Advances in One Shot Learning

A Thesis Submitted to the Department of Computer Science and Communications Engineering, the Graduate School of Fundamental Science and Engineering of Waseda University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

Submission Date: July 18th, 2020

Ravi Jain (5118FG20-1)

Advisor: Prof. Hiroshi Watanabe

Research guidance: Research on Audiovisual Information Processing

In this thesis work, we go through several advances in computer vision, and demonstrate techniques for enabling computers to learn with a limited number of examples.

This work is divided into 7 sections, the first section is introduction, the second section, describes the use of Stand-Alone Self Attention, along with a modification to its formulation, applied for few shot image classification. The third section describes a novel way to drop units in a neural network, the fourth section describes a binary neural network-based system for one shot learning.

In the fifth section, we demonstrate certain statistics related to human brain functioning, and how these would help to create mathematical models of the same, we conclude in the sixth section. In the final section, we point out certain ethical concerns associated with computers beginning to learn as humans learn.

Table of Contents

1. Introduction

The field of deep learning, appears to be growing more and more popular in 2020, as we see computers doing tasks that only humans could do a few years ago, like, translating languages, giving us a smart reply on our mails, classifying images, text, and so on, but is all of this excitement only so recent, or has it been going on for a much longer duration.

Could someone have predicted that computers would be able to do these tasks in 2020, even 70 years ago?

It turns out that, people did predict that computers would be able to do such tasks, way back around 1950s (A. M. Turing 1950), but computers of 1950s were not strong enough, so one could predict, but not actually see the applications, the same pattern applies for the year 2020 also, one can predict certain aspects of what all computers would be able to do in the upcoming years, but we will have to wait to see the future unfold.

In this research work, we demonstrate, one such task, that is the task of learning with few examples, humans can learn with a few examples, an infant neocortex, or a 3-4 year old human (in 2020), when shown one picture of an object, along with describing its name/class, would be able to classify the object, when shown with some variations. One does not need to show the 3-4-year-old human, thousands of images of an object, before they will be able to classify the same object when shown variations. Humans are even able to accomplish tasks like complete the picture when shown part of it, one common example is, being able to tell Albert Einstein's name, when only shown his tongue. Humans are also able to do tasks like spot the difference between two images, finding patterns that are in one image, but not in the other. Computers in 2020, need to be trained on datasets, with 1000s of images, for them to learn to classify, that is during training, they are shown 1000s of images of objects, and during the test phase, when shown with new images of the same objects, they are expected to be able to classify them correctly. But can computers learn to classify objects just like humans do, that is after creating a model of the environment, when shown with one image of a novel class, they would be able to classify its variations.

In order to answer this question, we need to understand, how we are doing these tasks, that is how is a human brain, a biological neocortex able to accomplish one shot learning. Certain image classification techniques, in 2020, are based on convolution based systems, in this research work we demonstrate certain limitations

of such systems, show results of applying attention based mechanisms on one shot classification, and dive deep into how do we learn patterns, along with going through a binary neural network based approach for one shot learning based on papers written around the 1950s.

2. Few shot Image Classification

2.1 Problem Statement

We describe how few shot learning differs from standard neural network training. In standard neural network training, we use 1000s of images in our training set, and these images belong to associated classes. We update the parameters of our neural network during the training phase, by asking it to predict classes of these 1000s of images, and using these updated parameters, we expect our neural network to predict the class of an image correctly during test phase. In standard neural network training, the classes seen during test phase are the same that our neural network has seen during training phase.

So, what is the difference in few shot learning?, first let us see how do humans do few shot learning, when shown one image of an object with its class, we are expected that we will be able to label a variation of that image correctly, but this is only after we have seen a lot of objects, that is we have created a model of the world.

In the current work on one shot learning, the techniques are given abundant training examples for the base classes (similar to the concept of creating a model of the world) this leads to having updated parameters, and our neural network is expected to learn to recognize novel classes (only seen during test, not shown during training) with a limited number of labeled examples (similar to us being able to classify variations after seeing an image once). Ideally one would want, the neural network to be able to learn patterns from one image, along with its class, for images belonging to novel class, and be able to classify variations of images belonging to the same novel class correctly.

Recent work uses 1 shot 5 way, and 5 shot 5 way approach, for K shot N way, we want our neural network to be able to classify an image into one of the N candidate classes. That is in the case of 1 shot 5 way, for novel classes (only during meta-testing phase), there is one image for each of the 5 classes in the support set (or training set during meta-testing phase), and 5 shot 5 way means we have 5 images for each of the 5 classes in the support set (or training set during meta-testing phase). In the query set (or test set during meta-test phase), it does not matter how many images we have per class, the K shot reference is only for number of examples per class in the support set. In our experiments we use a Baseline++ approach (Chen, et al. 2019), so our support set and query set are only during the meta-test phase (that is for the novel classes), while training phase is similar to standard neural network training procedure (that is for the base classes).

Our work describes use of Attention based approach instead of using convolutional neural networks, along with modifications to the formulation of Stand-Alone Self Attention (Ramachandran, et al. 2019), and describe results obtained.

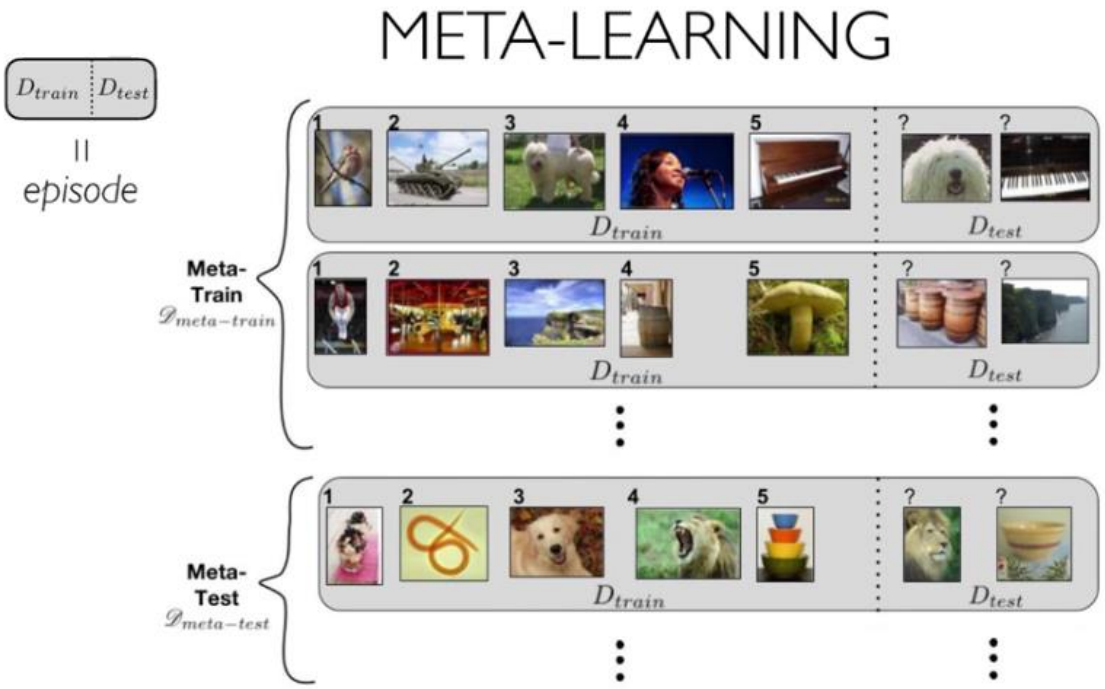


Figure 1 - An image describing this form of learning (Ravi and Larochelle 2017)

But in the Baseline++ approach, our episodes during meta-training stage follow standard neural network training procedure, and are not divided into support set (D_{train} in Figure 1) and query set (D_{test} in Figure 1).

In the next section we describe Baseline++ approach in more detail, and in the third section we describe Stand Alone Self Attention, and show a modification to its formulation applied on the task of few shot image classification.

