often have multiple phases during their execution, periods of time where one type of activity is occurring followed by a period of time where a completely different activity is occurring. Often these different phases will have very different load patterns from each other. Training the LSTM in an online scenario would allow it to dynamically adjust its weights to be good as possible for the activity that’s happening purely in the present. It would no longer have to be trained in a way where it had to generalize to the whole program at once.

Though training the network in real-time could use more dynamic energy, it would also likely lead to a smaller network being required. This in turn would save energy. If the network could be trained quickly enough in real-time, then it would only need to be good enough to model the cache misses that are likely to happen in the imminent future. It would not need to be so complex as to consider all misses that might happen at all times. Training in real-time would allow correct prefetches to act as a kind of positive feedback loop, while incorrect fetches could be used to adjust its course.

7.4 Not Using a Recurrent Model

In this project we have only looked at using recurrent networks to model prefetchers, but there is no technical reason why other types of neural networks could not be tried as well. There are many cases where recurrent models can be replaced feed-forwards models [19]. This could be desirable because feed-forward models can be more easily pruned, which could help with reducing the size of the network in the future [20].
References


27


Appendix

Cache Simulator Settings

<table>
<thead>
<tr>
<th>Cache Simulator Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
</tr>
<tr>
<td>Cache Line Size</td>
</tr>
<tr>
<td>Associativity</td>
</tr>
<tr>
<td>Replacement Policy</td>
</tr>
<tr>
<td>Initial Instructions Skipped</td>
</tr>
<tr>
<td>Instruction Window</td>
</tr>
</tbody>
</table>

Table 7: Cache settings used for the cache simulator.

Neural Network Values

<table>
<thead>
<tr>
<th>Layer</th>
<th># Neurons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>3</td>
</tr>
<tr>
<td>LSTM 1</td>
<td>128</td>
</tr>
<tr>
<td>LSTM 2</td>
<td>128</td>
</tr>
<tr>
<td>Dense</td>
<td>10K</td>
</tr>
</tbody>
</table>

Table 8: Values chosen for the number of neurons present in each layer of the neural network. Total number of trainable parameters was slightly over 2 million.

Pseudo Code for Microbenchmarks

Microbenchmark 1

```
// Sum the values from an array.
for (int i = 0; i < 128; ++i){
    for (int j = 0; j < data_size/sizeof(int); ++j){
        result += data[j];
    }
}
```

Microbenchmark 2

```
// Sum the values from two arrays with different strides.
for (int i = 0; i < 128; ++i){
    for (int j = 0; j < data_size/sizeof(int); ++j){
        result += data1[j] + data2[j*4];
    }
}
```

Microbenchmark 3

```
// Sum the values from two arrays, but with a constantly changing stride.
for (int i = 0; i < 512; ++i){
    for (int j = 0; j < data_size/sizeof(int); ++j){
        result += data1[j] + data2[((j+j) % data_size/sizeof(int))];
    }
}
```
# Network Hyperparameters

train_prop = 0.8  # Proportion of training data vs validation data
sequence_length = 1  # Sequence length fed to the LSTM
batch_size = 32  # Batch training size
vocab_size = 10000  # Class size
lstm_neurons = 128  # Number of neurons in each LSTM layer
epochs = 10  # Number of training epochs

# Add network layers
model = Sequential()
model.add(LSTM(return_sequences=True, stateful=True))
model.add(LSTM(return_sequences=True, stateful=True))
model.add(Dense(vocab_size, activation='softmax'))

# Compile network
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['categorical_accuracy'])

# Train network
history = model.fit_generator(shuffle=False)