Using Rank-N Approximation and Matrix Manipulation To Speed up Convolutions in Convolutional Neural Networks (CNNs)

Sri Jaladi

I. Introduction

Recently, convolutional neural networks (CNNs) have taken over image classification and image recognition deep learning and AI models. The most key component component to CNNs is the component known as a "convolution". This is, generally explained, as the process where one filter matrix is slide around another larger matrix and only the portion underneath the filter matrix is explored and the result corresponds to a single value in the final convoluted matrix. This is explained in more detail in the paper.

However, one of the biggest issues that these models, along with any deep learning models, run into is time. These models can get fairly complex, large, and can be doing computations over a very large number of images and/or datasets. As such, the speed of the pure model can be extremely important.

The naive implementation to the convolution function utilizes for loops, which are extremely expensive in python. However, the convolution function can be sped-up through the usage of matrix manipulation on what the actual convolution function is doing along with utilizing the best rank-n approximation matrix given to us via Eckert-Young. In fact, the lower the rank during rank-n approximation, the faster the convolution function will process, but the lower the accuracy of our model will be. The reasoning for this is explained throughout the paper. Further, experiments are conducted to verify these ideas, analyze the trade-off between rank/time and accuracy, and even beat PyTorch (an AI/ML Python library commonly used throughout many real-world industries) on speed.

II. Notation

For notational purposes and consistency, in a 2d matrix C the the ith row is designated by $row_i(C)$, the jth column is designated by $col_j(C)$, and the element in the ith row and jth column is designated by $C_{i,j}$. The submatrix of C including rows $i \rightarrow k$ and columns $j \rightarrow l$ inclusive is represented as $C_{i:k,j:l}$. Throughout the paper convolutions are going to be convoluting the filter $F \in \mathbf{R}^{m_f x n_f}$ over the matrix $A \in \mathbf{R}^{m_a x n_a}$. In general, the matrix P is such that $P = A_{1:m_f,1:n_f}$

III. Convolutions Background

A convolution applies a Filter matrix F (usually a square matrix of odd dimension) on another matrix A (usually a square matrix). Assume that $F \in \mathbf{R}^{m_f x n_f}$ and $A \in \mathbf{R}^{m_a x n_a}$. Note that for a convolution to be possible/valid it must be true that $m_f \leq m_a$ and $n_f \leq n_a$. The convolution then creates a resulting matrix B such that $B \in \mathbf{R}^{(m_a - m_f + 1) x (n_a - n_f + 1)}$. Further, each element $B_{i,j} : \forall i \in \{1, 2, 3, (m_a - m_f + 1)\}, \forall j \in \{1, 2, 3, ...(n_a - n_f + 1)\}$ is determined as $\sum_{w=i}^{i+m_f-1} \sum_{z=j}^{j+n_f-1} F_{(w-i+1),(z-j+1)} \cdot A_{w,z}$.

The more intuitive approach to the process of a convolution is to imagine it in pictoral form (which is the way it is taught and approached in general). This is the commonly approached manner with computing convolutions as well. Imagine overlapping F in the top-left corner of A such that $F_{1,1}$ overlaps with $A_{1,1}$ and F_{m_f,n_f} overlaps with A_{m_f,n_f}. Now, we simply take the product of each pair of overlapping elements and take this sum. This determines the top-left most element in the resulting B. We then slide F over by 1 column and repeat this process to determine another element of B. We continue to do this over and over until sliding F would cause a column in F to go beyond A. Then we reset F to its original position and slide it down by 1 row and repeat the sliding process for this row. We do this over every possible row and column valid overlap possible to fully determine B. A pictoral representation can found in figure 1.

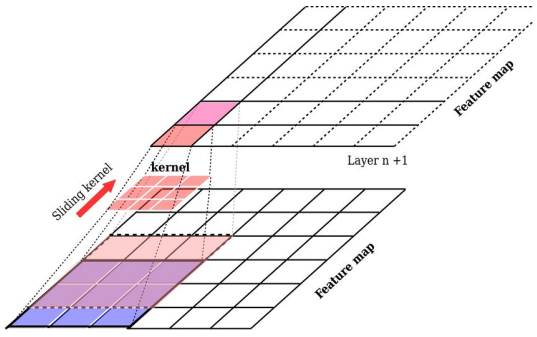

Fig. 1. Sliding Filter Convolution Pictorial Representation ($Citation1$)

Convolutions are most commonly used in Convolutional Neural Networks (CNNs), which is a deep learning algorithm/process which consists of chaining together multiple convolutions. Usually, CNNs are used in image classifications and convolutions are often utilized to gather certain features or components of an image. For example, the filter $F = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$ is often utilized to detect vertical edges

in a given picture due to its ability to try and emphasize on contrasting columns in the image.