2.2 Baseline++ Approach Overview

In the Baseline++ approach, during the training stage, we have a feature extractor f_φ, with network parameters, φ and the classifier, C, with parameters as a weight matrix, $W_b \in R^{d\ X\ c}$, where the dimension of the reduced feature (final feat dim) is d, and the number of output classes is c (the n in n_way). The task is to minimize Cross Entropy Loss using the training examples in the base classes. The classifier consists of a cosine distance, followed by a softmax function. In the baseline approach, the classifier consists of a linear layer, followed by a softmax function.

(In our experiments we found Baseline++ to give significantly better results, so most of the accuracies are reported using the same).

This use of cosine distance-based classifier has been described in (Chen, et al. 2019). We do not alter it.

During the test, that is meta test stage, when given novel classes, the pre trained network parameters φ are kept fixed, in our feature extractor f_{φ}, and the weights associated with the classifier are updated by minimizing the Cross Entropy Loss, using the support set (that is the support set provided during meta test phase), of novel classes.

The model is provided abundant base classes labeled data during the meta training stage, and few novel classes data during the meta test stage, and is expected to give correct predictions, for the query set during the meta test stage.

In simpler terms, the model parameters are learnt during the meta-training stage, and are kept frozen during the meta-test stage, only the parameters associated with the classifier are updated even for the support set in the meta-test stage, and accuracy is computed for the query set in the meta-test stage.

2.3 Stand Alone Self Attention Overview

The whole concept of Stand-Alone Self Attention, is based on one principle, that the representation of pixels depends on what all pixels they are surrounded by, and this attention-based mechanism is a better way to mathematically model this concept as compared to convolutions. Each of the neighboring pixel has an impact on the representation of a particular pixel.

One naive example, could be seen with an image of a green circle, when shown with just an image of a green circle, with no other pattern surrounding it, one would call it just a green circle image, but when the same image is a part of a food product, then one would start considering it as an indication that the food is made up of vegetables. If the same image was a part of a traffic lighting system, then it would be considered as a green traffic light. If the same image was part of a company's logo, then it would have a different meaning. That is all of the neighboring pixels are changing the meaning of a particular pattern. This same concept applies for text also, so, when given a sentence like, "The animal didn't cross the street because it was too tired", the word 'it', means 'the animal', while for the sentence, "The animal didn't cross the street because it was too wide", the word 'it', means 'the street'. (Vaswani, et al. 2017)

So, in the case of Stand-Alone Self attention, the way to solve this problem is done by a Query, Key and Value based system, along with considering relative distance between pixels. Each of the pixels in the Query have an initial random representation, a linear transformation is applied to them.

The same applies for Key and Value, both of them are blocks surrounding Query pixels, each of the pixels in these tensors also have an initial random representation, and a linear transformation is applied to them.

The way to consider impact of each key pixel on the query pixel is described by a Matrix Multiplication in the original paper. After this, the next step is to apply a Softmax on the output obtained, a Softmax, brings the sum of output to one, so each of the outputs obtained after applying Softmax, would represent how much does a particular pixel in key, has an influence on the query pixel. These outputs are then multiplied with the Value tensor, and the result is added, to get a final learnt representation of our query pixel.

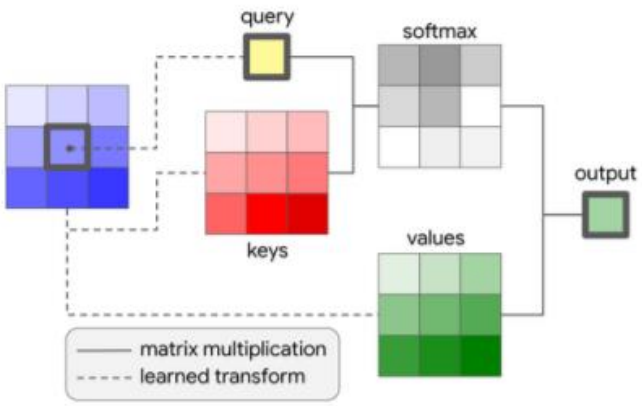


Figure 3: An example of a local attention layer over spatial extent of $k = 3$.

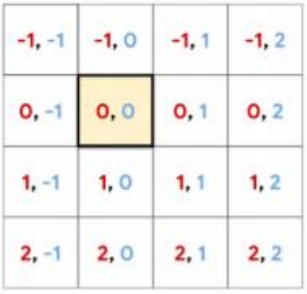


Figure 4: An example of relative distance computation. The relative distances are computed with respect to the position of the highlighted pixel. The format of distances is *row offset*, *column offset*.

Figure 2 – Demonstration of Stand-Alone self-Attention as described in the original paper (Ramachandran, et al. 2019)

For our input $x \in R^{h \times w \times d_{in}}$ with height h, width w and input channels d_{in}, with a particular pixel, $x_{ij} \in R^{d_{in}}$, we extract a block of pixels, in positions ab belonging to the nearby block of our input pixel x_{ij}. The pixel output, is denoted by $y_{ij} \in R^{d_{out}}$

Our Query, Key and Value matrices are denoted by,

$$q_{ij} = W_Q x_{ij}$$

$$k_{ij} = W_K x_{ij}$$

$$v_{ij} = W_V x_{ij}$$

Where W denotes our weight matrix.

Relative distance is denoted by,

$$r_{a-i,b-j}$$

These are embeddings of shape half of the number of out channels as described in the original paper.

2.4 Proposed Approach

The way we apply this technique, differs a bit from the way it is described in the original paper, in this section we describe the shape of the tensors in the way we apply this technique, that leads to an improvement in the accuracy obtained on few shot image classification experiment.

One example, of applying the technique we use along with the shape obtained is described below,

Input image -> [1, 3, 84, 84] (1 image, 3 channel, CUB images are 84x84 size)

We use a 1x1 Conv2d with, out channels as 64 -> [1, 64, 84, 84] (64 channel 84x84)

Query shape -> [1, 64, 84, 84, 1] (1x1 patch)

Key shape -> [1, 64, 84, 84, 9] (3x3 patch, flattened)

Value shape -> [1, 64, 84, 84, 9] (3x3 patch, flattened)

In our approach we take a scalar dot product between Query and Key, so the shape obtained after this step is,

Scalar dot product between Query and Key -> [1, 64, 84, 84, 9]

After applying Softmax over the final dimension -> [1, 64, 84, 84, 9]

And finally we use a einsum operation with Value, to obtain a final output tensor -> [1, 64, 84, 84] (we do the dot product with Value along with summation, using the

einsum operator, this leads to an improvement in accuracy, as compared to other techniques that we used)

For relative distance computation,

Input image shape is, -> [1, 64, 84, 84]

Split the channels into half and get an unfolded tensor -> [1, 32, 84, 84, 3, 3] (2 such tensors)

Add a [3, 1] embedding to the first tensor and a [1, 3] shape embedding to the second tensor. We do not alter the relative positioning technique.

Concatenate the results to get a 64-channel image back.

One notion that we thought of is the notion of a weighted impact, that is each pixel will have a weighted impact on the neighboring pixel, to implement this, we add a notion of a Priority matrix, along with the Key, so we even do a scalar dot product with this Priority matrix. The results obtained after using this priority modification are included in the next section, so along with using a scalar dot product between Query and Key, we also use a scalar dot product with Priority matrix.

Another modification that we made is to use a scalar dot product between Query and addition between Query and Key matrix, that is $q_{ij} * (q_{ij} + k_{ab})$, this additive similarity modification gives an improved accuracy.

The eventual formulation becomes

With priority matrix,

$$y_{ij} = \sum_{a,b \in N_k(i,j)} softmax_{ab}\left(q_{ij} * \left(q_{ij} + k_{ab}\right) * p_{ab} + q_{ij} * r_{a-i,b-j}\right) v_{ab}$$

Equation 1 – Proposed Stand-Alone Self Attention with Additive Similarity and Priority Matrix

Without priority matrix,

$$y_{ij} = \sum_{a,b \in N_k(i,j)} softmax_{ab}\left(q_{ij} * \left(q_{ij} + k_{ab}\right) + q_{ij} * r_{a-i,b-j}\right) v_{ab}$$

Equation 2 – Proposed Stand-Alone Self Attention with Additive Similarity but without Priority Matrix

Where the summation and dot product with Value matrix part is done by einsum operator.

Reasoning for using an additive similarity,

If we have two pixels, black and white, and want to represent each combination of them differently.

