Diminished Triad : Consistent Hashing in Redis

Pallavi Agarwal, Manindra Kumar Moharana, Sanjeev Jagannatha Rao
University of California, San Diego
p1agarwa,mmoharana,sjrao@ucsd.edu

1. Abstract

Diminished Triad is a (partial) diminished implementation of chord by the three authors on Redis backends. Redis is an open source, BSD licensed, advanced key-value cache and store. While Redis offers an extremely fast key value cache, its design suffers from some major drawbacks. Redis does not support consistent hashing, the slaves of a particular master are only aware of the master that they are replicating and unaware of the state of the rest of the system. Redis also does not support automatic migration of keys when a new node joins or readjustment of responsibilities when a node fails. Diminished Triad aims to address these drawbacks of Redis at the cost of an additional layer of indirection and its associated overheads.

2. Introduction

"Redis is an open source, BSD licensed, advanced key-value cache and store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, sorted sets and bitmaps"[**?**] . Twitter, GitHub, Weibo, Pinterest, Snapchat, Craigslist, Digg, StackOverflow and Flickr are some of the well known companies currently using Redis. Redis prides itself on providing very fast performance and several of its design choices have been made with this primary objective. For example, Redis supports asynchronous master slave replication. However, if the master dies before the data is replicated to the slaves, that piece of data is lost. Redis backends are also either only a slave or master and can not be both. A slave replicates a particular master and is unaware of the rest of the system. If a Redis system has r backups, it can support at most r random node failures. If we loose a master and all its slaves we loose all the master's data. Redis contains 16348 key slots. Every key maps to one of these key slots. Redis does not support consistent hashing. When a system is started all the key slots are evenly distributed among the masters. When a new node joins the system, keys need to be migrated manually to the new node.

In order to achieve its outstanding performance, Redis works with an in-memory dataset. So it offers very high write and read speeds with the limitation that data sets can't be larger than memory. Being an in memory dataset, Redis can work with complicated data structures and supports several of them with very little internal complexity. Redis can persist data by either dumping the dataset to disk every once in a while, or by appending each command to a log. We have adopted the latter approach with our Redis backends as it works like a redo log to migrate data when a new node joins the system.

In Diminished Triad (DT), we hope to provide a fault tolerant systems using Redis backends that improves on several drawbacks of the existing Redis system described earlier. First, DT follows a consistent hashing system similar to Chord [1]. It hashes every backed and keys to a position on the ring. Second, DT automatically moves keys between nodes every time a node joins or leaves the system. Third, DT system, like chord is completely distributed and no node is more important that the other.

DT maintains two invariants

1. Each node is the master for all the keys that hash between its predecessor and itself.

2. All the data that a node is responsible for (keys for which it is the master) is replicated in the r nodes that follow it on the ring.

Every node is thus a master for some of the keys and is a backup for the r nodes that precede it. Every time a node is added to the system or a node fails and leaves the system, the keys are redistributed amongst all the backends to maintain the two invariants.

The rest of the paper is organized as follows. Section **??** contains the detailed design of Diminished Triad. This section mentions the various components of DT, their individual roles, the APIs used to communicate between these components. Section 4 provides the implementation details of DT. Section 5 provides various metrics that measure the performance of Diminished Triad. This section discusses the relative merits and demerits of Diminished Triad over off the shelf Redis.

3. Design

We first define several terms that will be helpful in describing the system.

1. User/Client - A user or a client is the customer for the Diminished Triad service. For example, a client requests to set values for some keys, get values etc.

2. DTServer - All requests like get and set are made by the client to the DTServer. The DTServer then forwards the request to the appropriate backends that are responsible for the key being queried through the DTClient

3. DTClient - The DTClient is the only channel of communication with the Redis Backend. The DT client transforms the requests received by the DTServer into form understandable by the Redis Backend and queries the appropriate Redis Backend.

4. DTInstance - DTInstance comprises of the DTServer and a pool of DTClient instances

5. Redis Backend - This is the off the shelf Redis Server instance that provides all the key value store functionality to Diminished Triad.

6. DTClientAPI - The user talks only to the DT instance and is unaware of the intricacies of the system. From the user's perspective the DT instances provide a fault tolerant distributed key value store. The DTClientAPI defines all the functions exported by the DTServer to the client. This API makes hides the internal details of DT and makes it look like a fault tolerant key value store to the programmer. A list of all the functions exported DTClientAPI is mentioned later.

