Math 385 Final Project Report

Landon Minkow

Convolutional Neural Networks

1 PART 1

Why Are Big Data Matrices Approximately Low Rank?
Madeleine Udell and Alex Townsend

ChatGPT Summary Reflection:

Summaries can be tailored to serve different purposes. When it comes to highly technical papers, I take more away from a summary that focuses on the effects of the discovery more than the formulas and proof. ChatGPT describes the paper from a *why* perspective — it cites reasons why the low rank structure appears in real-world data, stating, "The authors argue that low-rankness is not a mysterious empirical coincidence but rather a natural consequence of several common generative mechanisms." However, in the introduction of the paper, the authors claim that despite researchers' arguments, "Our main theorem shows that low rank structure can persist even without an underlying physical reason" (Udell and Townsend 146). While this paper does explain the importance of using low rank techniques in machine learning, the authors (despite the paper's title) seem more focused on the *so what?* with regard to the existence of low-rank structure. Whereas the ChatGPT summary labels measurement limitations, smoothness, and latent variable structure as the major causes, I did not identify these terms myself until Section 2.4 of the paper, meaning ChatGPT extends beyond the introduction of the paper. That being said, I struggled to fully understand these definitions in the paper, so the high level summary by ChatGPT helps me see their significance. What the ChatGPT summary does not explicitly mention is the Johnson-Lindenstrauss lemma and the concept of upper bounding the rank of a matrix using LVMs, which to me was the main takeaway. Overall, the ChatGPT summary provides an appropriate level of detail (no need to give explicit values or formulas) for the paper, but it implies certain explanations exist for the low-rank phenomena, when in reality, there is more attention placed on the usage of that low-rankness.

Rewritten Summary:

With the emergence of big data, solving real-world problems requires algorithms that are both cost-efficient and time-efficient. In their 2019 paper, Madeleine Udell and Alex Townsend highlight the frequent occurrence of low rank matrices and the power of exploiting that low rank structure to develop fast algorithms. The authors explain that any sufficiently large matrix can be approximated within a desired accuracy threshold by a lower rank matrix. Moreover, as the dataset scales upward in size, the value of the rank will remain relatively contained. In particular, nice latent variable models that express hidden structures in the data lead to matrix ranks that grow only logarithmically compared to the dimensions of the matrix. The effect of this in real terms is that large, complex datasets can be represented by smaller datasets while still capturing the essence of the original data. Thus, data scientists can better allocate computational resources and solve problems quickly. Specifically, truncated singular value decompositions (SVDs) can help identify a smaller rank approximation for a matrix, which in turn requires less mathematical operations in computer algorithms. Additionally, principal component analysis (PCA) uses the rank of a matrix to extract the dominant components, allowing scientists to examine the variables that most impact a given outcome. The key behind low rank matrices is that they approximate the original matrix well. Employing such algorithms on the smaller dataset lead to optimal solutions that extend to the original.

2 PART 2: Convolutional Neural Networks

2.1 Introduction

Consider a traditional fully-connected neural network. Each input node within a given layer connects to each output node. As the size of the input data vector grows, so does the number of node-to-node connections in the network, and hence, the number of relationships that are calculated.

Take a relatively small 128 x 128 image as input, for example. That would be 16,384 pixels in size. If the photo is not grayscale, then this input would be 128 x 128 x 3 = 49,152 pixels, where the dimensions are multiplied by 3 for the RGB channels. This means there are over 49,000 inputs to the network, which get flattened into a one-dimensional vector. Say the first hidden layer contains 512 nodes; then there are already 25 million parameters. Evidently, networks grow quickly for complex inputs.

This is where Convolutional Neural Networks (CNNs) come in. CNNs are particularly effective for visual, grid-like datasets. The underlying theory, which will be further investigated below, is to define a smaller matrix (kernel) that covers separate portions of the image one by one. Suppose that kernel is 3 x 3 x 3 in size (much smaller than the 128 x 128 x 3 image). Thus, we begin with 27 weights. If we want to detect 20 features in this layer, there are 27 x 20 = 540 parameters—requiring far less computation to learn than before.

In this report, we will describe the structure of CNNs and when they provide efficient alternatives to traditional networks. Along with code samples, we will highlight how CNNs relate to Math 385 course content, specifically with respect to solving inverse problems.