`black`	`(Q)`
`white`	`(K)`
`black white`	`(QK)`
`White black`	`(KQ)`
`black black`	`(QQ)`
`white white`	`(KK)`

Table 1 – Representation of combination of pixels, denoted by Q and K

A scalar dot product between query and key would give same result for

`black white` and `white black`

Other options could be, subtract key from query and then do a scalar dot product with query, but then,

```
black black
white white
```

would give same result

or completely avoid a dot product, only do an addition between query and key, that would mean, either

```
black white
white black
```

would give same result

or if we do subtraction between query and key, then,

```
white white
black black
```

would give same result

or

consider only query / only key, then all four combinations would give the same result, other options would require higher computation, like concatenating Query, and Key together.

Therefore, we go with a scalar dot product with additive similarity.

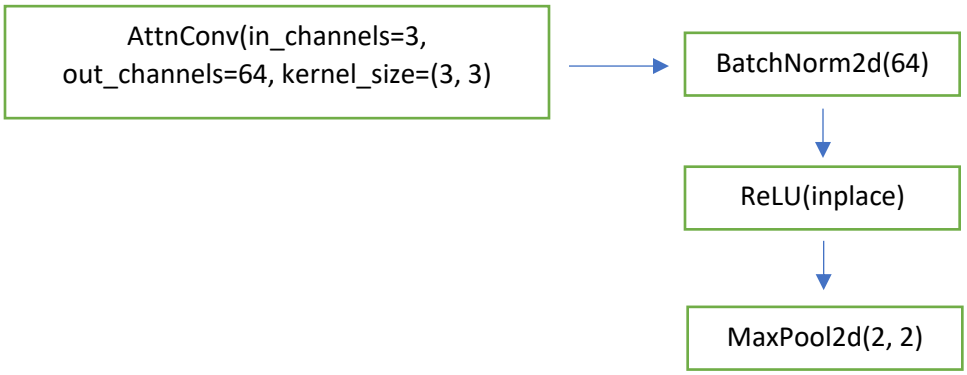
2.5 Dataset

We use CUB-200-2011 dataset (Welinder, et al. 2010) The CUB dataset contains 200 classes and 11,788 images in total. We split the dataset into 100 base, 50 validation, and 50 novel classes, the change that we make here is that we do not use a random split, and the split is the same for all the experiments.

Figure 3 - An image of CUB dataset (Welinder, et al. 2010)

2.6 Architecture, Training Details

The architecture we use is,



- Where the AttnConv denotes Stand Alone Self Attention layer
- AttnConv4, denotes the above architecture repeated four times
- We use MaxPool only for first four layers.
- Further training details are described in Table 2.

Dataset	CUB
Loss function	CrossEntropyLoss (with LabelSmoothing during test phase)
Learning rate	0.001
Optimizer	Adam (SGD during test phase)
Epochs trained for	400
Loss Type	distLinear
Technique used for all experiments	Baseline++
Training Augmentation	Enabled
Shot, Ways	5 shot 5 way, 1 shot 5 way
AttnConv4 denotes (same for AttnConv6, repeated 6 times) (we use MaxPool only for first four layers)	Stand Alone Self Attention -> BatchNorm2d -> ReLU -> MaxPool2d(2)
Kernel size	3x3
Outdim	64
Final Feat dim	1600

Table 2 – Training details

2.7 Experiment Results

- 5 shot 5-way classification with Baseline++ technique, kernel size 3 (increasing the kernel size, results in gpu crash)

- Time to train would be around 38000 to 42000 seconds when using einsum (400 epochs)
- *highest accuracy obtained after fine tuning is reported
- Higher means compared to the accuracies mentioned in (Chen, et al. 2019) (described in Table 6)

Dataset	**Backbone**	**Accuracy**	**Dropout**	**Formulation**
CUB	AttnConv6	67.01% +- 0.72%	0.3 after every AttnConv block	Scalar Dot Product
CUB	AttnConv6	69.52% +- 0.73%	0.2 after every AttnConv block	Scalar Dot Product
CUB	AttnConv6	82.75% +- 0.58% (higher)	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity
CUB	AttnConv6	58.25% +- 0.77%	Only once at the end (0.3)	Matrix multiplication
CUB	AttnConv6 with LPPool instead of MaxPool	78.50% +- 0.67%	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity
CUB	AttnConv4	75.69% +- 0.70%	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity along with Priority Matrix

Table 3 – Results on 5 shot 5 way, using Baseline++, on CUB dataset

- We see here, a six layered Stand-Alone Self Attention neural network using scalar dot product with additive similarity gives a high accuracy.
- Adding dropout multiple times to increase the sparsity, did not give an improvement in accuracy. (at the end means before passing the reduced feature through the classifier)
- Use of LPPool instead of MaxPool also leads to a reduction in the accuracy obtained.

- 1 shot 5 way classification with Baseline++ technique results are described in Table 4 and Table 5 , with kernel size 3 (increasing the kernel size, results in gpu crash)

Dataset	**Backbone**	**Accuracy**	**Drop Out**	**Formulation**	**Time to train (sec)**	**Technique**
CUB	AttnConv4	61.84% +- 0.88% (higher)	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity	37277	Baseline++
CUB	AttnConv6	66.12% +- 0.94% (higher)	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity	38262	Baseline++
CUB	AttnConv6	56.09% +- 0.82%	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity along with Priority Matrix	41789	Baseline++
CUB	AttnConv4	56.45% +- 0.89%	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity with Priority Matrix initialized with kaiming normal	41684	Baseline++
CUB	AttnConv4	59.56% +- 0.88%	0.3	Scalar Dot Product	39120	Baseline++
CUB	AttnConv4	45.84% +- 0.77%	0.3	Scalar Dot Product	38930	Baseline

Table 4 - 1 shot 5 way classification with Baseline++ technique results

- We see here the longer training time, due to the use of einsum operator, for carrying these experiments, as compared to a standard ConvNet.
- For both four and six layered neural networks, Stand Alone Self Attention with scalar dot product with additive similarity gives higher accuracy.

- Use of Baseline approach, leads to a significant reduction in accuracy obtained.

Dataset	Backbone	Accuracy	Dropout	Formulation	Time to train (seconds)	Technique
CUB	AttnConv4, without batch norm	58.08% +- 0.89%	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity	36397.97	Baseline++
CUB	AttnConv4, replacing Maxpool with LPpool	59.42% +- 0.86%	Only once at the end (0.3)	Scalar Dot Product with Additive Similarity	37139.27	Baseline++
CUB	AttnConv4, (Maxpool, with batch norm)	61.70% +- 0.92% (higher)	increase dropout to 0.5 (Only once at the end)	Scalar Dot Product with Additive Similarity	36583.36	Baseline++
CUB	AttnStem4	61.99% +- 0.90% (higher)	increase dropout to 0.5 (Only once at the end)	AttnStem formulation, no modification	64541.34	Baseline++

Table 5 - 1 shot 5 way classification with Baseline++ technique results continued

- Increasing the DropOut rate as long as it is applied once leads to an improvement in the accuracy.
- Removal of batch norm layer, replacing MaxPool with LPPool did not give an improvement for 1 shot 5 way also.

Dataset	Shot, Way	Conv4	Conv6	Technique
CUB	1 Shot 5 Way	60.53±0.83	66.00±0.89	Baseline++
CUB	5 Shot 5 Way	79.34±0.61	82.02±0.55	Baseline++

Table 6 – Accuracies reported in the paper CloserLookFewShot (Chen, et al. 2019)

We demonstrate the application of Stand-Alone Self Attention along with a few modifications to its formulation applied in the Few shot classification domain.

One problem with using Stand Alone Self Attention is the longer training time as compared to standard conv net. Thus, it becomes difficult to carry experiment using different Few Shot classification techniques.

Another problem that we see here is dropout rate, and how to decide the dropout rate, as we see varying the DropOut rate results in varied accuracies, in the next section, we go through a technique discussing this problem in detail.