7. DTServerAPI - The DTInstance, in particular the DTClient talks to the Redis Backends though a set of functions exported by the Redis Backend. This API is completely internal to the DTInstance and the user is completely unaware of the DTServerAPI.

8. Sentinel - The sentinel is the watchdog for the Redis Backends. It actively monitors if the Redis Backends are working as expected. The Sentinel also notifies through an API when something is wrong with one of the monitored Redis instances.

9. DTSentinel - The DTSentinel listens to notifications from the sentinel for any changes like a node leaving the system or joining the system and takes appropriate action in each case to maintain the DT system invariants.

3.1. System Layout

Every node runs a Redis Backend and a DTInstance. All DTInstances are identical and none of them are more important than the other. A client can talk to any DTInstance using the DTClientAPI. The DTClient will forward the request to the appropriate Redis Backend using the DTServerAPI. Note that the Redis Backend servicing the client's request may or may not be on the same machine as the DTInstance. At the moment we have a single Sentinel and DTSentinel instance in our system. We can make our system tolerant to failures of the Sentinel and DTSentinel with minor configuration changes to the Sentinel configuration. The section **??** on future work briefly mentions how this can be accomplished. Without loss of generality we will be discussing the system with just one instance of Sentinel and DTSentinel and assume them to be free from failure. Figure **??** shows a typical set for Diminished Triad.

3.2. DTClientAPI

Diminished Triad exposes the following functions to the user. This is a subset of the functions exported by the Redis Backend in the DTServerAPI. A complete implementation of Diminished Triad will ideally expose every single function exported by the Redis Backend. We have limited our scope to the functions listed below.

- get(key) - gets the value associated with a particular key
- set(key,value) - set the value associated with a particular key
- getStrlen(key) - get length of a value associated with a particular key
- append(key,value) - append a value to the value already associated with a particular key
- lPush(key,value) - insert a value in the first position to the list associated with a particular key
- lPop(key) - remove and return the first element of the list associated with a particular key
- lIndex(key,index) - return the element at a particular index in the list associated with a particular key
- lLen(key) - get length of list associated with a particular key

3.3. The Chord Ring, Hashing Keys, Master and the R Backups

The Chord ring has N positions in it. Every node hosting a DTInstance and a Redis backend is hashed to one of these N positions. Every key is also hashed to one of these

N positions. As we move clockwise in the chord ring, we encounter increasing values of hash values until we reach the maximum and then go back to 0. Every backend is the master for the all the keys that hash to its position in the ring through the hash positions up to but not including its predecessor in the ring. The predecessor of a backend is defined as the first live backend that is encountered as we move anti clockwise in the chord ring. Similarly the successor is the first live backend that is encountered as we move clockwise in the chord ring. The r backends that immediately succeed the master serve as the r backups of that particular master.

3.4. Handling Read and Write Requests

A read or write request is issued by the user or client to the DTInstance. The request reaches the DTServer running in the DTInstance through the DTClient API. The DTServer receives the request, hashes the key to figure out the master and the r backups corresponding to the key. It then performs the actions required for the request on the r + 1 redis backends (mater plus the r backups) using the DTClient to connect to them which implements the DTServerAPI to talk to the redis backends. If the request is a write request, the DTInstance writes synchronously to the master and all the backups. If it is a read request, the DTInstance returns on the first successful read starting with the master and then trying the r backups.

3.5. The Sentinel, DTSentinel

The Sentinel is a watchdog for all our Redis backends. In Diminshed Triad, we use the Sentinel for performing the following:

- Monitoring - Sentinel constantly checks if the Redis Backends are working as expected
- Notification - Sentinel notifies the DTSentinel thorough pub/sub messages that something is wrong with one of the monitored Redis Backends.
- Configuration provider - The sentinel maintains the configuration of all the currently alive Redis Backends.

The Sentinel itself is designed to run in a configuration where there are multiple Sentinel processes cooperating together. We have not configured our Sentinel to work this way but this can be accomplished with minor configuration modifications. These changes make the Sentinel fault tolerant to failures.

The DTSentinel listens to notifications from the Sentinel. Every time a node fails or a new node joins, the DTSentinel receives the notification from the Sentinel and is responsible for redistributing data and maintaining the two Diminished Triad invariants. The next section describes how data is moved when a node fails or a new node joins the system.

