Machine Learning for Cache Replacement Policy

Wenhao Yu, Sherry Huang, Bodong Wang
yuwenhao@ucla.edu,sherryhxr99@ucla.edu,bow017@ucla.edu

ABSTRACT

Performance of high performance computing system is closely related to the performance of its cache memory. For this project, we aim to discover whether we would use machine learning to improve on current cache replacement policies. Moreover, we also propose to experiment with different type of machine learning based models to discover which kind might be most beneficial for cache replacement policy.

1 INTRODUCTION

With rapid development of high performance computing systems, improving the performance of cache memory is crucial to optimizations for such systems to bring better computational efficiency. One simple strategy would be increasing cache size, but it can be quite expensive. As such, we would like to improve on cache replacement policy. Better cache replacement policy allows for selecting correct positions for where data should be stored, which reduces memory access latency. Ideally, optimal cache replacement policy would create cache such that data is much more likely accessed directly from the cache, as a cache hit, than from a lower level in the storage hierarchy, as a cache miss. General approach for increasing cache hit can be further divided to two parts: (1) increase the cache hit rates by data and instruction prefetching to prevent future cache misses. (2) judiciously selecting which data to evict from the cache when making space for new data.

This project focuses on single-level cache replacement. In other words, when a new block of data is added to the cache, an existing block of data in the cache must be evicted to make space for the new block of data if the cache is full. This idea is illustrate in in figure 1. To achieve this goal, we focus on using machine learning models, CNN (convolutional neural networks) and LSTM (long short-term memory), to follow the above strategy. Specifically, our goal for this project can be summarized as below:

(1) We use machine learning models to pick candidate that resemble Belady's algorithm the most from existing cache replacement policy.
(2) We develop a pipeline such that the model predicts which cache replacement policy to use to select eviction candidate.
(3) We evaluate performance of the cache replacement policy generated by the model and perform analysis.

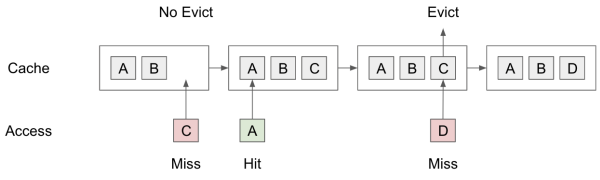


Figure 1: Single-level cache replacement.

2 BACKGROUND AND RELATED WORK

In this section, we will cover related cache replacement algorithms that either are used in our project or similar to our idea.

2.1 Conventional Cache Replacement Algorithm

Belady's algorithm The optimal cache replacement algorithm that aims at discarding the data that will not be needed for the longest time in the future. However, as we cannot know the future, this algorithm is not implementable in practice. In our project, we choose our candidate algorithm based on how close it is to Belady's algorithm. Further details on how we decide the closeness would be covered in the next section.

First in first out (FIFO) This queue-based policy works the same way as a FIFO queue. It evicts data in the order in which it was cached without considering the frequency or recency of thee data. This algorithm is one of the choice in our policy candidates.

Least recently used (LRU) This recency-based policy discards the least recently used data first as the name suggested. This algorithm is one of the most used cache replacement policies. It is also one of our available choices in cache replacement policy candidates.

Least frequently used (LFU) This frequency-based policy counts how often a block is used. It evicts data that is used fewest time from the cache. The eviction of entries with same frequency are picked arbitrarily. This algorithm is also in our choices.

2.2 Machine Learning based Cache Replacement Algorithm

LeCAR LeCAR[2] is a reinforcement learning based cache replacement algorithm that is designed for small cache sizes. It follows the same idea that targets at using machine learning to improve on existing cache replacement strategy as our project. It uses LRU and LFU as its base replacement policies and one adaptive weights for each policy. Each cache miss is served by either of the policy based on the probability distribution of weights of LRU and LFU. It maintains a FIFO queue that records the recent evictions from the cache, which is used for regret minimization. Regret minimization is an idea that suggests a better decision should be made in the past. As such, if a reference found in the FIFO queue, the regret associated with the policy made that decision would be increased and the weight associated with the other policy would also be increased indicting a better decision would been made.