3. DropSame

In this section, we describe, a novel way to drop units in a neural network, named as DropSame, along with describing the limitations of using a floating point-based parameter system, and why we think that a binary neural network is a better way to make computers learn as humans learn. This section is divided into following sub sections,

3.1 Problem Statement

A common technique that is used in neural network training, is known as DropOut (Srivastava, et al. 2014), which zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Another variant is known as DropConnect (Wan, et al. 2013) in which instead the parameters are dropped at random. Another common technique is DropBlock, (Ghiasi, Lin and Le 2018) in which entire blocks of elements is zeroed out. But all of these techniques require specifying a manual hyperparameter input, that is in the case of DropOut, one needs to specify a DropOut rate, specifying the probability of an element to be zeroed out, and there is no specific way to decide what this probability should be, so, is there a way that we drop elements without having to specify a manual hyperparameter input?

3.2 Proposed Approach

Let us describe a technique that does not require specifying a rate at which elements should be dropped, on one input tensor, and then we will show the results of applying the same on CIFAR10 dataset.

i. We start with a input tensor, with height h, width w and input channels d_{in}, for simplicity let us assume this tensor to be a single channeled 5x5 image, we extract all $n \times m$ sized patches from this tensor, the size of a patch could be varied, in the example we extract 3x3 sized patches from our input tensor.
ii. After this, we will have certain patches, extracted from our input tensor, we also specify a threshold, which is a parameter, our neural network would be learning this threshold.
iii. For all of these patches, if the absolute difference between values is less than absolute value of threshold, then we drop one of them, and if a value is zeroed out in any of the patches, then we zero it out in the final output tensor and keep rest of the values same.

iv. We apply this technique in two directions, that is forward and reverse, so when applied in the forward direction, if the difference between values is less than absolute value of threshold, then the second element would be dropped, while when applied in the reverse order, the first element would be dropped.

v. After this we do an OR operation, on the output obtained by applying this technique in forward and reverse directions, and get an output tensor, with the same shape of the input tensor.

One example, of an input tensor, along with extracted patches, and output tensor is shown below.

- Input, shape 5x5, single channel

1.1763	-0.0154	0.5744	0.9928	0.6800
-0.9231	-0.3832	0.4796	-0.2925	1.3877
2.4501	-2.6819	0.2484	0.7291	0.3478
0.8366	1.4944	1.3979	1.5541	-1.2613
0.8092	-1.2060	-1.9764	-0.7072	-0.6003

Table 7 – Input to DropSame

- Threshold (a parameter, so it requires gradient)

tensor([0.4737], device='cuda:0', requires_grad=True)

- After unfolding to get patches, each of these patches (a row) has nine elements

1.1763	-0.0154	0.5744	-0.9231	-0.3832	0.4796	2.4501	-2.6819	0.2484
-0.0154	0.5744	0.9928	-0.3832	0.4796	-0.2925	-2.6819	0.2484	0.7291
0.5744	0.9928	0.6800	0.4796	-0.2925	1.3877	0.2484	0.7291	0.3478
-0.9231	-0.3832	0.4796	2.4501	-2.6819	0.2484	0.8366	1.4944	1.3979
-0.3832	0.4796	-0.2925	-2.6819	0.2484	0.7291	1.4944	1.3979	1.5541
0.4796	-0.2925	1.3877	0.2484	0.7291	0.3478	1.3979	1.5541	-1.2613
2.4501	-2.6819	0.2484	0.8366	1.4944	1.3979	0.8092	-1.2060	-1.9764
-2.6819	0.2484	0.7291	1.4944	1.3979	1.5541	-1.2060	-1.9764	-0.7072
0.2484	0.7291	0.3478	1.3979	1.5541	-1.2613	-1.9764	-0.7072	-0.6003

Table 8 – Unfolded output

- Zeroing the values in the forward direction (values in a patch are compared). This step takes longer time, as we compare each element in a patch, with each other element. A way to reduce the time required to carry this step is a research question.

1.1763	-0.0154	0.5744	-0.9231	0.0000	0.0000	2.4501	-2.6819	0.0000
-0.0154	0.5744	0.0000	0.0000	0.0000	0.0000	-2.6819	0.0000	0.0000
0.5744	0.0000	0.0000	0.0000	-0.2925	1.3877	0.0000	0.0000	0.0000
-0.9231	-0.3832	0.4796	2.4501	-2.6819	0.0000	0.0000	1.4944	0.0000
-0.3832	0.4796	0.0000	-2.6819	0.0000	0.0000	1.4944	0.0000	0.0000
0.4796,	-0.2925	1.3877	0.0000	0.0000	0.0000	0.0000	0.0000	-1.2613
2.4501	-2.6819	0.2484	0.8366	1.4944	0.0000	0.0000	-1.2060	-1.9764
-2.6819	0.2484	0.7291	1.4944	0.0000	0.0000	-1.2060	-1.9764	-0.7072
0.2484	0.7291	0.0000	1.3979	0.0000	-1.2613	-1.9764	-0.7072	0.0000

Table 9 – Unfolded output after zeroing out elements in the forward direction

- After folding the above tensor (for each 3x3 patch, the difference between any two values exceeds the threshold). Notice that, the way we fold the above tensor, is that we keep a value zeroed out, if it was zeroed in any one of the above blocks, the PyTorch version of fold, does not allow for this, so we had to implement a custom mask, to implement this.

1.1763	-0.0154	0.5744	0.0000	0.0000
-0.9231	0.0000	0.0000	0.0000	1.3877
2.4501	-2.6819	0.0000	0.0000	0.0000
0.0000	1.4944	0.0000	0.0000	-1.2613
0.0000	-1.2060	-1.9764	-0.7072	0.0000

Table 10 – Folded output after zeroing out elements in the forward direction

- Zeroing the values in the reverse direction, this step once again takes longer time, similar to zeroing the values in the forward direction.

1.1763	0.0000	0.0000	-0.9231	-0.3832	0.0000	2.4501	-2.6819	0.2484
0.0000	0.0000	0.0000	0.0000	0.0000	-0.2925	-2.6819	0.2484	0.7291
0.0000	0.0000	0.0000	0.0000	-0.2925	1.3877	0.0000	0.0000	0.3478
-0.9231	-0.3832	0.0000	2.4501	-2.6819	0.2484	0.8366	0.0000	1.3979
0.0000	0.0000	-0.2925	-2.6819	0.2484	0.7291	0.0000	0.0000	1.5541
0.0000	-0.2925	0.0000	0.0000	0.0000	0.3478	0.0000	1.5541	-1.2613
2.4501	-2.6819	0.2484	0.0000	0.0000	1.3979	0.8092	-1.2060	-1.9764
-2.6819	0.2484	0.7291	0.0000	0.0000	1.5541	-1.2060	-1.9764	-0.7072
0.0000	0.0000	0.3478	0.0000	1.5541	-1.2613	-1.9764	0.0000	-0.6003

Table 11 – Unfolded output after zeroing out elements in the reverse direction

- After folding the above tensor, we follow the same masking-based folding here also.

1.1763	0.0000	0.0000	0.0000	0.0000
-0.9231	0.0000	0.0000	-0.2925	0.0000
2.4501	-2.6819	0.0000	0.0000	0.3478
0.0000	0.0000	0.0000	1.5541	-1.2613
0.8092	-1.2060	-1.9764	0.0000	-0.6003

Table 12 – Folded output after zeroing out elements in the reverse direction

- Final output obtained (OR operation on the folded tensors)

1.1763	-0.0154	0.5744	0.0000	0.0000
-0.9231	0.0000	0.0000	-0.2925	1.3877
2.4501	-2.6819	0.0000	0.0000	0.3478
0.0000	1.4944	0.0000	1.5541	-1.2613
0.8092	-1.2060	-1.9764	-0.7072	-0.6003

Table 13 – Final output obtained

We make sure that the backpropagation works properly when using this technique, and there are no null values during training.

3.3 Dataset

The technique is more computation intensive as compared to DropOut, so it takes much longer to use this on Few shot image classification, so we do standard image classification using CIFAR10 dataset.

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class.

There are 50000 training images and 10000 test images.