IV. The Time Issue

```
def conv_reg(F: np.ndarray, A: np.ndarray):
    M_a, N_a = np.shape(A)
    M_f, N_f = np.shape(F)
    M_res = M_a - M_f + 1
    N_res = N_a - N_f + 1

    res = np.zeros((M_res, N_res))

    for r in range(M_res):
        for c in range(N_res):
            res[r][c] =\
            np.sum(A[r:r+M_f,c:c+N_f] * F)

    return res
```

```
A = np.random.randn(1002,1002)
F = np.random.randn(3,3)
start = time.time()
conv_reg(F,A)
end = time.time()
print("Time_taken: "+str(end-start))
```

```
Time_taken: 7.74310302734375
```

```
A = np.random.randn(10000,1000)
B = np.random.randn(1000,10000)
start = time.time()
C = np.matmul(A,B)
end = time.time()
print("Time_taken: "+str(end-start))
```

```
Time_taken: 4.093238830566406
```

Fig. 2. Pythonic For Loop vs Vectorized Matrix Multiplication

Above is the naive but most regularly used implementation for creating a convolution function which applies a convolution filter F onto the matrix A as described previously. The major issue with this is that we are using pythonic for loops, which are, in comparison to many other operations utilizing numpy, extremely slow.

Part of the reason that numpy along with its many functions, capabilities, and tools are so efficient and fast is that they are not only written in Fortran/C code (which is significantly faster than code written in python) but they also are able to take advantage of components of the computer itself to compute values, computations, and formulas in parallel.

While this may seem odd or unintuitive, the basic idea is that doing as many operations as possible through purely numpy is significantly better than adding ANY pythonic loops. This is especially true when we are working with large-scale data and hefty AI/ML models which require significant amounts of copmutation and data management. Pythonic for loops are very dangerous.

A simple experiment can display the true difference in speeds and potential danger of pythonic for loops. See Figure 2 for these results.

The first cell does approximately a total $10 \cdot 1000 \cdot 1000 =$ 10million computations. The second cell actually does a total of approximately $10000 \cdot 1000 \cdot 10000 = 10000 \cdot 10000000$ = 100billion computations. And it does it in approximately half the time. This displays the inefficiencies that are generated by pythonic loops (and actually generalized to loops in general) in comparison to the heavy heavy efficiency benefits of vectorization.

The justifications and reasonings for this deficiencies are beyond the scope of this class and paper and are more concerned with computer systems, architecture, and the design and code of these libraries. This paper is more focused specifically on convolutions and why and how they can be sped up. In order to do this, the main idea is that we need to transition the convolution from the sliding window element-by-element product and summation to something more optimal (a matrix multiplication for example).

V. Recent Innovations and Goal

Now, due to recent development, libraries such as PyTorch have been written to be able to utilize for loops in other faster languages (such as C++ or C) to compute convolutions. Python itself is not one of the fastest languages out there and languages such as C/C++/Java and others can have clear advantages. Due to this, despite potential inefficiencies in their convolution functions, due to being written in an extremely fast language, when converting to python such inefficiencies can become negligible.

PyTorch, that AI library that was used throughout this project, for example, has a backend significantly comprised of C++ code. In fact, PyTorch uses a very large proportion of the computational tricks that numpy uses for speedup as well, so models being sent through PyTorch are able to go as fast as possible. PyTorch itself comes with an in-built convolution function which is used extensively in CNN models throughout industry and production released AI models. Such convolution function does seem to be computed in the sliding window manner, which requires for loops. It's important to note that numpy does NOT come with a convolution function (at least for 2 or more dimensions). Another item to note is that PyTorch is also a significantly more recent development (being created in 2016) in comparison to numpy whose origins date back to 2005.

Our essential final or hopeful goal throughout this project is to speed up the convolution process utilizing only numpy in order to compete against the speed of an industry-wide utilized PyTorch. Obviously, a goal like this is rather lofty, and so it comes with some understanding that there is likely to be some form of tradeoff. In this case, we expect this tradeoff to come in the form of accuracy. The reasoning for this will be discussed in a future section. However, our more reasonable goal is to display a convolution speed-up using

the normal implementation and sped-up implementation with BOTH in numpy and NOT involving PyTorch. This is because comparing two numpy-based implementations allows a valid comparison using the same underlying systems. Comparing to PyTorch is the idealistic goal but is comparing to a somewhat different system entirely. But these two general goals are what we take forward.

VI. Convolution if F is Rank-1 (Submatrix of A)

Recall the mathematical/pictoral formulation for a convolution defined in the "Convolutions Background" Section. Now assume that F (the filter) is a rank-1 matrix. Then, we can write F in the form of a single outer product as $F = fg^T$ where $f \in \mathbf{R}^{m_f}$ and $g \in \mathbf{R}^{n_f}$. This means that we can write each element of F as $F_{i,j} = f_i g_j$. Revisiting the convolution formula for using the filter F over the matrix A, we can write the result of this convolution as B. Firstly, note that each element $B_{i,j}$ in B can be extracted solely using the values of F and $A_{i:(i+m_f-1),j:(j+n_f-1)}$. Thus, we can just focus on the computation of $B_{1,1}$ as this process can simply be generalized to every $B_{i,j}$ anyway since they all work with identical shapes/sizes. Let's denote the matrix P such that P is the upper-left most matrix in A with shape $m_f, n_f \Rightarrow P = A_{1:m_f,1:n_f}$. Then element $B_{1,1}$ can be written as