2.2 What is Convolution?

In classification and other machine learning tasks, the input of a neural network is often a multidimensional array of data. For now, we will focus on inputs in the form of an image, since that is the context in which CNNs really shine. Most images can be thought of as 3-way tensors of dimensions m x n x p, with (m x n) representing the image size and the p slices being the RBG channels.

In an image, the m x n pixels are each assigned an x_i and stacked in a vector. This vector becomes the input to the first layer. Unlike the fully connected networks we diagrammed in Math 385, where each input x_i of the input vector is connected to each node in the first hidden layer by a weight value $w_{ij}x_i$, a CNN takes a **patch** of the image and runs a **kernel** on it. This kernel might be a 3 x 3 x 3 tensor, for instance, and each element is a weight.

The kernel starts over the top left patch of the image, and a dot product is calculated between the kernel and the pixels of the patch. This dot product's output denotes the strength of a particular feature in the current location in the image.

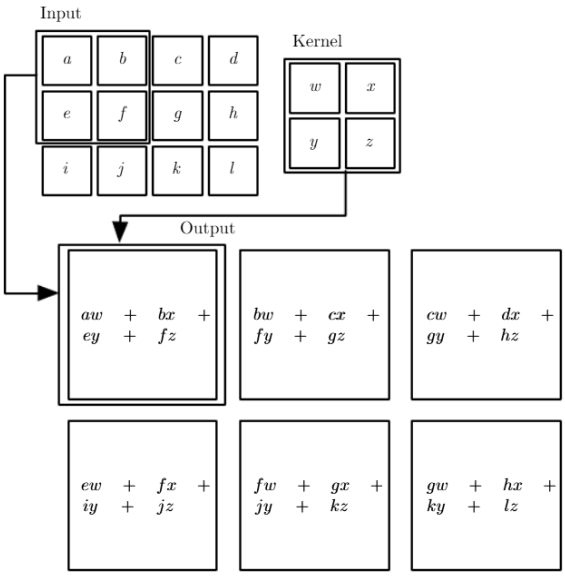


Figure 1: **Note:** This image comes from Ch. 9 of *Deep Learning*, p. 330. The 2 x 2 kernel starts in the upper left patch of the image, and the element-wise dot product is taken. The kernel then moves to the right by 1 space, and is multiplied with the next patch, until it has covered the entire image (6 patches).

.

The dot product over that first patch will then be placed at the (1,1) position of a **feature map**. The kernel then slides to the next patch, and that dot product fills the (1,2) position. The kernel continues to slide across the image, one patch at a time. This is **convolution**.

2.3 How do CNNs work?

Once the kernel has covered the entire image, the feature map is completely filled in. One parameter, the stride length, affects the step size of the kernel; so a stride of 2 would slide the kernel over two spaces at a time (compared to stride length 1 in the above image). The indices of the feature map denote the activation level, or presence of a given feature at that location of the image. If the user designs the model to detect 20 features in the first layer, for instance, 20 separate kernels are created, leading to 20 different feature maps.

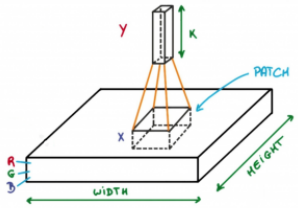


Figure 2: **Note:** This diagram comes from the GeeksforGeeks "Introduction to Convolution Neural Network" article. The kernel k slides across the plane x, where it is multiplied by each patch underneath. Note that the element-wise multiplication over a given patch captures all three RGB channels of the image.

At this point, suppose a CNN has detected 20 features (like edges and basic shapes) each with a feature map showing where those details appear in the image. Now, an **activation function** is applied element-wise across those feature maps. As mentioned in class, *ReLU* is a common choice. The ReLU function sets any negative element in the feature maps to 0, thus introducing nonlinearity into the model without changing the dimensions.

Then, there is the **pooling layer**, which captures local patterns and reduces dimensionality. Consider a maximum pooling with 2x2 filters and stride length 2:

Maximum pooling takes the maximum element among nearby neighbors. With 2x2 pooling, the maximum of every 2x2 submatrix is taken, reducing the size of the feature maps by half. The 20 feature maps, now smaller, are stacked together and passed to the subsequent layer of the CNN. The first convolutional layers of the CNN tend to look for simple features, like edges. In the next layer, new filters will slide across to detect more abstract details than before (corners, shapes), with the depth p of the kernel being 20. After N layers, the outputted feature maps are flattened into a fully connected output layer, and the classification is done.