PARROT PARROT[1] is a imitation learning based cache replacement policy. It is the first method to construct an end-to-end cache replacement policy that approximates Belady's. Similar as PARROT, our project also attempts at approximating Belady's. This algorithm first converts cache accesses (memory address and embedded PC)

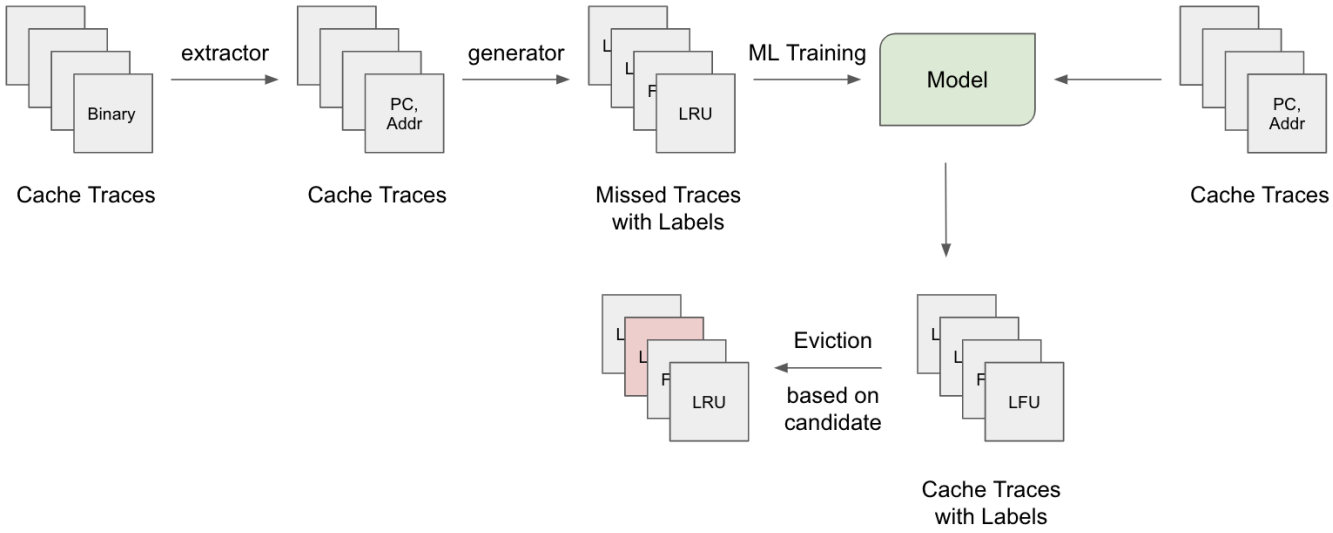


Figure 2: Proposed Approach

into states, which are further processed to minimize compounding errors. The states are then fed as input to a LSTM model, which is trained using back-propagation. While PARROT has proven to accurately approximate Belady's algorithm, it is relatively unpractical in real applications due to the overhead created by data processing and learning process.

3 METHODOLOGY AND DESIGN

In this section, we will discuss our proposed designs and how we evaluate our designs. Additionally, we will also discuss why the reasons behind taking such approaches. The general pipeline of our project is illustrated in figure 2: we first extracts program counters (PCs) and memory addresses from original cache traces; after extracting these information, we then use a generator to generate "labels", or selecting cache replacement policy, for missed data; once we have our training data, we train machine learning models to learn about how to select cache replacement policies that resembles Belady's the most; after obtaining the model, we use it to inference with the entire dataset; finally, we make eviction decision based the model's proposed candidate policy.

3.1 Extractor

As the first step in our proposed approach, we extracts information we need for the next step from original cache traces. This approach is adapted from PARROT[1] and mainly used ChampSim for collecting LLC access traces. Cache maintains a portion of data from a larger memory. If the cache is constructed optimally, we would have faster access to data. During a memory access, we need to search in the cache for the requested data. Often times, CPU caches are W-way set-associative caches of size N X W, where there are a total of N cache sets and each of them holds W cache lines. Programs would read from or write to memory addresses by executing load or store instructions. If the address is located in the cache, there is a cache hit. If the address is not found, there is a cache miss. Therefore, the bare minimum information we need from a cache line is memory address. If the cache already contains W lines, the cache replacement has to evict an existing line like we illustrated in Figure 1. Moreover, each of the load or store instructions has a program counter. Although PCs are not necessary for determining cache hit or miss, it is an important feature to keep for training.