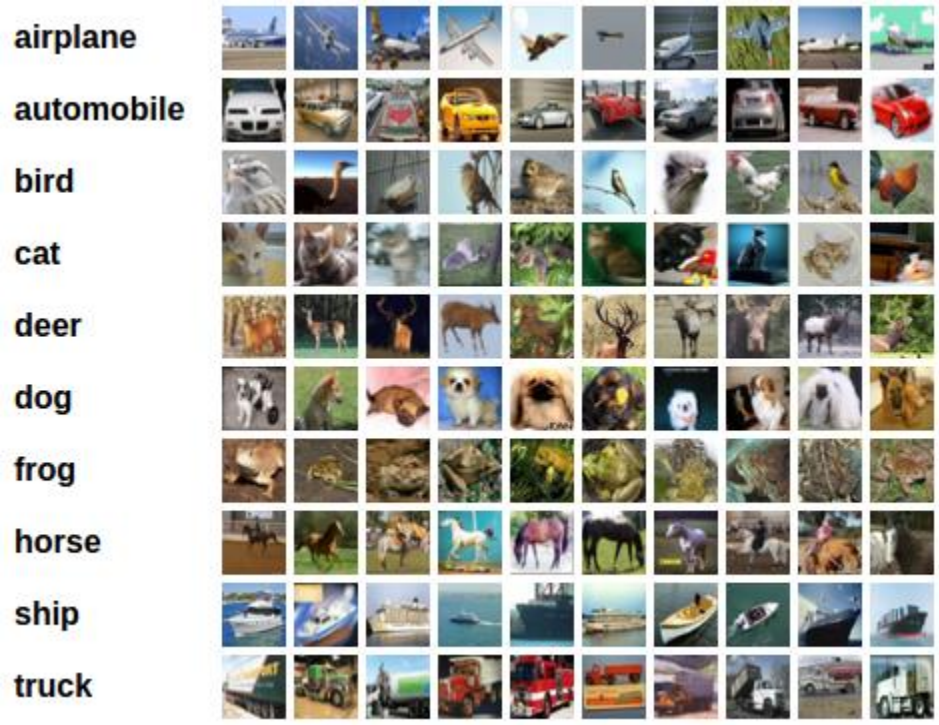
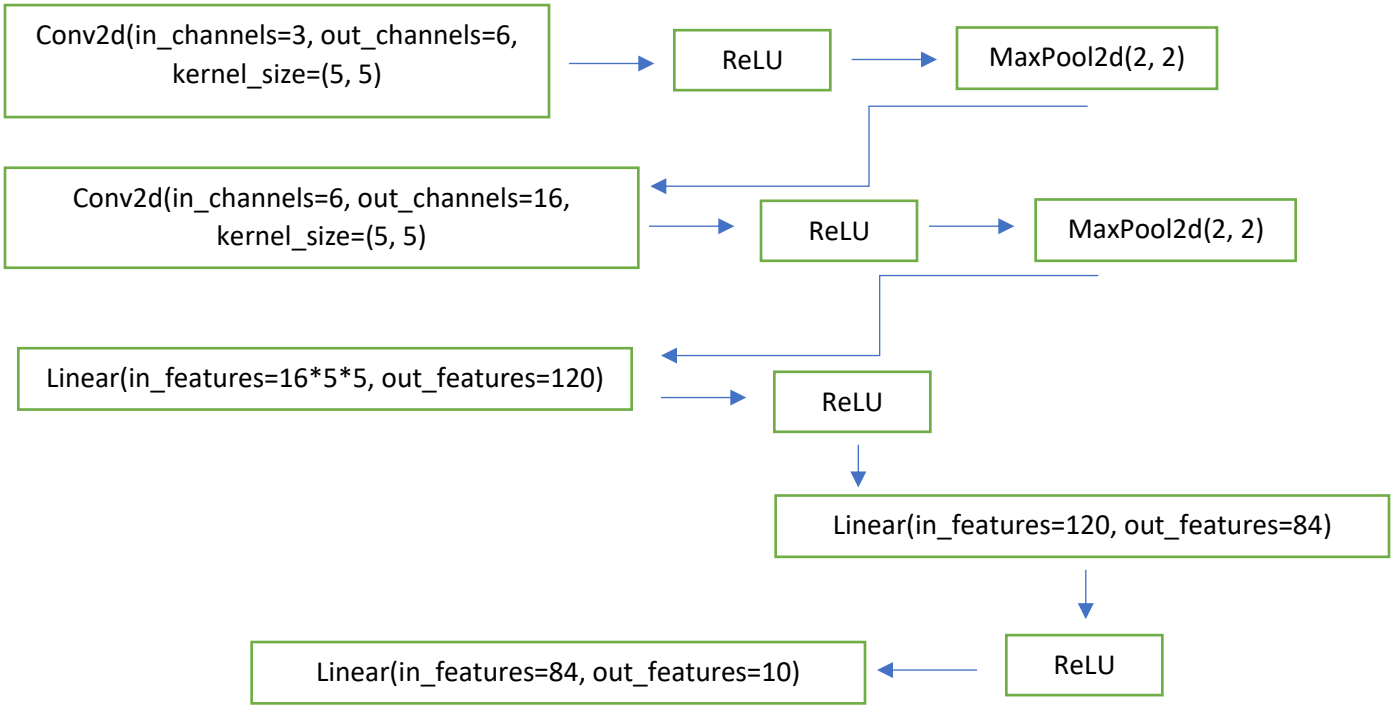


Figure 4 – An image of CIFAR10 dataset

3.4 Architecture, Training Details

The architecture details are as follows –



Training details are as follows -

Loss function	CrossEntropyLoss
Optimizer	SGD
Learning Rate	0.001
Momentum	0.9
Batch Size	4
Epochs	2

Table 14 - Training Details for DropSame Experiment

In the next section we show results obtained after using DropSame with this architecture.

3.5 Experiment Results

The results obtained are described in following tables.

Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	On 10000 test images
50 %	62 %	35 %	30 %	42 %	48 %	59 %	44 %	50 %	58 %	48 %

Table 15 – Using DropSame (3, 3) after second MaxPool with ReLU

Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	On 10000 test images
51 %	57 %	25 %	23 %	58 %	47 %	55 %	52 %	68 %	56 %	49 %

Table 16 – Using DropSame (3, 3) after second MaxPool without use of first two ReLUs

Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	On 10000 test images
67 %	47 %	33 %	2 %	1 %	42 %	3 %	62 %	26 %	46 %	33 %

Table 17 – Using DropSame (3, 3) after second MaxPool without use of first two ReLUs, disable DropSame at test time

We found that disabling DropSame at test time, gives a reduction in accuracy, as compared to using DropSame, both during training and at test time. So, we prefer not disabling DropSame at test time. It might even have to do with the threshold set at a too high, or a too low value.

Comparing accuracies with DropOut –

Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	On 10000 test images
54 %	59 %	40 %	43 %	44 %	28 %	72 %	55 %	69 %	50 %	51 %

Table 18 – With DropOut (0.5), after 2nd MaxPool with ReLU , (only during training)

Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	On 10000 test images
44 %	66 %	44 %	39 %	34 %	20 %	72 %	60 %	73 %	74 %	53 %

Table 19 – With DropOut (0.3), after 2nd MaxPool with ReLU , (only during training)

Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	On 10000 test images
39 %	58 %	11 %	29 %	41 %	40 %	63 %	63 %	65 %	63 %	47 %

Table 20 – With DropOut (0.7), after 2nd MaxPool with ReLU , (only during training)

DropSame does outperform DropOut at 0.7 rate. (These results were obtained using a fixed value of threshold, we found in our implementation, the threshold value would not update, and stays as it was initialized. Updating the threshold value, would lead to different results.)

3.6 Conclusion and Future Work

We demonstrate a novel way to drop units in a neural network, that does not require specifying a manual drop rate.

As we described, that this technique takes more time to train as compared to using DropOut, plus if the problem is about dropping elements in a neural network, and zeroing them out, then one question that comes, is that why not use a binary neural network based system, instead of floating point, integer based systems, and the whole notion of dropping elements in such a system, would mean flipping bits, from 1 to 0, in the next section we explore, one such technique, and describe its application on a toy dataset.

Further experiments on DropSame, would involve, reducing the time to compare elements, and that would enable to implement it in neural networks with a lot more parameters than the neural network described here.

4. Binary Neural Networks

In the above two sections, both the techniques, use floating point-based parameters, in this section we describe a binary neural network-based system, we demonstrate its application on a toy dataset, it is possible to scale it up to even larger datasets.