$B_{1,1} = \sum_{w=1}^{m_f} \sum_{z=1}^{n_f} F_{w,z} \cdot P_{w,z} =$
$\sum_{w=1}^{m_f} \sum_{z=1}^{n_f} f_w \cdot g_z \cdot P_{w,z} =$
$\sum_{w=1}^{m_f} f_w \cdot (\sum_{z=1}^{n_f} g_z \cdot P_{w,z}) =$
$\sum_{w=1}^{m_f} f_w \cdot (row_z(P)g) =$
$f^T P g.$

Thus, if F is a rank-1 matrix, we can transform the extraction of any element $B_{i,j}$ into a matrix multiplication with the sub-matrix $A_{i:i+m_f-1,j:j+m_f-1}$ and the vectors which create the outer product of F (f, g). This allows us to get away from the concept of needing to loop through element by element, taking a product, and then finally taking the sum. While this may seem minor, stepping away from abstract products and summations and transforming them into matrix multiplications can significantly help with computations, properties, efficiency, and time.

In terms of numpy, without this formulation of a convolution, the previous convolution formulation (for element $B_{1,1}$) would be implementation as $np.sum(F * P)$ where $*$ is the element-by-element product operation (returning a matrix equal in size to F and P) and $np.sum$ takes the sum over all elements in the matrix.

Assuming F is rank-1, with our formulation, we can write the result of the convolution for element $B_{1,1}$ as $np.matmul(f, np.matmul(P, g))$ where $np.matmul$ represents matrix multiplication in order of the given variables.

VII. Convolution if F is Rank-1 (Full Matrix A)

In the previous section converting a convolution over a single submatrix of A (with the same size as F) was covered under the assumption that F was rank 1. Under the same definitions and notation from the previous section the formulation for $B_{1,1}$ was $f^T P g$.

However, this only extracts a single value for B. Now, we turn our attention over to trying to generalize this multiplication so that it applies to the entire matrix A and can get us the resulting matrix B without any need for repetitions in the form of a for loop situation. Essentially, we want to get B in terms of some matrix multiplications of A.

Let's start simple. Let's just assume we only want $B_{1,1}$ but instead of using P, we only want to use matrix A. To do this, let's utilize our formulation $B_{1,1} = f^T P g$. Notice that we just need f as a row vector and g as a regular column vector attached to the top-left-most portion of A. Every other value in A is not needed for this calculation. Thus, we can essentially create a new row vector in the following form $f'^T = \begin{bmatrix} - & f^T & - & 0 & 0 & ... & 0 \end{bmatrix}$ and a column vector in the form $g' = \begin{bmatrix} - & g^T & - & 0 & 0 & ... & 0 \end{bmatrix}^T$. Notice that we can now utilize these to get $B_{1,1} = f'^T A g'$. What this is essentially doing is recreating the convolution but in the form of a matrix multiplication instead and is 0ing out unnecessary terms that aren't being used to compute this certain value.

This process gets even more interesting because it turns out we can actually start to generalize this situation for all $B_{i,j}$. All we have to do is slide f^T down and over by 1 each time to get the following matrix.

$f^{*T} = \begin{bmatrix} - & f^T & - & 0 & 0 & ... & 0 \\ 0 & - & f^T & - & 0 & ... & 0 \\ 0 & 0 & - & f^T & - & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & 0 & - & f^T & - \end{bmatrix}$. We then go the same for the g portion to get the matrix

$g^* = \begin{bmatrix} - & g^T & - & 0 & 0 & ... & 0 \\ 0 & - & g^T & - & 0 & ... & 0 \\ 0 & 0 & - & g^T & - & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & 0 & - & g^T & - \end{bmatrix}^T$. For the sake of consistency let's call these matrices the "extended-slide matrix of a vector" which we denote as v^* for some vector v.

Combining these two with A, we get our final result to give $B = f^{*T} A g^*$. While this may seem unintuitive on first glance and is a bit tedious to prove straight away, working upward from the steps displayed through the paper shows that this computation is actually correct. For a sanity check, one can easily verify that the dimensions do match up and that each value computed in B is essentially 0ing out everything other than the specific submatrix needed from A (if requiring additional convincing one can rigorously test and prove this in a tedious manner and/or view the results of the data and model later which provides evidence to this formulation).

This is a very important step that has been reached because this formulation has transformed the convolution from its original formulation into a product of 3 matrices. As discussed previously, matrix multiplications and their computations are significantly faster than regular convolution approaches. This formulation will be essential for the rest of this paper.

As a note, it is important to disclose that creating the matrices of f^* and g^* does require a pythonic for loop to shift f and g over. However, this is of the total runtime (in terms of the python for loop) order of $O(R+C)$ instead of $O(RC)$ like usual convolution (where R is number of rows and C is number of columns) so this is an extremely large improvement (something that is supported by the data/results later as well).