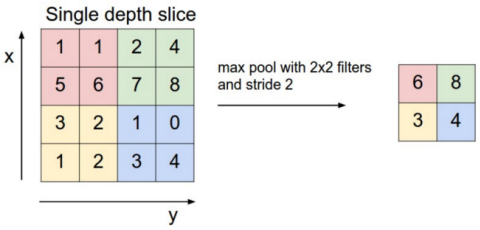


Figure 3: **Note:** This example comes from GeeksforGeeks "Introduction to Convolution Neural Network" article.

2.4 What Makes CNNs Effective?

The book *Deep Learning: An MIT Press Book* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville argues that CNNs improve on other networks through sparse interactions, parameter sharing, and equivariance.

2.4.1 Sparse Interactions

In traditional neural networks (like those discussed in Math 385), matrix multiplication is performed between all inputs and outputs in a layer. In CNNs, however, the kernel is much smaller (less pixels) than the image. Because that same kernel is used throughout the image, it contains a fixed and significantly smaller number of weights to be learned. In other words, with m inputs and n outputs to a layer, we only use k x n parameters rather than m x n, where k << m. Consequently, the model requires less operations and memory space.

2.4.2 Parameter Sharing

The kernel represents a single set of weights for detecting a given feature. That kernel gets reused as it slides across the image, so the parameter values that detect a feature at one location are the same values that detect the feature elsewhere. This is in contrast to traditional networks, where weights are only used once.

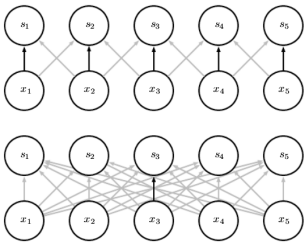


Figure 4: **Note:** This figure comes from *Deep Learning*, Ch. 9, p. 333. The diagram compares the fewer interactions in a convolution layer with a size-3 kernel (top) to the fully-connected network (bottom) using matrix multiplication.

2.4.3 Equivariance

A function is equivariant if changes to the input lead to equivalent changes in the output. Along with the shared weights of the kernel, the pooling stage of CNN layers allows for generalization. With maximum pooling, the maximum value summarizes all nearby elements in the feature maps. In other words, it provides a local representation that is less likely to change if a feature moves slightly within the image.

2.5 CNNs in Action

In the first demo, we examine an image of a simple number "7" from the MNIST test dataset. We define the following small CNN model:

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5, padding=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, padding=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.fc1 = nn.Linear(32*7*7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

The model uses 5 x 5 kernels to produce 16 feature maps in the first layer. It uses padding of 2 (adding extra pixels to the outside of the image to preserve dimensionality when the kernel reaches the boundary). After 2 x 2 max pooling, the second convolutional layer outputs 32 distinct feature maps, which are then passed into the fully-connected linear layers before classification.