3.6. Handling Failures, New Joins and Data Redistribution

Figure 1 shows the data migration that is done when a redis backend fails. For the purposes of this section, we assume our system is configured to have two backups for every master. The initial configurations of the system is shown inside the chord ring. M represents a particular master, B1 and B2 its two backups, M-1 and M-2 its two predecessors. In the figure, M is assumed to have failed. The new configuration after M fails is shown outside the chord ring. In order to get to the configuration and maintain the two Diminished Triad invariants, we need to perform 3 data migrations. In general, if we have r backups, we need to perform r + 1 data migrations. Each of these data migrations is shown by an arrow in figure 1. The arrow originates along a section of the chord ring and terminates at a redis backed. A copy of all the keys that are hashed that section of the chord ring have to migrated to the redis backed that the arrow is pointing to from a redis backend that has all the keys that are being migrated. For example, when M fails, we will move keys for which it was the master , i.e. the keys that originally hashed between M and M-1 to redis backend B2. Since B1 was originally a backup of M, it will have this data and all this data will be migrated from the original B1 to the new reconfigured B2. This data migration is shown by the arrow at originated between the original M and original M-1 redis backends. Similarly we move data corresponding to each of those arrows. Once data migrations is completed, the Diminished Triad invariants would have been restored after the failure.

Similarly, when a node joins, if the system is configured to have r backups, a set of r+1 data migrations is sufficient to bring the node into the system and restore the Diminished Triad invariants. Figure 2 shows the data migration that is carried out when a new node joins.

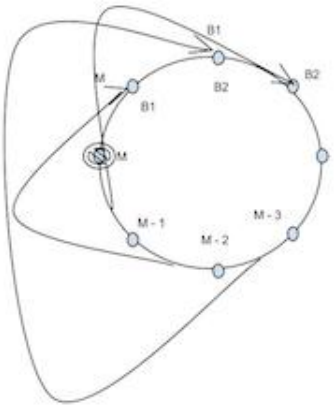


Figure 1. Data Migration During Failures

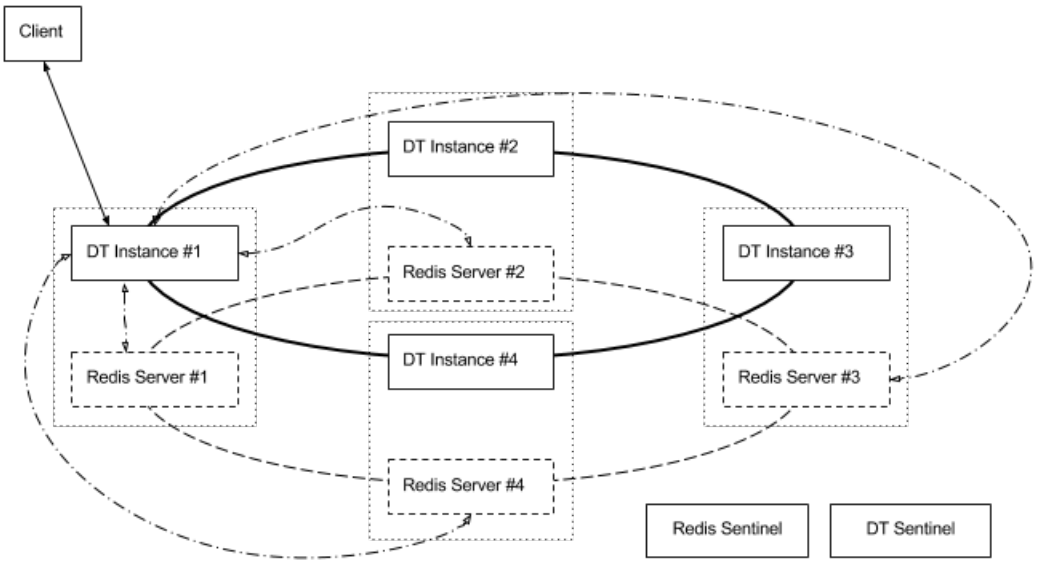


Figure 3. svg image

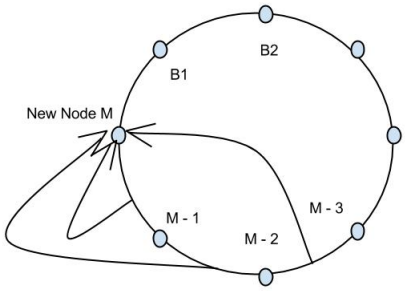


Figure 2. Data Migration During Node Join

4. Implementation

We used Python 2.7 to implement the DT Server, DT Client and DT Sentinel. Python was chosen because of its flexibility and ability to prototype and iterate faster.