VIII. Convolution if F is Rank-n

Now that it has been covered how a convolution F can be applied to a full matrix A if F is a rank-1 matrix, it can now be explored how to replicate some form of this process similarly if F is NOT a rank-1 matrix (which convolutions after significant amounts of backpropagation are usually not). Once again, let's start simple and assume without loss of generality that F is a rank-2 matrix. Then, this means we can write F as the sum of two outer products of vectors so let's write $F = fg^T + xy^T$. It's not immedaite how the previous results can be applied to this situation. Due to this, we must work back through the some previous steps but conducted on this rank-2 matrix. Revisiting previous work, let's focus first solely on extracting the value $B_{1,1}$ through matrix multiplication.

Keep the notation of $P = A_{1:m_f,1:n_f}$. Then, applying the convolution F to P gives us $B_{1,1}$. As noted before, the method of generated $B_{1,1}$ as a summation is equal to
$B_{1,1} = \sum_{w=1}^{m_f}\sum_{z=1}^{n_f} F_{w,z}\cdot P_{w,z} =$
$\sum_{w=1}^{m_f}\sum_{z=1}^{n_f}(f_w\cdot g_z + x_w\cdot y_z)\cdot P_{w,z} =$
$\sum_{w=1}^{m_f}\sum_{z=1}^{n_f} f_w\cdot g_z\cdot P_{w,z} + x_w\cdot y_z\cdot P_{w,z} =$
$\sum_{w=1}^{m_f}\sum_{z=1}^{n_f} f_w\cdot g_z\cdot P_{w,z} + \sum_{w=1}^{m_f}\sum_{z=1}^{n_f} x_w\cdot y_z\cdot P_{w,z} =$
$\sum_{w=1}^{m_f} f_w\cdot(\sum_{z=1}^{n_f} g_z\cdot P_{w,z}) + \sum_{w=1}^{m_f} x_w\cdot(\sum_{z=1}^{n_f} y_z\cdot P_{w,z}) =$
$\sum_{w=1}^{m_f} f_w\cdot(row_z(P)g) + \sum_{w=1}^{m_f} x_w\cdot(row_z(P)y) =$
$f^TPg + x^TPy$. As it turns out, things work out nicely and we are now able to express $B_{1,1}$ as the combination of a matrix multiplication pair and a summation.

Generalizing this process using the results from before, we create our "extended-slide" matrices for f, g, x, y and we can write B in the form $f^{*T}Ag^* + x^{*T}Ay^*$. And with that, we are once again able to represent B using matrix multiplication (and an addition, but this isn't that computationally strenuous)!

Once again, through a bit of extension, we can generalize this for any convolutional filter F of rank-n. Any filter F can be written as the sum of outer-products (which is equal to the rank of F). Assume this summation is $F = f_1g_1^T + f_2g_2^T + f_3g_3^T + ... = \sum_{i=1}^{n} f_ig_i^T$ (where for this once instance f_i is some vector because we are running out of lowercase letters to use). Under this, we can take the "extension-slide" of each of these vectors and get the post-convolution matrix B in the form of $B = \sum_{i=1}^{n} f_i^{*T}Ag_i^*$. As such, we can take any filter F and apply convolution on A using just matrix multiplications and matrix additions and avoid the regular convolution method.

Computationally, higher-rank filters ARE going to take longer than when they were just rank-1, but in terms of numpy (and compared to the regular convolution) this is still significantly better than the regular option. However, this is additional time, and in the pursuit of being the fastest, it would be ideal to turn every filter into a rank-1 filter. Unfortunately, this is not possible, but it is possible to come close, through rank-1 approximation, and more generally rank-n approximation.

IX. Using Rank Approximations on F to reduce computation

As stated previously, it isn't possible for all (or even that many post-training) filters to be rank 1 or anything besides full rank (due to the precision of their decimals). However, if these filters are close enought to a lower-rank filter, we may be able to save some computation time while NOT sacrificing much accuracy. At the most extreme end, this would mean findint the closest rank-1 matrix to a filter F in order to save as much computation time as possible.

In order to do this, we take advantage of work by Eckert-Young in computing the best rank-m approximation for a matrix. Note almost every filter ever used in convolution is a square matrix and so under this, any $F \in \mathbf{R}^{n_f}$ can be written in the form of $F = \sum_{i=1}^{n_f}\sigma_i\mathbf{u}_i\mathbf{v}_i^T$ where σ_i is the ith largest singular values of F and $\mathbf{u}_i$ and $\mathbf{v}_i$ are orthonormal sets of vectors which correspond to each σ_i. Eckert-Young gives that the best rank-m approximation for such an F (assume $m \leq n_f$) occurs at $F'_m = \sum_{i=1}^{m}\sigma_i\mathbf{u}_i\mathbf{v}_i^T$.