This section is subdivided into following sub sections,

4.1 Problems with floating point, integer-based parameters

The whole notion of using gradient descent based optimization is based on floating point parameters (that is fp32), or quantizing these parameters to int8, but a simpler way to update these parameters, is if we make them binary, that is limited to 0s and 1s, now on these binary parameters, the update rule becomes applying a NAND gate, or a XOR gate, or a different logic gate, as long as we do not have parameters stuck at a particular value. This even makes it simpler to use location information of each pixel within an image, during training phase, while extracting patterns from an image.

One approach that we thought of is described in the below steps, that involves pattern extraction from the image, along with certain issues encoding location information when using floating point parameters.

Steps -

i. To extract multiple patches from our input image.
ii. For each of these patches, we extract location
iii. Based on this location, we update the representation of each patch, that is if we extracted

[X] -> appears at index [0, 0] -> represented by embedding

[Y] -> appears at index [1, 1] -> represented by embedding

[XY

YX] -> appears at index [5, 5] (consider the location of first pixel for 2x2 patch) -> -> represented by embedding

Then the embedding of first patch would be multiplied (because in floating point we do not have the freedom to use logic gates) with the third patch embedding, as the first patch is part of it, same for second and third patch.

iv. We reduce the representation of the patch that represents the entire image to number of classes in our train set, and get a loss -> backprop -> optimize

v. After this all the patches that we have extracted from the image would have a learnt representation
vi. During the test phase, when given a new image, then we again extract multiple patches, along with location.
vii. Some of these patches would have a learnt representation, as they were found in training images, so our neural network has a learnt representation of some of these patterns.
viii. We once again apply step 3) for test image, and reduce representation of the patch that represents the entire image into number of classes -> get accuracy.

One problem with this approach is, what all patches do we extract from the input image, along with encoding the location information is a memory intensive task, plus due to floating point parameters, updating the embeddings with a XOR or a NAND based logic gate is not applicable.

4.2 Using a binary parameter-based system on one shot learning

Works in 1948 (A. Turing 1948), have described binary neural networks to be the simplest possible model of the nervous system. In our experiment, we need to understand one shot learning, using a binary neural network, for this we carry the following experiment on a 3x3 image.

i. Suppose our image is the following 3x3 tensor,

```
image = torch.tensor([[0., 1., 2.],
                      [0., 2., 1.],
                      [0., 0., 0.]])
```

this image has three variants of pixels, that is 0, 1 and 2, we could assume them to be black, white and grey. So, we use two bits to represent these pixels in binary. Our binary image would now be,

```
binary_image = torch.tensor([[[0, 0], [0, 1], [1, 0]],
                             [[0, 0], [1, 0], [0, 1]],
                             [[0, 0], [0, 0], [0, 0]]])
```

ii. The next step is to extract location information of each pixel, since we use a 3x3 image, so we use 4 bits to represent location information, first two bits indicate row, last two bits indicate column, our location embedding would now be,

```
location = torch.tensor([[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]],
                         [[0, 1, 0, 0], [0, 1, 0, 1], [0, 1, 1, 0]],
                         [[1, 0, 0, 0], [1, 0, 0, 1], [1, 0, 1, 0]]])
```

iii. Now, we concatenate the above two tensors, to obtain a embedded_image_with_location tensor, which contains binary representation of each pixel, along with its location, which would now be,

```
embedded_image_with_location = torch.tensor([[[0, 0, 0, 0, 0, 0],
                                              [0, 1, 0, 0, 0, 1],
                                              [1, 0, 0, 0, 1, 0]],

                                             [[0, 0, 0, 1, 0, 0],
                                              [1, 0, 0, 1, 0, 1],
                                              [0, 1, 0, 1, 1, 0]],

                                             [[0, 0, 1, 0, 0, 0],
                                              [0, 0, 1, 0, 0, 1],
                                              [0, 0, 1, 0, 1, 0]]], dtype=torch.int32)
```

the first two bits represent the pixel variation, and the last four bits represent location.

iv. The next step is to create a pattern_extractor from this embedded_image_with_location tensor, one question that comes, is what all patterns do we extract, for this we extract patterns at random, because our neural network needs to learn representations of patterns, initially, it would be extracting patterns at random, suppose our pattern_extractor is,

```
pattern_extractor = torch.randn(5, 3, 3).int().bool().int() # we extract 5 patterns
```

and

```
pattern_extractor[0]
```

is

```

tensor([[1, 1, 1],

[0, 1, 0],

[1, 1, 0]], dtype=torch.int32)

```

this means that the first pattern contains these 6 pixels, similarly for the next 4 patterns. We expect our neural network to learn to extract patterns. So, we do not specify a particular way to extract patterns, although a better technique to extract patterns from the input image is a research question.

v. Now, we extract patterns from our input image based on our pattern extractor, so we get a tensor like,

```

patterns_extracted # these are the five patterns extracted

```

which is,

```
[tensor([[0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0],
        [1, 0, 0, 1, 0, 1],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 1]], dtype=torch.int32),
 tensor([[0, 0, 0, 1, 0, 0]], dtype=torch.int32),
 tensor([[0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 1],
        [0, 0, 1, 0, 0, 0]], dtype=torch.int32),
 tensor([[0, 1, 0, 0, 0, 1]], dtype=torch.int32),
 tensor([[0, 0, 1, 0, 0, 0]], dtype=torch.int32)]
```

The last four bits indicate location, the first two bits indicate the pixel variation.

We can see here, that the 4th pattern is a part of the 3rd pattern, the 5th pattern is also a part of the 3rd pattern, that is there is overlap.

The next step is to determine whether there is an overlap between these, and accordingly create a link dictionary, for the above example, the link dictionary would be, (assuming indexing begins from 0, and 5 represents our image)

```
defaultdict(list, {0: [5], 1: [5], 2: [0, 5], 3: [0, 2, 5], 4: [0, 2, 5]})
```

this implies that the first pattern (index 0) is a part of the entire image (index 5), the 4th pattern (index 3), is a part of the first (index 0), the third (index 2) pattern and the entire image (index 5).

vi. Now, we ask our neural network what class does this image belong to, in the beginning it will make a random prediction, it is like asking a person who has never read a word in Arabic language, the meaning of an Arabic word, but we give options, so the person would choose a random option. Suppose the options are,

```
options = torch.randn(4, 20).int().bool().int()
```

which means that we give four options, each of them are represented by 20 binary digits (20 is a hyperparameter).

vii. The initial representation of the 6 (including the entire image) patterns is random (not yet learnt),

```
representation_of_patterns = torch.randn(6, 20).int().bool().int()
```

again by 20 digits.

viii. When told with the correct answer, our neural network will update the linking system, suppose options[1] is the correct answer, now the linking system would look like, (options[0] indexed by 6, options[1] indexed by 7)

```
defaultdict(list, {0: [5], 1: [5], 2: [0, 5], 3: [0, 2, 5], 4: [0, 2, 5], 5: [7], 7: [5]})
```

We create a two-way link between our image pattern (index 5), and its text class (index 7).

ix. The next step is to update the representation of these patterns based on the representation of the correct option, to do this, we do a simple XOR operation on the representation of each pattern, the updated representations would now be,

```

7 -> stays as it is

5 -> 5 XOR 7

4 -> 4 XOR 5 XOR 7

3 -> 3 XOR 5 XOR 7

and so on

```

By the end of this step, our neural network has an updated representation of each pattern, plus an updated linking system.

x. When come up with a variation of the same class, during the test phase, if the image contains a pattern that our neural network learnt during training, then the linking dictionary would get triggered.

xi. The prediction would be based on how many steps does it take to reach a particular pattern, so if the task is to classify an image, then our system searches for patterns within the image, and number of steps it takes to reach a label is the way it makes the prediction. We want the correct prediction to be at minimum distance / number of links from the patterns within an image.

4.3 Text Example

This concept can be more easily seen from a text example,

Let us see what would happen if all of these patterns were words

For the sentence,

I ate an ____

0 1 2

the task is to predict the missing word

options

a) apple - 3

b) furniture - 4

c) key - 5

d) bed - 6

the correct answer is a) apple

we want the link dictionary to be like,

I -> ate

0 -> 1

Means on seeing the word ‘I’, the word ‘ate’ gets triggered.