DT Server: Each DT Server functions as a RPC server. We use the spyne rpc toolkit for python to create the RPC server. For each function in the DTClient API (get, set, append, etc.) there's a corresponding RPC function call along with the necessary parameters. RPC calls can be made by HTTP post requests, passing in the redis command and parameters as the post arguments. A DT Server has a upto date list of alive nodes in the ring. When a get/set command is received for a particular key, the server hashes the key and finds out which redis server the key is mapped to. It then forwards the request to the particular server and its backup using DT Client.

DT Client: This the component that talks to the redis servers. Each DT Client is connected to all the redis servers in the ring. We use the redis-py python library for redis. It greatly simplifies the process of talking to a redis server by exposing all redis commands via easy to use API. When the DT Client receives a request for issuing a command for a redis server, it issues the command with the help of redis-py.

DT CLI: To make it easy for the user to send commands to redis, we also designed a utility DT CLI which provides similar behaviour as redis-cli. User can input redis commands like set foo bar, get foo, etc. These commands are converted into a http request and sent to the DT Server.

DT Sentinel: The DT Sentinel runs in a separate process. Its task is to monitor any configuration changes in the ring and update all the active nodes about the latest configuration. The DT Sentinel is backed by the redis sentinel which is a special sentinel mode of operation of a redis server. In sentinel mode, a redis server doesn't store any key data but monitors other redis servers and can send push notifications on configuration change. DT Sentinel starts up a redis sentinel and subscribes to status change notifications. Any time a node leaves or joins the ring, DT sentinel gets the message from redis-sentinel. DT Sentinel then performs any

necessary data migration between nodes and finally sends the latest node configuration to all nodes in the ring.

Deploying on AWS:

We tested our system on Amazon AWS. We created 4 instances with 1 GB RAM, 2.2 GHz CPU, running Debian. We started 3 redis servers and instances on each of the machines, redis sentinel and DT Sentinel on our local machine. We had written scripts which would install all dependencies and dt instances to automate the entire installation process.

We also wrote scripts for evaluating the performance of the system. These scripts wrote random keys and values and measured the latency/consistency with varying values of num-backups, num-servers, etc.

5. Evaluation

5.1. Comparing Fault Tolerance and Probability of Losing Data between DT and Redis

Consider a system with N nodes and r backups. Assuming infinite capacity at each node and sufficient time in between failures, DT can handle $N-1$ node failures. Under more practical considerations (just sufficient capacity, network bandwidth etc.), system is guaranteed to handle $N-r-1$ random node failures and still maintain r copies of all data. An off the shelf redis cluster system with r backups can loose data with $r+1$ targeted node failures as the backups are unaware of anything outside the master that they are replicating.

Lets assume that all nodes can fail with equal probability p. In the redis cluster system, assume a master has failed.

$$\begin{aligned} Pr(LosingData) &= Pr(FailureofAllItsBackups) \\ Pr(FailureofAllItsBackups) &= p*r/(N-1) \\ Pr(LosingData) &= p*r/(N-1)*p*(r-1)/(N-2)* \\ &\quad p*(r-2)/(N-3)*...*p/(N-r) \\ &= p^k*r!*(N-r-1)!/(N-1)! \end{aligned} \tag{1}$$

Lets compute the probability of losing data in the Diminished Triad System. DT is assumed to be fault tolerant to N - r - 1 random node failures.

$$\begin{aligned} Pr(LosingData) &= Pr(N-rNodeFailures) \\ &\quad Pr(N-rNodeFailures) \\ &= p/N*p/(N-1)*p/(N-2)*...*p/(N-r-1) \\ &= p^{N-r}*(N-r-2)!/(N)! \end{aligned} \tag{2}$$

The probability of losing data is significantly lower in DT as compared to Redis Cluster.

5.2. Consistent Hashing

One of the big improvements that DT offers over Redis Cluster is consistent hashing. We would like our consistent hashing scheme to have following properties suggested in Chord [1]. For any set of N nodes and K keys, with high probability, the following is true.

1. Each Node is responsible for at most $(1+\epsilon)$*K/N keys for some small ϵ
2. When a $(N+1)^{th}$ node joins or leaves the network, the responsibility for O(K/N) keys changes the hands.