Thus, we can extract the best rank-1 for F by computing the SVD of F (which can be done using numpy's built-in functions) and using $\sigma_1\mathbf{u}_1\mathbf{v}_1^T$. For simplicity, throughout the rest of this paper and in the code, we will just combine $\sigma_1\mathbf{u}_1$ into f and $\mathbf{v}_1^T$ into g to get the consistent notation used so far for a a rank-1 approximation of F as $F'_1 = fg^T$. Essentially, to get our outer product, we are just mulitplying in the σ_i terms with the $\mathbf{u}_i$ vectors since this doesn't actually modify the results in any way.

The important item here is that rank-approximation isn't restricted to just computing rank-1 approximations and can actually done to approximate the best rank-m approximation for any $m \leq n_f$. In fact, the larger m is, the closer the resulting approximated matrix will be to the original. However, it does mean the approximated matrix has higher rank, meaning the approximate filter would have higher rank and thus would require more computation using the formulation previously provided. At the same time, an approximate filter closer to the original would mean this filter can be used to deliver higher accuracy than a lower-rank filter.

The reasoning for this is that the original filter has been derived after many rounds of training. It's values exist because they perform almost as good as possible under the given model. Approximating these values is likely to cause a dropoff in performance. However, approximating these values can also speed-up the model's processing process.

This is the very crux of the experiments, code, data, and results the rest of the paper from here on out explores. While there is no more pure/theoretical math or proofs, everything beyond this is purely application of the material covered to this point.

The goal for the code and experiments is to see how much speed-up is derived from approximations at different rank-levels in comparison to how much accuracy dropoff is created. Further, how do these different rank-levels perform in both speed and accuracy against a regular convolution method (expected to be much slower) and even (as a bonus) against PyTorch's built-in convolution function and function which automatically runs the whole model.

X. The Data

The data we utilize comes from the CIFAR-10 Dataset, which consists of images of 10 different objects, each image comes pre-classified into one of the 10 classes. Each of the images is fairly small with a size of $32x32$ meaning the total pixels in each of the images is 1024. We get this dataset which has already been split into a train dataset of 5000 images in each class (50000 total images) and a test dataset of 1000 images in each class (10000 total images) from the following location: https://github.com/YoongiKim/CIFAR-10-images ($Citation2$). This comes from GitHub user YoongiKim and has been cited in the Works Cited.

Further, we turn each of the images into a 32x32 grayscale image so that our classification/model process is easier. This is instead of getting 3 32x32 matrices representing the RGB color values. Using a single grayscale value matrix is more efficient in terms of this and reduces complexity. Since the goal of this project isn't necessarily to maximize model accuracy and instead just get a decent model trained, grayscale images will do.

One of the classes is an "automobile" class. As an example of a before and after image (before and after grayscaling it), below is an example of a car picture before (meaning it is full color) and after grayscale is applied (note that image to grayscale and matrix conversion is done through the skimage library in python). Figure 3 is before grayscale and Figure 4 is after grayscale.

Fig. 3. Before grayscale (Full color)

XI. The model

Due to hardware limitiations and the lack of any provided resources such as online credits to run this model, the model that had to be used needed to be relatively simplistic. Once again, the goal of the project wasn't to create an extremely robust model but to just take a decently trained model and compare accuracies and times.

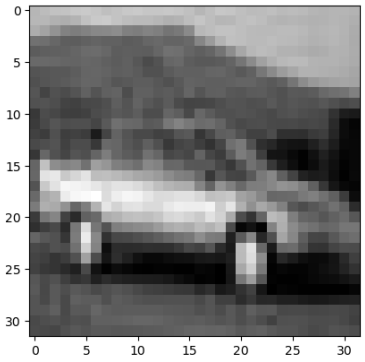

Fig. 4. After grayscaling

Under this background, the CNN model that was utilized and trained followed the following structure:

1) Convolution Layer 1 with a trained $F_1 \in \mathbf{R}^{5x5}$ and a trained bias $L_1 \in \mathbf{R}^{5x5}$ which is added at the end of the convolution.
2) ReLU Activation Layer (setting min value to 0)
3) Max Pooling Layer (Drops size by 0.5x on each dimension)
4) Convolution Layer 2 with a trained $F_2 \in \mathbf{R}^{5x5}$ and a trained bias $L_2 \in \mathbf{R}^{5x5}$ which is added at the end of the convolution.
5) ReLU Activation Layer (setting min value to 0)
6) Max Pooling Layer (Drops size by 0.5x on each dimension)
7) Flattening layer which just flattens the 2d matrix up to this point into a single vector
8) Trained linear weights with bias layer
9) ReLU Activation Layer (setting min value to 0)
10) Trained linear weights with bias layer which outputs a vector of size 10 representing logits for the 10 possible classes

At the end, we simply take the largest of the logits and get the predicted class by the model as the class which this logit value corresponds to. In this manner, passing in our 32x32 matrix which represents the grayscale image into this model goes through this process to output a vector of size 10, which then helps make the final predicted classification.

The components that are changed over time through training and backpropogation occur in steps $1, 4, 8, 10$ where the weights for these steps/layers change.