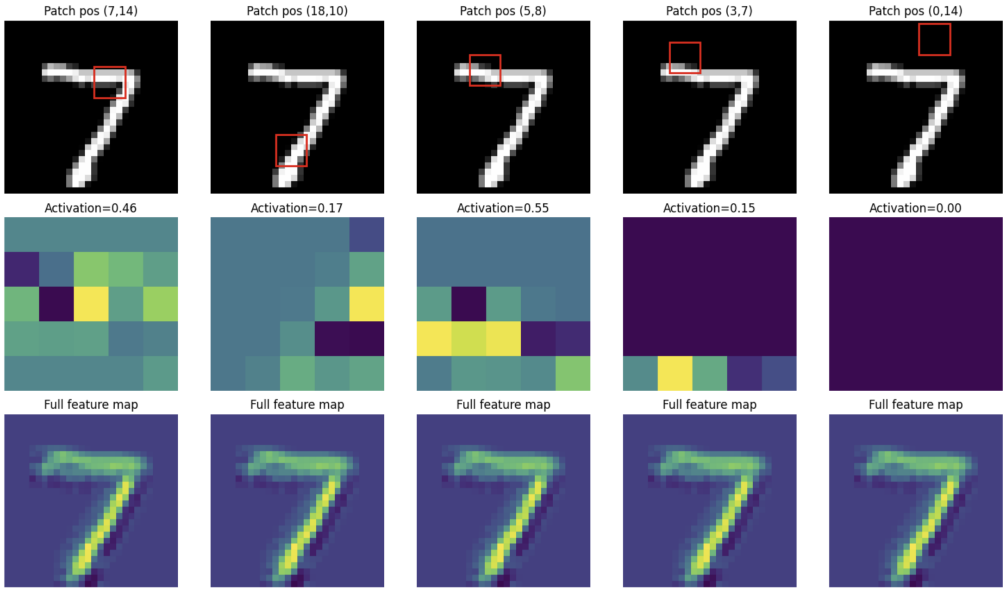


Figure 5: We randomly select 5 example patches from a single convolution. One filter/kernel slides over the image, tracking one particular feature. The activation magnitude depends on the strength of the feature at the patch's location, while the sign indicates the orientation of the feature compared to the kernel. The corresponding feature map (bottom row) is produced by the kernel.

In a second example, we run a simple CNN with 3 x 3 kernels and padding of 1. This time, however, we

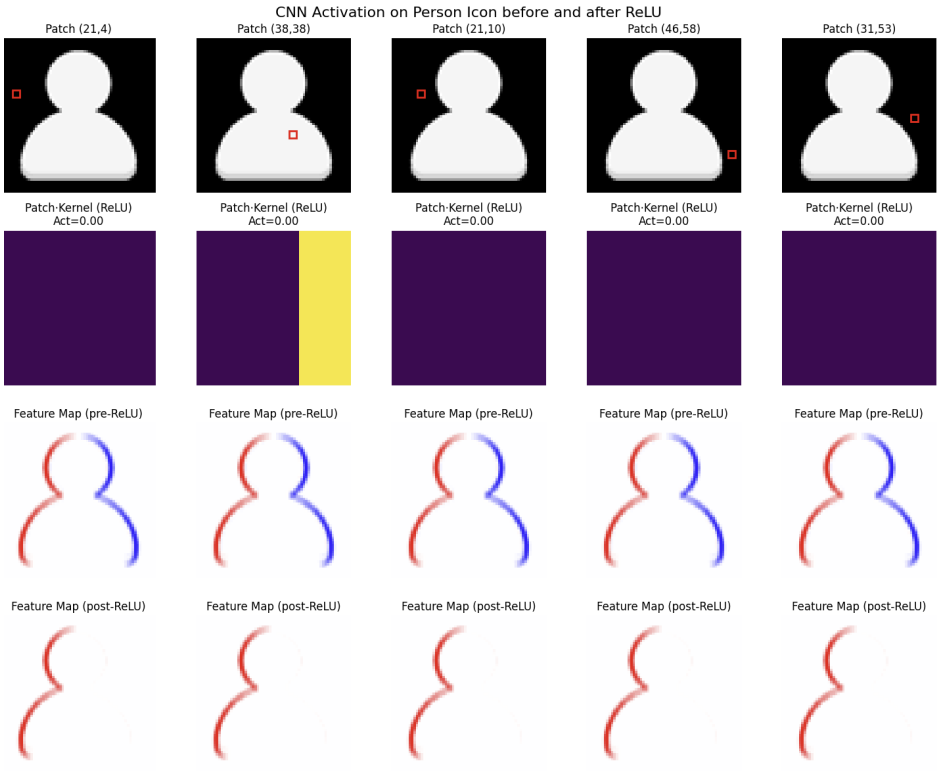
define a custom kernel to detect vertical edges:

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Recall that the level of activation over a given section is given by the dot product between the kernel weights and the pixels:

$$A(i,j) = (I * K)(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n) \cdot K(m,n)$$

Using what we've learned about matrix multiplication, the left column of the kernel multiples pixels by -1, while the right multiplies by +1. In other words, the kernel captures sharp contrasts, which indicate vertical edges or changes in color, as it moves sideways across the image.



In the third row, we observe the feature map outputted for this particular kernel. The left side indicates the indices where the multiplication between the kernel and patch is positive, while the right represents the indices with negative outputs. Before applying ReLU, the feature map shows both the positive and negative activations. After ReLU (last row), we are only left with the positive outputs, since the ReLU activation function takes the max(0,i) at each index.