Figure 4 shows a box plot of number of keys that were migrated when a new node joins with a certain number of existing nodes and keys in the system with measurements from several runs of the experiment with the same initial conditions. For example, in figure 4(a) the system had 10 nodes. We measured the number of keys that were migrated when an eleventh node joined the system. We generated keys at random and stored data in the system. The x-axis shows the number of keys that were in the system when the eleventh node joined the system.

The black line shows the value of K/N where K is the number of keys in the system and N is the number of nodes in the system. The black line helps compare the performance of our consistent hashing scheme with our theoretical targets for the system described above. The number of key migrations is consistently below the black line. As expected the variability in the number of keys migrated reduced as the number of keys and the number of nodes in the system increases.

6. Conclusions

conclusion and summary design here

7. Future Work

future work here

8. Acknowledgments

future work here

9. Dataset

The CIFAR-10 dataset [**?**] consists of 60000 32x32 images belonging to 10 classes with 6000 images per class. The training set consists of 50000 images and the test set consists of 10000 images. The classes are mutually exclusive with almost no overlap between images.

10. Why CNN?

Traditional neural networks (NNs) receive some input and transform it through a series of hidden layers. Every

layer consists of some neurons which are fully connected to all neurons of the previous layer. The neurons in a layer function independently without sharing connections.

In CIFAR-10, images are 32x32x3 (3 color channels), so a regular neural network would have 32*32*3 = 3072 weights. If we need multiple such layers, the number of weights could increase very quickly to an unmanageable degree. Therefore, this full connectivity is undesired and training too many parameters can lead to overfitting. CNNs have a special architecture to take advantage of the 2D shape of images. The layers in a CNN are arranged in 3 dimensions - width, height, depth. The neurons in a layer are connected to a small region of the layer before it, instead of a fully connected manner. Therefore they have lesser parameters than a fully connected network with same number of hidden units.

11. Recent Work with CNNs

We will now take a look at some of the existing work in image classification using CNNs. CNNs were first introduced by Kunihiko Fukushima in 1980 [**?**]. LeCun et. al. [**?**] improved the design and proposed LeNet-5, a 7 layer CNN for classifying handwritten digits in the MNIST dataset. Krizhevsky et. al. [**?**] broke new grounds in CNNs by achieving state of the art error rate on ImageNet LSVRC-2012 contest using a 8 layer deep CNN. Hinton et. al. [**?**] proposed an enhancement for CNNs using a random dropout layer that greatly reduced overfitting. The state of the art accuracy of 91.78% in CIFAR-10 was achieved by Lee et. al. [**?**] using their Deeply Supervised Net architecture which uses an additional companion objective function for the hidden layers, that is minimized along with the global loss. The network in network model by Lin et. al. [**?**] achieved an accuracy of 91.2%.

12. Layers in a CNN

A CNN consists of a number of convolution and subsampling(pooling) layers, followed by a fully connected layer as output.

12.1. Convolution Layer

The convolution layer can be interpreted as a 3D volume containing neurons along its depth [**?**]. Its parameters consist of a set of learnable filters which extend along the depth. In the forward process, each filter is convolved with a particular patch of the image along the depth. Intuitively, the network will learn filters that activate when a particular feature is observed in that part of the image.

12.2. Pooling Layer

Pooling layers are usually added between convolution layers in a CNN. This layer is used to reduce the size of the representation so that the computation and number of parameters in the network are reduced. It also helps prevent overfitting. This layer operates on every depth slice of the input layer and reduces it spatially using a pooling function. Common pooling functions are max pooling, average pooling and L-2 norm pooling. For example, if the kernel size is 2x2, it reduces 4 inputs to 1 output by applying the pooling function on the inputs.

12.3. Fully connected Layer

The neurons of this layer are fully connected to all activations of the previous layer.

12.4. Loss Layer

This layer computes the loss with respect to a target and assigns the cost needed to mininmize the loss. The loss is computed in the forward pass and the the gradient w.r.t loss is computed in the backward pass. Common loss functions include softmax loss, euclidean distance loss and hinge loss.

12.5. ReLU Layer

The rectified linear unit is a recently introduced layer that has gained popularity. This layer computes the function $f(x) = max(x, 0)$. It's basically a zero thresholding layer. It has been shown to significantly decrease the training time (by a factor of 6 as reported by Krizhevsky et al.).

12.6. Dropout Layer

The dropout layer is a recent important invention in neural networks by Hinton et. al. [**?**]. It addresses a fundamental problem in machine learning - overfitting. It accomplishes this by setting certain activations to zero (dropping out) during the training phase. For each training example, a different set of random units is selected to drop. The number of activations to be dropped is controlled by a dropout ratio parameter.