3.2 Generator

During this step, we uses the extracted information from the previous step to generate a proposed candidate for each memory address. As we mentioned previously, we select LRU, LFU and FIFO as our candidate policies. These policies were selected because they are widely used and are from different categories. In the future works, it would be more beneficial to add more policies to the candidate pool. However, adding more policies would bring a larger overhead to this approach. As such, more details should be investigated before we get to that.

While both memory address and PC are used as input, only memory address is used to generate candidate during simulation. The method is shown in Algorithm 1: for each *trace* of the extracted dataset, we first check if the memory address of it is in the cache; if it is in the cache, we only update buffers for each policy and cache; otherwise, we check if eviction is needed; if cache is not full yet, we again only update buffers for each policy and cache; if an eviction has to be made, we iterate through all the proposed eviction choices, and calculate and compare its distance to belady's eviction choice; there, we pick the closest one; if they are the same, we would choose the first option. It is worth-noting that such decision of picking the first option when they are equal creates unbalanced dataset. Additionally, this approach assign labels to missed traces only, so the dataset is relatively small. These two factors make it hard for neural networks to learn. Therefore, we also construct a few alternative versions to create different datasets.

Because of the issues from the Algorithm1, we make some modification to the Algorithm1 1. To solve unbalancing of the sample, we pick the candidate randomly choice candidate from the candidate

Data: *traces*
Result: $X = (PC, memory\ address),\ y = policy$
initialization;
build index for belady's ;
for *each trace* **do**
  Pop the visit position for belady;
  if *addr* ∈ *cache* **then**
    update *LRU* queue and *LFU* frequency;
  else
    if *cache is not full* **then**
      update *LRU* queue, *LFU* frequency and *FIFO* queue;
      add *addr* to *cache* ;
    else
      each policy propose a candidate ;
      initialize *picked_candidate*, *picked_policy*, *max_access_position* ;
      for *each proposed candidate* **do**
        update *max_access_position* as belady's max distance;
        compare candidate's trace distance with max access position;
      end
      picked_candidate = candidate with longest trace distance;
      picked_policy = policy that generated *picked_candidate* ;
      $X \leftarrow trace$;
      $y \leftarrow picked\_policy$;
      update *LRU* queue, *LFU* frequency, *FIFO* queue and cache;
    end
  end
end

Algorithm 1: Label Generator

poll instead of the picking the first choice when reuse of candidates distances are equal. Alternatively, we transfer the question into a multi label questions. We labeled the optimal candidate from the candidate pool without picking them, and hope our model can learn to label the optimal candidate as well. We will pick the candidate in the inference stage, and the one being picked with be the candidate with the highest possibility to be the optimal choice. For dataset size issue, we use a map that stores previous choices for each trace to replace X, y; this way, if we encounter this trace in cache hit, we would choose its previous choice for it.

3.3 Machine Learning Approaches

After obtaining the dataset, it would be passed into machine learning models as the next step in our proposed approach. In this section, we will discuss the different types of models we experiment with, as well as additional data processing steps we take as a part of training procedure.

For models to work correctly, we need to first encode the input X (PC, Memory address) as they cannot take hex numbers directly as input. Hence, we encode it in two ways (1) convert hex to decimal, (2) one hot encoding. However, both of the approaches have their flaws and might neglect some structural information. During our experiments, the first approach seems to fail reveal valid patterns for models to recognize. While the second approach allows models to learn properly, it is extremely costly in memory and seems unpractical in real applications. More details on their effects can be found in Evaluation section.