At each step multiple words would get triggered, for example, when reading the above sentence, after reading the word 'I', one might say, 'ate', 'went', 'go', 'am' and so on, that is all of these words are linked to the word 'I'. We can see this from our own experience, as we start thinking of multiple words, when come across a single word.

I ate -> an

0 1 -> 2

After reading the word 'ate', now we have a combined pattern 'I ate', which would in turn once again trigger certain patterns, one might say 'today', 'an', 'after' and so on.

I ate an -> apple

0 1 2 -> 3

After reading the word 'an', now we have a combined pattern 'I ate an', which would in turn once again trigger certain patterns, one might say 'apricots', 'avocados', 'apple' and so on.

In the beginning, this linking system would be empty, (just like an infant cannot speak a language just after birth, an infant's brain learns patterns over time), so links would be created, and patterns would be combined. If our model gives a wrong prediction, then that link gets broken. If our model gives a correct prediction, then that link stays.

The way we combine patterns is using a XOR operator, we use XOR, because other operators would lead to representation getting stuck at 1 or 0, one could even use a NAND operator, or an XNOR operator.

So, if we have,

'I' as index 0, represented by 20 bits 00010101001010100101

'ate' as index 1, represented by 20 bits 10101011001010100000

(20 bits is a hyperparameter, these are random representations)

then

'I ate' would be index 2, represented by XOR between 'I' and 'ate'

'I ate' would be 10111110000000000101

'an' as index 3, represented by 20 bits 01011110010100101001

'I ate an' would be index 4, represented by XOR between 'I ate' and 'an'

'I ate an' would be 11100000010100101100

The link dictionary would be,

0 -> 1

2 -> 3

4 -> 5

'I ate an apple', that is the entire sentence, would also be assigned an index, let’s say 15

Or another way to represent the link dictionary is,

((((((0 -> 1) = 2) -> 3) = 4) -> 5) = 15)

When come up with another sentence

I went to a ___

0 6 7 8

the word 'I' would update its link dictionary

0 -> 1, 6

'I went' as index 10, would be XOR between 'I' and 'went', linked to the word 'to', at index 7

10 -> 7

'I went to' as index 11, would be XOR between 'I went' and 'to', linked to the word 'a', at index 8

11 -> 8

'I went to a' as index 12, would be XOR between 'I went to' and 'a'

12 -> our prediction (let’s say the word 'park', indexed by 13)

14 (our entire sentence 'I went to a park')

Finally, our link dictionary would look like,

0 -> 1, 6

2 -> 3

4 -> 5

10 -> 7

11 -> 8

12 -> 13

14

Or another way to represent the link dictionary is,

((((((0 -> 1) = 2) -> 3) = 4) -> 5) = 15)

|

(((((((0 -> 6) = 10) -> 7) = 11) -> 8) = 12) -> 13) = 14)

On receiving a negative feedback, a link gets broken, while on receiving a positive feedback, the link gets created.

Our neural network creates this massive linking system, and when come up with new sentences, if a pattern (word in the case of text) is already linked to another, then it would get triggered, otherwise memory would be allocated for a new pattern / a new link.

4.4 Conclusion

This work demonstrates a novel binary neural network-based system for one shot learning, further experiments would involve scaling it up to a bigger dataset. Plus, modification to the technique to extract patterns from the original image, as here we extract patterns at random.

5. Functioning of Human Neocortex

In this section, we describe, certain statistics related to human brain, as in order to enable computers to learn as humans do, we need to have an understanding of how human brain works, and due to advances in both noninvasive and invasive brain scanning techniques, we keep getting more and more insights into understanding our own actions and behavior. Although in 2020, we do not entirely know that this is how our brains work, but we do have some information based on better digital reconstructions of human brain.

5.1 Linking based learning system

The way we learn concepts is based on a massively parallel pattern recognition system, which is unorganized, that is patterns are linked to each other. One insight is that we do not store patterns in 2d, or in 3d in memory, but instead in 1d, the evidence for this is that, we are in general not so good at drawing objects, even which we see on a regular basis, and even sophisticated artists need to look at the object that they are drawing. So, there is a strong probability that we store patterns in 1d, since the pattern recognizers in our brain consider images, text, the same way, that is we do not have different pattern recognizers for text, images, so these are all stored in 1d.

Another evidence for this pattern recognition system is that people learn their first language pretty quickly, while it takes way longer for someone to learn a language by the time they have reached 20 years of age, one way to see this is that most of the pattern recognizers are occupied by the time one is 20 years old, and when we are learning our first language, these pattern recognizers are free.

We appear to be creating links between these patterns, for example, when one comes across the word 'apple', then they might start thinking of other words, like, 'fruit', 'taste', 'red', 'tree', that is a whole linking system gets triggered.

And when learning a new pattern, we again create a link towards it, for example, if we are asked the meaning of a word in a language that we have not learnt, then we will not be able to come up with any answer, because there is no link from that pattern to any other pattern.

For example, let the word in new language be 'X', initially, if asked what is 'X'?, then we will have no answer, but if told that 'X' means 'car', then we will create a link between the pattern of the word 'X' and the pattern of the word 'car', so the next time we read the word 'X', it will trigger the whole link associated with 'X' -> 'car' and all

the patterns linked with the word 'car', for example, 'driver', 'self-driving', 'autonomous' and so on will get triggered.

Now, the pattern linking system, has an updated pattern, suppose, that we say this word 'X' does not mean 'car', but instead 'furniture', so the link associated between 'X' -> 'car' gets broken, and a new link between 'X' -> 'furniture' gets established, the next time we read the word 'X', it will trigger the whole link associated with 'X' -> 'furniture' and so on. This breaking of link does not happen instantaneously, therefore, one might still trigger the link associated with 'X' -> 'car'.

This learning system can also be seen in gaming environments, for example, people face difficulty when told to change key binds in a game, and often end up pressing the old key bind that they were using earlier.

That is, it takes time, to break a link, and to create a new link between these patterns.

This is similar to a punishment, reward-based system, where on getting a reward, the link gets stronger, while on getting a punishment, the link gets weaker.

Another insight is that learning a new language when shown words only in that language is not possible, for example, suppose we have four words in a language that we have not learnt 'A', 'B', 'C' and 'D', when asked with the question, what is the meaning of a combination of words 'A' 'B' 'C', and told with the answer 'D', still, we will end up in a state, ok, but what is 'D'?, I do not know that also, as all these four words are in a language we have not learnt, in order to learn the meaning of these words, we will have to link them to patterns in a language that we have learnt before, so one could say that the meaning of 'A' 'B' 'C' is 'ant', then we will link these patterns with each other, plus we also have the word 'ant' linked with the image of the pattern 'ant', that is how we learn our first language, linking text with images.

We are not able to learn a language without linking patterns of that language with patterns of images, patterns of a language that we have already learnt. And for our first language, we link words with images, so, to learn the meaning of the word 'apple', we will have to see the picture of an apple, otherwise if we only see words, then we will end up in a recursive state of, ok, but what does that mean? So, we create a linking system, between words and images, probably all of them stored in 1d, it is possible to store a 2d image, or a 3d image in 1d, as demonstrated in the previous section.

Any event that leads to destruction, is avoided, and any event that has a constructive outcome is carried on, for example, one would carry an action like eating a candy in

the real environment, as eating a candy leads to delicious taste that is a constructive outcome, one might as well start eating multiple candies, while if the same task of eating a candy was in a game environment, and the game developer reprogrammed the rules, that there would be no delicious taste, and instead your game points would reduce for doing the task, then one would start avoiding doing that task, as it has a destructive outcome.

Furthermore, concepts such as humor, or understanding a joke, only appear to happen after learning, for example, we do not laugh on a joke in a language we do not understand, only after we have created a linking system between patterns, are we able to comprehend and create humor in a particular language.