13. Caffe

Caffe is an open source deep learning framework developed by the Berkeley Vision and Learning Center. It's implemented in C++/Cuda and provides command line, python and Matlab interfaces. Caffe is built with speed in mind, and provides seamless switching between CPU and GPU. Another advantage of Caffe is that models can be written entirely using schemas, without writing any code. For this project, I'll be making use of Caffe to train multiple CNNs for CIFAR-10 classification.

14. Experiments

I created and trained multiple Caffe models and evaluated their performance on the CIFAR-10 dataset. For maintaining consistency, I trained each network for 20K epochs

using SGD and compared their performance on the test set. I used a learning rate of 0.001 for the first 10K epochs and 0.00001 for the next 10K epochs. I used the softmax loss function in the last loss layer.

The first network I tried is a 4 layer network consisting of 2 pairs of convolution and max pooling layers. It's architecture is described in table 14.

Layer Type	Parameters	Outputs
CONV	Kernel: 5x5, Pad: 2, Stride: 1	32
MAX POOL	Kernel: 3x3, Stride: 1	
CONV	Kernel: 5x5, Pad: 1, Stride: 1	32
MAX POOL	Kernel: 3x3, Stride: 1	
FC		10

Table 1. CNN 1

This network had an accuracy of 72.91% after 20K epochs. I had tried different kernel sizes of 3x3, 5x5 and 7x7, and zero padding values of 1,2 and 3. Kernel size of 5x5 for convolution layers and 3x3 for max pooling layers gave me the best accuracy.

I tried adding ReLU layers at the end of each max pooling layers in the above network. This improved accuracy by a decent margin - 75.81%. Thus, proving that ReLU units indeed decrease training time, as the network was able to achieve higher accuracy with same number of epochs.

After this, I tried the 6 layer network by Krizhevsky and also added ReLU layers, as described in table 2.

Layer Type	Parameters	Outputs
CONV	Kernel: 5x5, Pad: 2, Stride: 1	32 + ReLU
MAX POOL	Kernel: 3x3, Stride: 1	
CONV	Kernel: 5x5, Pad: 1, Stride: 1	32 + ReLU
AVG POOL	Kernel: 3x3, Stride: 1	
CONV	Kernel: 5x5, Pad: 1, Stride: 1	64 + ReLU
AVG POOL	Kernel: 3x3, Stride: 2	
FC		10

Table 2. CNN 2

This layer makes use of 1 max pooling layer and 2 average pooling layers. It has an accuracy of 77.39%. I tried adding dropout layers to the above network to try to improve accuracy. I modified the network to the following:

Adding dropout further improved the accuracy to 79.57%. I tried other dropout ratios like 0.10, 0.20 and 0.50. 0.25 dropout gave the best performance. The filters of the three convolution layers are shown in figures 2, 3, 4.

As one can see from the filter images, the earlier stage filters activate on large image patches whereas deeper filters learn to activate on small/specific locations in the image. The intuition is that with successive layers, filters learn to focus on specific small features within an image.

Layer Type	Parameters	Outputs
CONV	Kernel: 5x5, Pad: 2, Stride: 1	32 + ReLU
MAX POOL	Kernel: 3x3, Stride: 1	
CONV	Kernel: 5x5, Pad: 1, Stride: 1	32 + ReLU
AVG POOL	Kernel: 3x3, Stride: 1	
DROPOUT	Dropout ratio: 0.25	
CONV	Kernel: 5x5, Pad: 1, Stride: 1	64 + ReLU
AVG POOL	Kernel: 3x3, Stride: 2	
DROPOUT	Dropout ratio: 0.25	
FC		10

Table 3. CNN 3

Finally I compared my results to the reference CIFAR-10 classifier included in Caffe. This net makes use of 2 normalization layers in addition to the Krizhevsky network. This net achieves an accuracy of 77.48%.

15. Results

I trained 4 different networks in Caffe, successively iterating and improving the accuracy. The results are summarised in table 4. My final 6 layer model using relu and dropout was able to beat the reference model included in Caffe by a small margin. I observed that adding higher number layers improves performance at the cost of training time. But adding too many layers can also have the potential downside of overfitting to the training data and performing badly on the unseen test data. Dropout can help mitigate this to a certain extent.