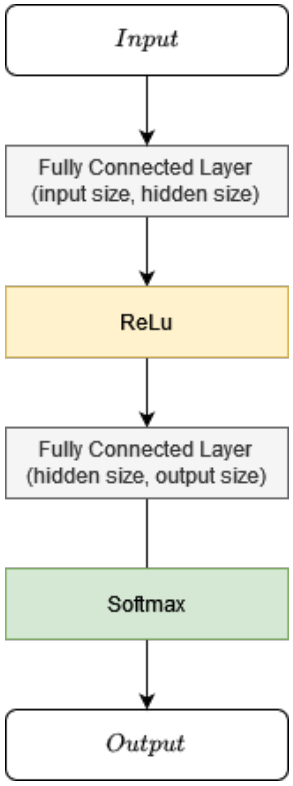


Figure 3: FCNN architecture

3.3.1 Fully Connected Neural Network. Fully connected neural networks consist of series of fully connected layers with each neuron in one layer connected to every neuron in adjacent layers. The intrinsic structural agnostic nature of FCNN means it is a more broadly and generally applicable neural network. With the neutral property of FCNN, we can apply it to our dataset with no special spatial and temporal assumptions made about the input. FCNN is also one of the simpler forms of neural network. For the above reasons, we decided to use a two-layer fully connected neural network with small hidden size of 10 as our baseline machine learning model in evaluation. We chose a two-layer fully connect neural network since our dataset only contains two training features, more complicated machine learning approach could lead to overfitting. Another advantage of fully connected neural network is that its output can be readily formulated into a closed-form expression, which can be easily fitted into a hardware implementation.

3.3.2 CNN. CNNs are commonly used for computer vision tasks as the architecture of convolutional neural networks mimics the neurons in human brains and organization of the visual cortex.

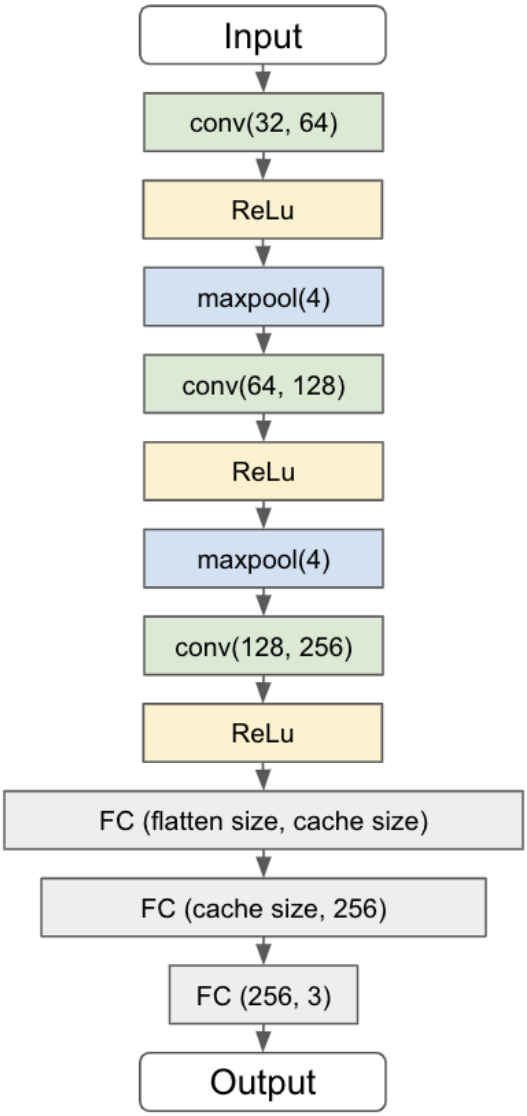


Figure 4: CNN architecture

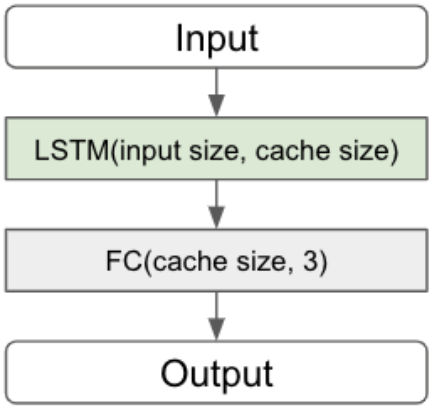


Figure 5: LSTM architecture

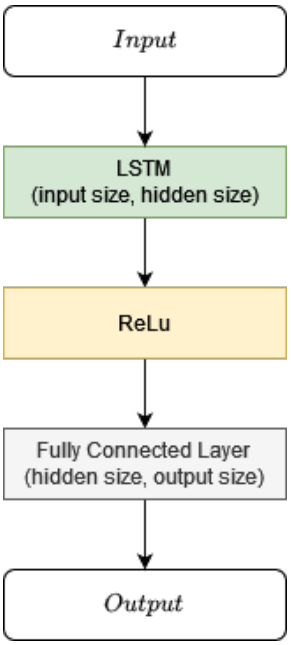


Figure 6: Alternative LSTM architecture

However, it has also shown to work with sequential data as it captures both the spatial and temporal dependencies of data. The structure of our proposed model is shown in figure3. The kernel sizes vary for two encoding types as the first approach has fewer features. Each of the convolution and pooling layer is in 1D. While it is possible to conduct 2D convolutions, it appears to be more reasonable to use 1D for sequences. In addition, we choose cross entropy loss for this task. The results are presented in evaluation section.