2.6 Connections to Math 385: CNNs for Inverse Problems

In class, we examined inverse problems in Section 2.5. In particular, we looked at deblurring, where the actual v. blurred signals are connected by the following equation:

$$g(s) = \int_0^1 exp\left(\frac{-(s-t)^2}{\sigma^2}\right) f(t)dt, s \in [0,1]$$

We called the kernel K(s,t) a **convolution kernel** and claimed the observed signal $g(s)$ is a convolution between the true signal $f(t)$ and a **point spread function**. The kernel helps map between the true signal

and the blurred signal. Really, the true signal f is the collection (summing up) of each individual point source. This is analogous to how a kernel in a CNN model slides across an entire image, summarizing the contributions of each patch/location it covers.

2.6.1 "A Review of Convolutional Neural Networks for Inverse Problems in Imaging"

This leads into a discussion of how effective CNNs can be in solving inverse problems. In their summary of CNNs in imaging-based inverse problems, authors Michael McCann, Kyong Jin, and Michael Unser consider a training-based approach to learn an optimal set of parameters while minimizing the error (in contrast to an objective function).

In this case, CNNs are advantageous because the forward model (from class – computing the observed signal given the unknown function) is built on convolutions, and as described earlier, convolutions are relatively small compared to looking at the entire signal, thus allowing for faster training. While the authors indicate CNNs have only shown minimal improvement over alternative strategies in deblurring problems, the computational efficiency they offer might still be beneficial, even at the expense of improved performance. However, training image-recovery models requires learning from noisy samples. Thus, it is essential to avoid overfitting. In Math 385, we introduced specific regularization parameters to balance the oversmoothed solutions with noisy ones. For example, we used the Tikhonov Solution:

$$\min_x ||\mathbf{Ax\text{-}b}||_2^2 + \lambda^2 ||\mathbf{x}||_2^2$$

Furthermore, this is where data scientists need to be intentional about the kernel/patch sizes or number of convolutional layers they employ in their CNN models.

This paper also compares the use of regular versus Stochastic Gradient Descent for learning. When learning about SGD in Math 385, we explained that using one sample y_j to compute the gradient (instead of considering the entire dataset) may lead to faster learning.

2.6.2 "When to use Convolutional Neural Networks for Inverse Problems"

A second paper by authors Nathaniel Chodosh and Simon Lucey explores the limitations of CNNs for certain types of inverse problems. Mainly, they claim that for CNNs to solve inverse problems effectively, a convolutional compressibility assumption must be satisfied by the input.

Recall that within a CNN's convolutional layer, filters (kernels) slide across the image to detect specific features throughout the input. In simple terms, the input image or function is convolutionally compressible if features are dispersed in local patterns in certain parts of the image – and not in most other locations. In other words, the features are distributed sparsely across the input. These local trends are further drawn on by the aforementioned pooling layers of the CNN, which summarize the neighboring activation levels (such as maximum pooling of each 2x2 chunk) or activation functions like ReLU, which force lower activation levels to zero. For example, the Toeplitz matrix from Math 385 is attractive in this sense because it has a local structure that is preserved across the matrix.

CNNs therefore assume, according to the paper, that samples of the input are similar to neighboring samples, and those similarities persist even if parts of the input are altered. To challenge this, the authors demonstrate that CNNs perform worse on a block matrix, in which patches of the larger image act as independent pieces. In this case, the CNN had to be retrained for each time the learning rate was adjusted, challenging its ability to generalize to certain inverse problems.

2.7 References

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning: An MIT Press Book.* 2016, pp. 326-366.

- Michael T. McCann, Kyong Hwan Jin, and Michael Unser. "A Review of Convolutional Neural Networks for Inverse Problems in Imaging". https://www.emergentmind.com/papers/1710.04011. 2017, pp. 1-11.
- Nathaniel Chodosh and Simon Lucey. "When to use Convolutional Neural Networks for Inverse Problems". https://ieeexplore.ieee.org/document/9157603. Computer Vision Foundation, 2020, pp. 8226-8235.
- GeeksforGeeks, "Introduction to Convolution Neural Network," *GeeksforGeeks*, https://www.geeksforgeeks.org/machine learning/introduction-convolution-neural-network/ (accessed December 15, 2025).
- GeeksforGeeks, "Math Behind Convolutional Neural Networks." *GeeksforGeeks*, https://www.geeksforgeeks.org/deep-learning/math-behind-convolutional-neural-networks/ (accessed December 15, 2025).
- Madeleine Udell and Alex Townsend. "Why are big data matrices approximately low rank?." SIAM Journal on Mathematics of Data Science 1.1 (2019): 144-160.