Model	Accuracy
4 layer	72.91%
4 layer + ReLU	72.91%
6 layer + ReLU, Avg Pooling	77.39%
6 layer + Dropout, ReLU, Avg Pooling	**79.57%**
Caffe Model + ReLU, Avg Pooling, Normalization	77.48%

Table 4. Results

16. Conclusion and Future Scope

In this project, I got to learn how to prototype models using the powerful Caffe framework and achieved decent results on the CIFAR-10 dataset. If I had more time, I would have tried to the train the models for much higher number of epochs. The state of the art models are trained for 150000+ epochs. Ensemble models are also known to improve performance. Data augmentation (mirroring, rotations, affine transformations) is another technique used to increase the size of training data and thus improve performance.

In the future, I would like to train deeper networks for higher number of epochs and also explore how to use en-

semble models and data augmentation techniques for improving performance.

17. Acknowledgements

I would like to thank Professor Zhuowen Tu and 2Lab for providing access to their lab server with which I was able to use GPUs to speed up the training in Caffe.

References

[1] *Chord: A Scalable Peer-toPeer Lookup Protocol for Internet Applications* Ion Stoica, Robert Morris, David Liben Nowell, David R.Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan

[2] *Learning a similarity metric discriminatively with Application to Face Verification* Sumith Chopra, Raia Hadsell, Yann Le Cunn

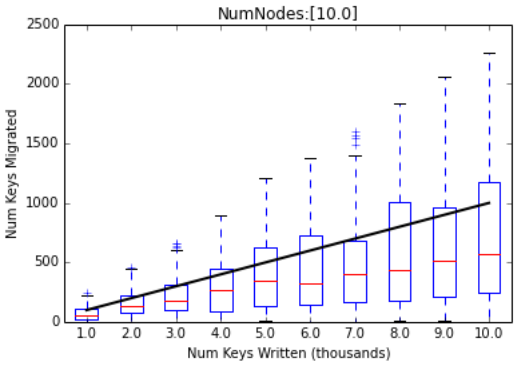
[3] *Dimensionality Reduction By Learning an Invariant Mapping* Raia Hadsell, Sumith Chopra, Yann Le Cunn

[4] *A similarity-based neural network for facial expression analysis* Kenji Suzuki , Hiroshi Yamada , Shuji Hashimoto

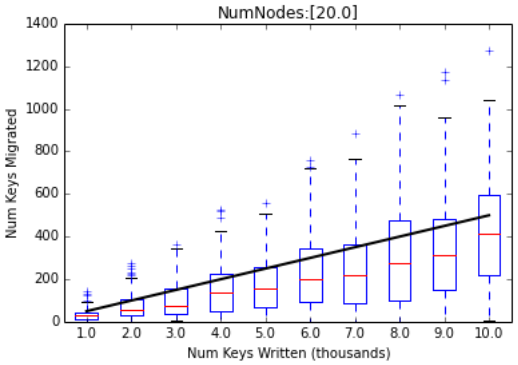
[5] *NimStim Face Stimulus Set* The MacArthur Foundation Research Network on Early Experience and Brain Development

[6] *Imagenet Classification with Deep Convolutional Neural Networks* Alex Krizhevsky, Ilya Sutsekver, Geoffrey E.Hinton

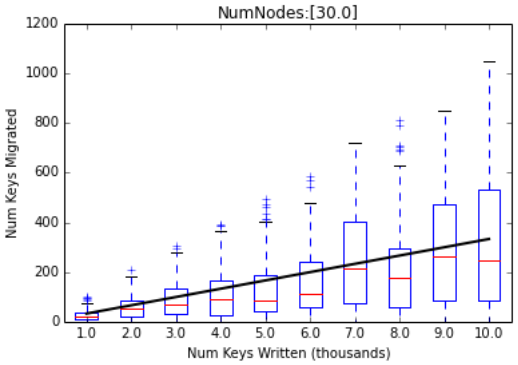
[7] *Caffe: Convolutional Architecture for Fast Feature Embedding* Jia, Yangqing and Shelhamer, Evan and Donahue, Jeff and Karayev, Sergey and Long, Jonathan and Girshick, Ross and Guadarrama, Sergio and Darrell, Trevor