While this model is not overly-complex, it is a shortened/condensed version of many classic well-performing CNN models that are used throughout industry. Despite its simplicity, the model contains two convolution layers (which is what this project is really focused on) which do a majority of the heavy-lifting in terms of getting the image to the point of classification. Thus, this model should provide a good

benchmark to help understand and evaluate the final data and results.

The code for the model was created with help in some functions from an article written by Adrain Rosebrock which describes components of utilziing PyTorch for CNN tasks (*citation*3).

XII. Training and Testing Process

In order to conduct the training and testing process, there were 250 total EPOCHS (because this was about as large as the hardware available could handle and this took about 2 consecutive days of non-stop running to manage). Each single EPOCH consisted of retraining the model over ALL the test images and then testing the model over 7 different testing methods.

Firstly, each epoch's training process consisted of first shuffling all 50000 training images in order and then batching them into groups of 64 and then conducting batch training on each of these batches (forward then backward propagation) in order to better the weights of the layers of the model. Each time, the total accuracy on these training images and total loss (utilized the cross entropy loss, which consists of taking the softmax of the result and then computing negative log likelihood on this result) were logged for later results checking.

During each epoch, after the training process, 7 different methods were tested. Each of these testing methods went one-by-one through the 10000 testing images in order (since no weights modification is done the order doesn't matter) and the total time and total accuracy were computed for each method and logged for later analysis.

The first method that was tested (which is designated as "rank:-1" throughout the results portion) is simply testing through PyTorch itself. Since the model was built using PyTorch this testing was the simplest and consisted of just passing each image through the model as regular. Note that this is the general method of testing used throughout real industry and is thus expected to yield the best times (and obviously will have the max accuracy compared to any approximation).

The next method that was tested was using a numpy-based regular convolution method (designated as "rank-0" throughout the results portion). This convolution method is implemented in the naive manner but retains all the model's weights values perfectly. Thus, this is expected to be significantly slower than the PyTorch method but should have the exact same accuracies due to using the exact same values (this occurs in the results). As a reminder, this regular convolution method's loops are pythonic loops (implemented in python), while PyTorch's loops occur all in C++, which is why PyTorch (while having the same process) will be much faster than this method.

Finally, the final 5 methods tested are done using "rank-$(1, 2, 3, 4, 5)$" approximations (one at a time) of the convolutions where the computations are done using the formulas derived throughout the paper using matrix multiplication instead of loops. Note that since our filters are 5x5, the accuracy of the rank-5 method should be identical to the regular convolution method and the PyTorch method due this approximation just maintaining the filters (this occurs in the results).

As a reminder, note that PyTorch also implements convolution in the naive manner, but due to implementing it in C++, the loops will be MUCH faster, meaning it will be faster than the "regular convolution" method (correspondin to

XIII. Results (Fig 5-9)

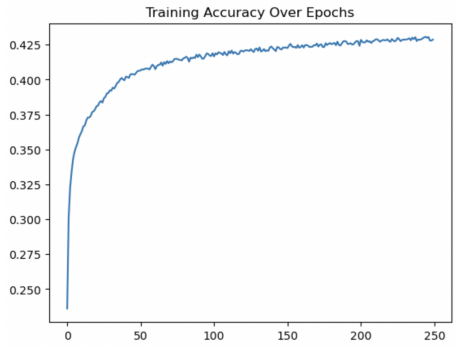

Fig. 5. Training Accuracy over Epochs

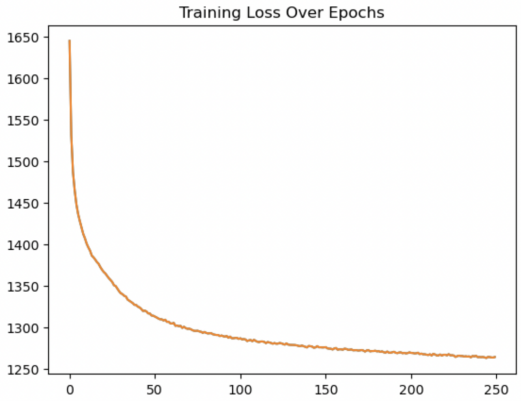

Fig. 6. Training Loss over Epochs

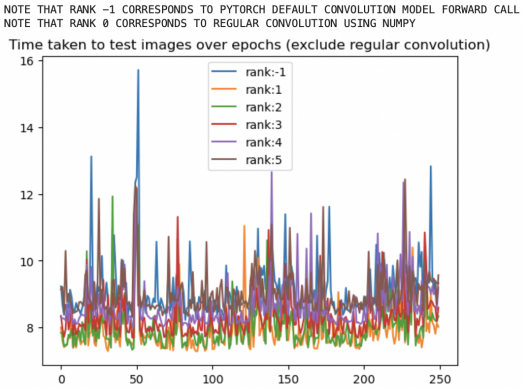

Fig. 7. Time Taken To Test Over Epochs(All but reg. conv.)