3.3.3 LSTM. LSTM is widely used for sequential data and popular among similar tasks. As we discussed in the previous section, PARROT[1] uses this method. The structure of our proposed model is shown in figure4. Unlike CNNs, we choose negative log-likelihood for its loss function. While negative log-likelihood and cross entropy loss are both for multi-class classification tasks, the former is more suitable for LSTM.

Our group also explored LSTM with additional dataset generated with different features and dimensions. Input generator was modified to generate more features that we considered to contain more temporal and spatial pattern. Spatial position of the entire cache state and each cache line's frequency and recency was encoded as input. Since this iteration of LSTM model uses a different dataset, we are also experimenting with altered parameters. For example, to further exploit the temporal sequential relationship between consecutive inputs, we tried using different sequence lengths. In order to fit the varying sequence lengths we also tested out different numbers of LSTM stacked layer. Additionally, to better capture the

long-term dependencies within the training data, we adjusted our hidden layer size.

3.4 Eviction

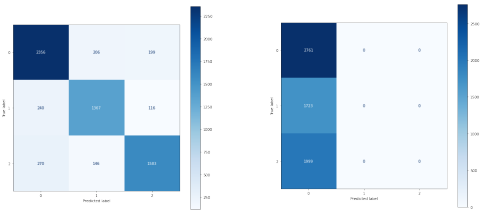
Data: *traces*
Result: *HitRate*
initialization;
for *each trace* **do**
  if *addr* ∈ *cache* **then**
    hit += 1 ;
    update *LRU* queue and *LFU* frequency;
  else
    miss += 1 ;
    if *cache is not full* **then**
      update *LRU* queue, *LFU* frequency and *FIFO* queue;
      add *addr* to *cache* ;
    else
      each policy propose a candidate ;
      use model to predict *picked_policy* ;
      eviction candidate = eviction chose by *picked_policy*;
      update *LRU* queue, *LFU* frequency, *FIFO* queue and cache;
    end
  end
  HitRate = hit / (hit + miss) ;
end

Algorithm 2: Eviction

As the last step in our proposed approach, we will first conduct inference with the model trained in the previous step and select eviction accordingly. The algorithm for eviction is similar as the generator as demonstrated in algorithm 2. Instead of assigning policy through calculations, we use the trained model to predict for policy. Once we obtain the prediction, we choose the eviction based on the policy. Under this setting, the model is supposed to work as a cache replacement policy that resembles Belady's behavior. We also calculate the hit rate using model as cache replacement policy during this step. Detailed results will be discussed in the next section.

4 EVALUATION

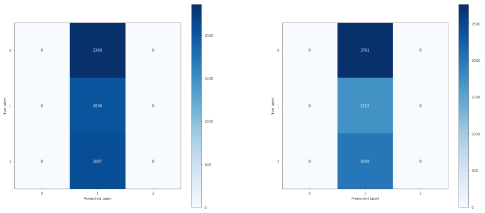
We used the data from the 2nd Cache Replacement Championship (CRC-2) and trained the models on a GPU. There is no specific hardware restriction for our experiments. There are 33437 cache lines in total. The training and testing data for the model is split based on 80:20 ratio. Each experiment can be complete on GPU in a short time as the model is relatively light-weighted. We generate a few datasets as described in the previous sections; yet, using unbalanced dataset leads to no meaningful result. The results shown below are all produced using balanced datasets.