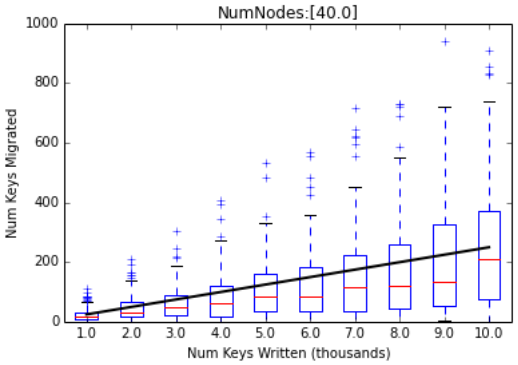
(a) Migration with 10 Nodes



(b) Migration with 20 Nodes



(c) Migration with 30 Nodes



(d) Migration with 40 Nodes

Figure 4. Migration Of Keys On Join

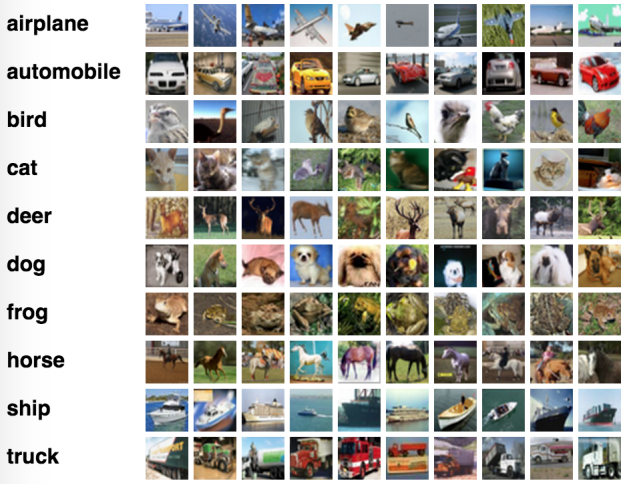


Figure 5. 10 random images from 10 classes of the CIFAR-10 dataset

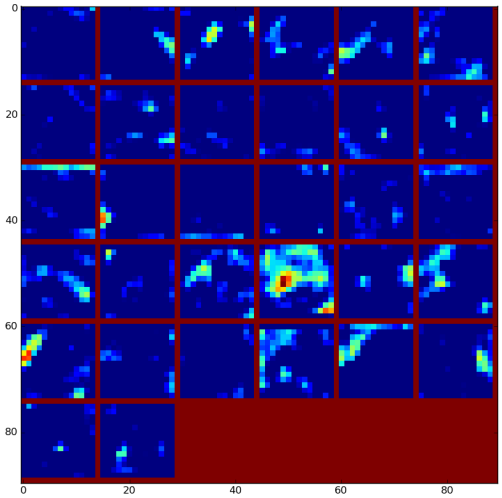


Figure 7. Filters in second convolution layer for CNN in table 3

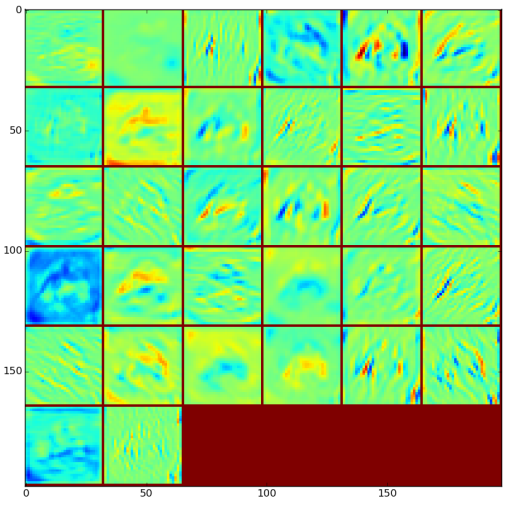


Figure 6. Filters in first convolution layer for CNN in table 3

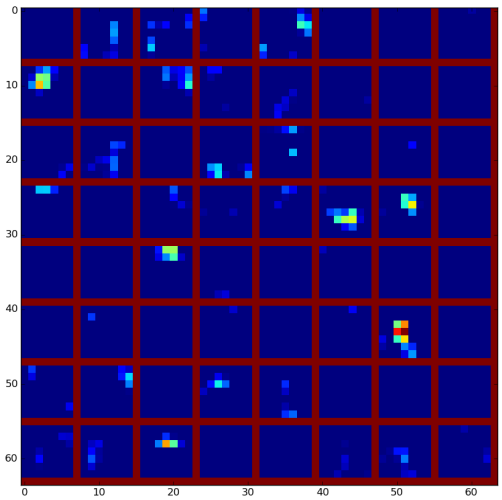


Figure 8. Filters in second convolution layer for CNN in table 3