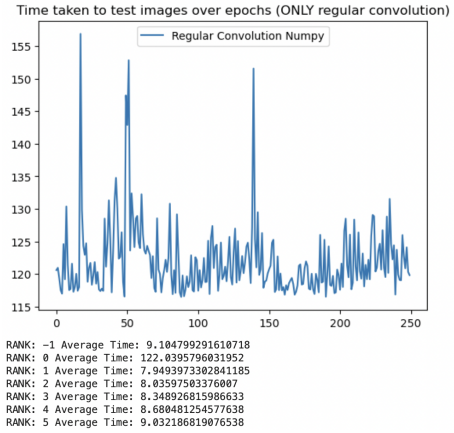

Fig. 8. Time Taken To Test Over Epochs(ONLY reg. conv.) Average Time Taken to Test

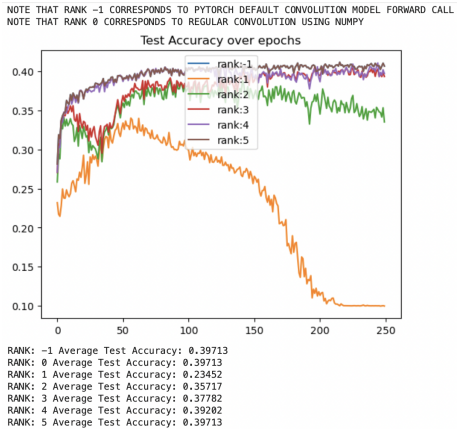

Fig. 9. Testing Accuracy Over EpochsAverage Testing Accuracy

XIV. Results Analysis

Overall, the results are as expected and very promising. To begin, note that the training accuracy and loss look to be what would be expected. The loss decreases at a slower and slower pace as the model goes through more epochs, and the accuracy increases at a slower and slower pace as the model goes through more epochs. The training accuracy ends at around 42.5% accuracy on the final epoch. While this isn't great, given the hardware/computation restrictions along with the fact that the model was very basic this is enough to make comparisons. Since there are 10 classes random guessing would be 0.1 accuracy, so this model is about 4x better than that. Again, the purpose of the project was NOT to make a wonderful and accurate model (that is a whole another project entirely), but to compare accuracies and timings, which we can now explore.

First, we can note the obvious. The regular convolution method implemented with Numpy and loops in python is VERY slow. It's so slow in fact that on average its over 13 times slower than the nearest method's average time to finish an epoch of testing. The accuracy, however, does match PyTorch's accuracy (which is exactly what we expected), and thus is tied for the highest accuracy out of the all the methods tested.

Looking at the PyTorch default convolution/model process next. It's average time to process the 10000 test images is around 9.1 seconds which is extremely good. As stated before, this also reaches the highest accuracy out of any tested methods at 0.39713.

Now we can begin to examine the rank-approximated convolutions. For the most part, their timings are as expected. The lower rank approximations completed each testing epoch on average faster than the higher rank approximations. While there is a lot of instability in the timings from epoch to epoch (which is more so due to the hardware itself and other programs), over a large number of epochs, the average will come out. Thus, the average time data is likely reliable. Note that the average increase in time per epoch is about 0.3 seconds for every rank we increase (except only 0.1 seconds going from rank-1 to rank-2). This is also about expected since each time we increase the rank, the number of computations/operations increases by a linear amount. The quickest out of these was obviously using rank-1 approximation which finished each testing epoch on average in approximately 7.95 seconds. In accordance with expectation, this is followed by rank-2 at 8.04 seconds, rank-3 at 8.35 seconds, rank-4 at 8.68 seconds, and finally rank-5 at 9.03 seconds.

One may notice, that the time for the rank-5 approximation convolution on average was actually better than the PyTorch time on average (9.03 vs 9.1). While this is a somewhat small margin (beats by about 1%), it is important because the testing method we used using PyTorch is the general method of testing that is used throughout industry in real-world settings and applications. Further, PyTorch's entire model is written in C++ while we used a significant amount of Python code (and a few small python loops as well). Despite this, using the matrix multiplication formulation for a CNN allows our convolution mechanism to actually outperform PyTorch in speed.

On top of this, because this is the rank-5 approximation, it is an exact approximation for F_1/F_2 since F_1 and F_2 are each 5x5 so this is just an SVD of them instead of any approximation. Thus, the rank-5 approximation uses all the same values as the PyTorch testing mechanism does, which is why it comes out with the exact same accuracy as well at 0.39713. In other words, the rank-5 approximation method gives a faster time with the same accuracy/results as PyTorch.

Looking at the other rank-approximation's accuracy results, a majority of them are what is expected. Rank-4 approximation would likely be very very similar to the exact matrix F, and the accuracy is 0.39202, only 0.5 off the best possible. Further, rank-3 and rank-2 see some expected drop-off to 0.37782 and 0.35717 respectively. Rank-1 sees heavy dropoff in accuracy to 0.23452 on average, but more interesting is its graph.