5.2 Certain statistics related to human brain

Based on advances in human genome, and human connectome projects, certain statistics are getting revealed, about both our bodies and brains, it turns out that the information content in human genome is around 750 MB in size, this includes both body and brain. The human brain is able to do around 10**14 calculations per second (Kurzweil 2012), computers in 2020, are not there yet, but are evolving fast. Supercomputers have already crossed this number, the state-of-the-art supercomputer, in 2020, Fugaku, already is capable of around 10**17 FLOPS.

This does lead to more and more of the things that our brains and bodies do, getting represented statistically. In the final section, we discuss certain ethical concerns associated with this.

6. Conclusion

The primary novel work in this thesis is described below.

We point out results on carrying experiments using non convolutional neural networks, that work on attention-based mechanisms. which show improvement in few shot learning.

We describe a novel way to drop units in a neural network, that works on a similarity basis and overcomes the need of a manual rate as input.

We list out the limitations of floating point based neural networks, in terms of updating the parameters, encoding location information in the case of images, and describe the use of Binary Neural Networks, to create a biologically inspired learning system.

We provide certain directions for creating a linking based system, to solve the problem of one-shot learning.

Further research on one-shot learning needs a convergence between natural language processing and computer vision, to create learning systems, like humans learn, looking at text, and then associated images.

Over the upcoming years, as we get more insights into human brain functionality with more detailed digital reconstructions of biological neocortex, it would lead to positive steps towards enabling computers to learn with a limited number of examples.

7. Epilogue

In this section we describe certain ethical, negative concerns associated with computers being able to do tasks that humans do, and instead of giving a negative outlook, we describe the positive factors, of advances in artificial intelligence. One needs to keep in mind, that this is a pattern in evolution, that computers keep getting stronger every year, in terms of number of calculations per second they are able to do, along with memory storage, this leads to them being able to do certain tasks that only humans did. In 2020, we have computer generated voice, computer generated avatars, computer generated music, all of which is evolving and getting better every year. After a certain stage, it becomes difficult to draw a line between this is not computer generated, and this is computer generated. For example, take the voice of Siri, the number of bits of information represented by this voice keeps increasing, the voice of real world avatars is also subject to evolution, but the computer generated voice is evolving at a faster pace, so we hear, such voices, while being in a game environment, or while reading a book with Siri, and so on. This leads to even voice trolling, voice cloning using computer generated voice on different online platforms. The same pattern applies for computer generated avatars, so in terms of pixel density, these avatars keep evolving, that is we see higher pixels per inch avatars in virtual environments, as computers keep getting stronger, this leads to creating avatars that look like humans, but do not have an existence in the real world. Another pattern in evolution is that number of interactions between people, that involve computers in between keep increasing, as the number of bits transferred wirelessly keep increasing with evolution.

Combining both of these things, leads to an identity issue, as one can appear with a voice of their choice, with an avatar of their choice, in a virtual environment, at this stage, is one's identity now a link?, and we cannot say that a person's appearance or voice is a strong measure of their identity. This can be further combined with the choice to be in anonymous mode, so one cannot tell who they are with even after seeing their name, as it is set to anonymous.

These issues will only keep increasing, as computers keep getting stronger, and they have already started in 2020, we can see these issues in virtual environments.

Another concern is related to what will humans do, if computers are being able to do all of the tasks that humans do, for example, why would one want to learn a language, when one can use a translation service, why would one want to attempt at looking better in the real world, when they can use a superimposed avatar, and go

entirely virtual, carrying all interactions with computers in between, overlaying an animated avatar over their real world avatar. One thing to keep in mind is over the upcoming years, certain feedbacks that are only possible in the real world, for example, an olfactory feedback, or tactile feedback, or taste, are all coming to virtual environments, so we can expect people spending a lot of their time in these environments, and it keeps getting difficult to draw a line between what is virtual and what is real.

Another way to look at this situation is we start considering the notion of identity, based on a person's actions and behaviors, for example, how exactly do they play a game matters, or what kind of memes are they making matters, we start finding a continuity in their identity based on the style in which they write text, that is the notion of their identity goes beyond how they look like, or what they sound like.

One more issue that arises with the notion of being able to look like whatever one wants to look like, or sound whatever one wants to sound like in a virtual environment, is imposter, as an avatar that one person takes, someone else also has the option to take the same avatar, this leads to impersonation and associated fraud acts.

The i/o bandwidth between humans and computers is also subject to increase in the upcoming years, so a lot of the interactions will be in a virtual/gaming environment, plus higher i/o bandwidth leads to stronger manipulation of the surrounding environment, leading to people spending more time in a gaming environment.

Advances in artificial intelligence will lead to creating simulated avatars in a virtual environment, who would be able to learn just like humans do, what is known as a 'bot' in games today, is evolving pretty fast, and one can only imagine, computer generated avatars, being able to learn to play just like humans do, over the upcoming years, this is the case in 2020 also, for some games, and these advances will lead to having more intelligent computer generated avatars in even more games, virtual environments.

A future scenario is one not being able to distinguish whether an avatar has a presence in the real world, or exists only in a virtual environment.

Eventually one can imagine a scenario where entire human brain is modelled both functionally and atom by atom in a virtual environment, one thing to keep in mind is that the same has already been done for a rats brain, and neuromorphic chips emulating the functionality of human brain are already available, after a particular

stage, we will see entirely simulated avatars, who learn as humans learn, or even replicas of one's real world avatar in a virtual world, as of now, we can only hope that by that time probably we will have some ways to solve this identity issue.

8. Acknowledgment

Thanks for reading this work. We had a fun time writing this work, we hope you had a fun time reading it.

This research is supported by JICA. All experiments were carried using NVIDIA GTX 1080 TI. We even found Google Colab to be useful for this work. We would also like to thank people involved in useful discussions carried out on PyTorch forums (https://discuss.pytorch.org/u/vainaijr/summary), Github issues (https://github.com/vainaijr), for bringing this work altogether.

9. Appendix

9.1 Bibliography

Subutai Ahmad, and Luiz Scheinkman. 2019. "How Can We Be So Dense? The Benefits of Using Highly Sparse Representations." *CoRR* abs/1903.11257. doi:Tue, 02 Apr 2019 11:16:55 +0200.

Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V. Le. 2019. "Attention Augmented Convolutional Networks." *CoRR* abs/1904.09925. doi:Fri, 26 Apr 2019 13:18:53 +0200.

Wei-Yu Chen, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. 2019. "A Closer Look at Few-shot Classification." *CoRR* abs/1904.04232. doi:Thu, 25 Apr 2019 13:55:01 +0200.

Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. 2018. "DropBlock: A regularization method for convolutional networks." *CoRR* abs/1810.12890. doi:Thu, 08 Nov 2018 10:57:46 +0100.

Roshan Gopalakrishnan, Yansong Chua, and Ashish Jith Sreejith Kumar. 2019. "Hardware-friendly Neural Network Architecture for Neuromorphic Computing." *CoRR* abs/1906.08853. doi:Mon, 24 Jun 2019 17:28:45 +0200.

Ray Kurzweil. 2012. *HOW TO CREATE A MIND.* London: Viking Penguin, a member of Penguin Group (USA) Inc.

Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jonathon Shlens. 2019. "Stand-Alone Self-Attention in Vision Models." *CoRR* abs/1906.05909. doi:Mon, 24 Jun 2019 17:28:45 +0200.

Sachin Ravi, and Hugo Larochelle. 2017. "Optimization as a Model for Few-Shot Learning." *ICLR.*

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15: pages = {1929-1958},.

Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. 2019. "Meta-Dataset: A Dataset of Datasets for Learning to Learn from Few Examples." *CoRR* abs/1903.03096. doi:Sun, 31 Mar 2019 19:01:24 +0200.

A. M. Turing. 1950. "Computing Machinery and Intelligence."

A. M. Turing. 1948. "Intelligent Machinery." https://weightagnostic.github.io/papers/turing1948.pdf.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." *CoRR* abs/1706.03762. doi:Mon, 13 Aug 2018 16:48:37 +0200.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. "Regularization of Neural Networks using DropConnect." Edited by Sanjoy Dasgupta and David McAllester. *Proceedings of the 30th International Conference on Machine Learning* (PMLR) 28: 1058--1066. doi:17--19 Jun.

P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona. 2010. "Caltech-UCSD Birds 200." (California Institute of Technology).

9.2 List of figures

9.3 List of tables