(a) CNN w/ one hot encoding

(b) CNN w/ integer encoding

Figure 7: Confusion Matrices for CNN



(a) LSTM w/ one hot encoding

(b) LSTM w/ integer encoding

Figure 8: Confusion Matrices for LSTM

4.1 Accuracy

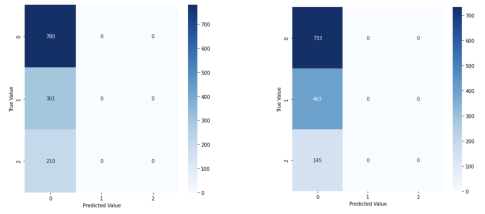
The accuracy of our proposed models using balanced datasets are demonstrated as confusion matrices for direct comparisons. As we can see from confusion matrices, CNN model with integer encoding fails to recognize valid patterns. However, CNN model with one hot encoding reaches accuracy over 80%. While CNN model seems work well with one hot encoded inputs, using one hot encoding requires us to encode all trace first. As such, we get 7579 element for each trace, which is quite large. Since our goal is to improve performance of cache, this practice seems counter-intuitive.

Unlike CNN model, LSTM model seems to favor 1 (LRU) over others. Since it is not outputting 0, it must recognize some patterns. However, it is worth investing why 1 (LRU) is strongly favored by LSTM. Since LRU actually achieves the best hit rate among our candidate policy, there might be a reason behind LSTM's behavior. It is also likely that this result is caused by the difference in loss function. Yet, CNN does not have a performance gain using negative likelihood loss instead of cross entropy loss.

Another interesting thing to observe is that while the alternative LSTM uses input of integer representation, the input itself includes different features and dimensions. Compared to the original LSTM methods, which outputs consist of solely 1 (LRU), the alternative LSTM favors output 0 (LFU). Compared to the original LSTM's input, this version of LSTM used input that has more spatial related features. The new input also added more recency and history information; however, the fact that prediction strongly favors LFU policy could suggest that the model fails to recognize some of the temporal pattern between inputs. It is something we would like to investigate further in the future.

Table 1: Hit Rate Summary. CNN1 represents CNN with one hot encoding, CNN2 represents CNN with integer encoding, LSTM1 represents LSTM with one hot encoding, LSTM2 represents LSTM with integer encoding and optimal presents hit rate generated by pure calculation and comparison with belady

Cache replacement policy	Belady	LFU	LRU	FIFO	CNN1	CNN2	LSTM1	LSTM2	Multi-Label	Optimal
Hit Rate	0.8146	0.2417	0.7006	0.5834	0.2787	0.2417	0.7006	0.7006	0.313	0.6940



(a) FCNN **(b)** Alternative LSTM

Figure 9: Confusion Matrices for FCNN and alternative LSTM

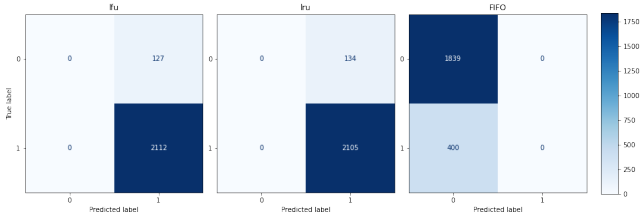


Figure 10: Confusion Matrices for CNN w/ Multi labels

4.2 Hit Rate

The hit rate simulated using each base policy and our resulting policy is shown in Table 1. While it appears that approaches use LSTM achieve higher hit rate among our approaches, it is not a progress advanced by LSTM. Instead, LSTM merely produces 1 (LRU). Therefore, it obtains the same level of hit rate as LRU. One thing particularly worth-noting is that our proposed approach–picking policy based on their similarity to Belady–does not work as expected. Ideally, it should reach a hit rate that closely resembles Belady's. However, it only obtains a hit rate around 0.69, which is worse than using LRU alone. For our CNN model that works properly, it only reaches a hit rate of 0.2787, which is only slightly higher than the worst of our base policy. While it does have the ability to predict for candidate policy, it is possible that switching policies frequently affects cache performance negatively.

REFERENCES

[1] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An Imitation Learning Approach for Cache Replacement. (2020). https://doi.org/10.48550/ARXIV.2006.16239

[2] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *HotStorage*.

A STATEMENT OF WORK

Work divided as follow:

Sherry Huang: Administered the project, built the CNN model.

Wenhao Yu: Built the multi-label model, adapated the trace extraction code.

Bodong Wang: Built the FCNN and LSTM model, organized the materials.