Notice that the rank-1 approximation's testing accuracy actually consistently rises to about epoch-50 and then suddenly begins dropping until it reaches basically just 0.1 guessing at epoch 200. This is telling about how CNN model's work

as a whole. The idea here is that the model is continuing to likely make larger changes to F until about epoch 50 (which is about where even training accuracy begins to stagnate) at which point it begins to get more specialized. Essentially, from epoch 50 and beyond, many of the large improvements that can be made, have been made. Improvements from here on out might just be modifying even 1 element beyond minute impact. These specialized changes are going to drag the F_1 and F_2 filters further and further from being close to rank-1. As such, a rank-1 approximation is likely getting less and less close to the model. The other important component of this is that there are 2 linear layers who are also constantly being trained based on how the filters F_1 and F_2 are being modified and trained. If the rank-1 approximation is further from F_1 and F_2, it's resulting values are going to be less close to what is being expected by the linear layers as well. Overall, this dropoff is interesting and justifiable, but the magnitude with which the dropoff occurs is very surprising (essentially going down to random guessing at the end). Note that there is also some dropoff for rank-2 but this isn't really witnessed for other approximate models.

XV. Conclusion

In terms of conclusions, one of the big one is that these experiments help show that taking the newly formulated matrix multiplication approach to convolutions is significantly better (even without any approximation as shown by the rank-5 case) in terms of time efficiency than the naive convolution implementation. This is especially true if the naive implementation is in python and using pythonic for loops. However, the experiment showed that this can also be true even against PyTorch as PyTorch also takes the naive convolution approach (though in C++). The quick reasoning for why PyTorch does this is that PyTorch can handle convolutions for 3, 4, and even higher dimensions, which this paper does not concern. In any scenario, in the 2-dimensional setting, the matrix multiplication formulation to convolution DID outpace PyTorch.

However, there is also some merit to using convolutions approximations as well. There were speed-ups witnessed every single time the rank for the approximation was dropped. At certain levels the accuracy did not drop by much. Take the rank-3 approximation for example. This performed nearly 10% faster than PyTorch and an accuracy dropoff of only 2%. Or take rank-4 approximation which performed about 5% better than PyTorch and had an accuracy dropoff of only 0.5%. Depending on what application one may be utilizing a model for, utilizing a rank-approximated CNN can be significantly beneficial in many settings where time is extremely vital. In areas such as quantitative finance or time is extremely important and speedups of 10% might be worth a small 2% drop in accuracy. While there is no one-size-fits all conclusion to which approximation might be best, based on an application-to-application basis, one can experiment with different rank-approximations for their model and see which tradeoff fits best.

As a general conclusion, the paper displayed three main ideas. The first is that utilizing matrix multiplications in convolutions instead of naively using the sliding window approach can and does speed-up AI/ML models. Secondly, rank-approximations to convolutional filters in the CNN also speeds-up the model even further. Finally, the convolutional filters which get trained in CNNs are complex, and thus there is a dropoff in accuracy for dropoff in time (increase in speed), creating a tradeoff opportunity for time against accuracy.

XVI. Future Work/Limitations

In terms of future work, the biggest way to redo this experiment to get even better and more promising results would be to increase one's hardware capabilities. A lot of these experiments were bottlenecked by hardware limitations so future work can be done using the same experiments but with better hardware. By doing this, one can repeat the same experiments but with more epochs, larger models, better images, and even higher-dimension filters. This can give even more detailed results. This would hit on a lot of the limitations of this paper (stemming primarily from hardware constraints) such as the model size, model complexity, image sizes, number of images, amount of testing, number of epochs, filter sizes, etc.

Additionally, some work could be done to try and code some of these components of the new matrix multiplication formulation for CNN in C++ or C, and re-compare to PyTorch and other libraries to see differences again.

Another component that this paper does not explore is actually utilizing rank-approximations during training itself. This is outside the scope of the paper (and even the class since it deals with more AI theory) but can be explored as to how it may affect training performance and training time as well.

XVII. Code Folder Guide

CIFAR-10-Images: Folder which contains a train and test folder, each with 10 folders for the 10 images classes. Each of these class folders contains numbered images

CNNRunFull10.py: File which is the entire script for running and saving the results that were contained in this paper. This scripts runs the 250 epochs over the CIFAR-10 dataset contained in the CIFAR-10 Folder.

loss_acc_time_data_10.txt: File which is where the results from running the python file above are logged into. The results are then expected to be retrieved later and analyzed via data analysis.

DataAnalysisPortion.ipynb: The Jupyter Notebook file where the data analysis portion was completed. This is where the code that took in the results data from the .txt file and created viewable graphs and average data is located.

XVIII. Works Cited

1) https://www.researchgate.net/publication/329746877_IPC-Net_3D_Point-Cloud_Segmentation_Using_Deep_Inter-Point_Convolutional_Layers
2) https://github.com/YoongiKim/CIFAR-10-images
3) https://pyimagesearch.com/2021/07/19/pytorch-training-your-first-convolutional-neural-network-cnn/