Training Report Day-2

Ankur (URN: 2302473)
Department of Computer Science and Engineering
Guru Nanak Dev Engineering College Ludhiana

June 27, 2025

Abstract

This report explores the foundational building blocks of neural networks with a practical focus on Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). It begins with an overview of RNNs and their applications in processing sequential data such as natural language. It then transitions into CNNs, highlighting their role in image classification and feature extraction through localized patterns using filters. The report further includes detailed implementation of 3×3 convolutional layers and max pooling from scratch using NumPy, illustrating how convolution and pooling reduce the input dimensions while preserving important features. Finally, it covers the Softmax layer and the Cross Entropy Loss function, which are essential for making predictions in multi-class classification tasks. This hands-on approach provides a strong conceptual and practical foundation for understanding and building neural networks.

1 Recurrent Neural Networks

Recurrent Neural Networks are the kind of networks that specialize in processing the **sequences**. That's why they are often used in applications such as the **Natural Language Processing (NLP)** because they can handle text very well.

A shortcoming of the vanilla neural networks or even the convolutional neural network is that they work with fixed size input and output. But an RNN does not work like that. A few examples of RNN are shown in the Figure 1

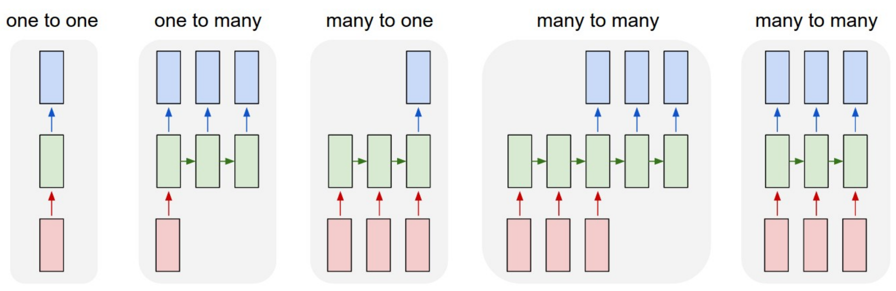

Figure 1: Examples of RNNs — Source

The blue nodes in the Figure 1 represent the input layer, the black ones are the hidden layers, and the green ones are the output layers.

This ability to process sequences makes RNNs very useful. For example:

- **Machine Translation** (*e.g. Google Translate*) is done with "many to many" RNNs. The original text sequence is fed into an RNN, which then produces translated text as output.

- **Sentiment Analysis** (*e.g. Is this a positive or negative review?*) is often done with "many to one" RNNs. The text to be analyzed is fed into an RNN, which then produces a single output classification (*e.g. This is a positive review*).

2 Convolutional Neural Netwoks

The **convolutional neural networks (CNNs)** or the **convNets** are designed to work with images. A classic use case of CNNs is image classification. For example, seeing an image and deciding whether it is an image of a cat or a dog.

Although a normal neural network can be used to accomplish the same task, using a CNN makes the task much easier. The reason for that is that the images are generally very large. For example, consider an image of size 224×224 pixels. If we use a normal neural network, then considering 3 color channels (RGB), there will be $224 \times 224 \times 3 = 150,528$ input features. Assume there are 1024 nodes in a typical hidden layer of such a network. Thus, we will have to train $150,525 \times 1024 = 154,140,672$ i.e., more than 150 million weights. The network will become very huge and difficult to train.

The second thing is that the position of the features can change to a great degree in the image. In that case, the neural network will fail to accomplish the task. The way we humans identify objects in an image or do classification is based on the localized features. We look for patterns in an area. CNNs also work similarly.

3 Convolutions

CNNs are neural networks that use convolutions. CNNs consist of filters, which can be imagined as 2D matrices. The output is produced by convolving the filter with the input image. This output is then further given to the next convolution block as input or is given to the output layer to output the result. The flow of convolutions is as follows:

- Overlay the filter on top of the input image
- Multiply the corresponding elements of the image with the filter and add them together.
- Add a bias to the sum
- place the result in the output matrix
- Repeat

The size of the input image, filter, and output matrices could be different. Different filters can be used to extract different kinds of information from the image. In the end, we obtain the output image as a result of going through these different filters. For example, consider the Figure 2 which shows a **vertical sobel filter**. This filter is used to extract the vertical edges from an image. Figure 2 shows a the result. Similarly the **horizontal sobel filter**

-1	0	1
-2	0	2
-1	0	1

Figure 2: vertical sobel filter

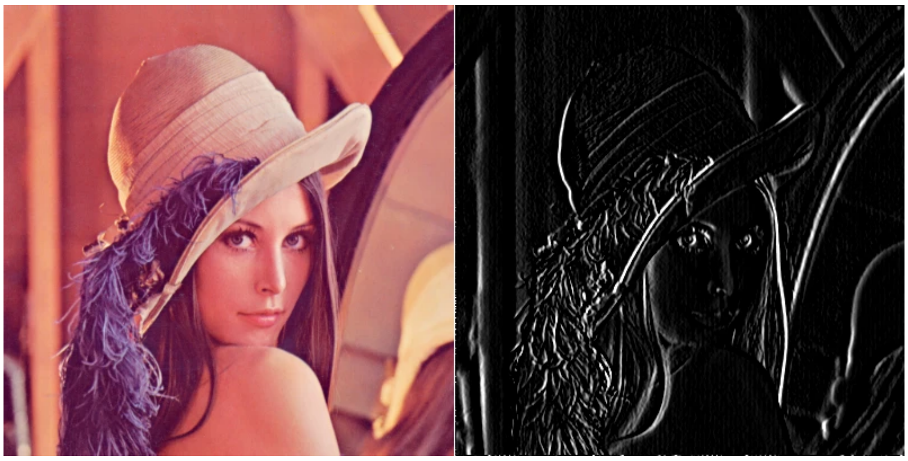

Figure 3: An image convolved with vertical sobel filter

1	2	1
0	0	0
-1	-2	-1

Figure 4: Horizontal sobel fiter

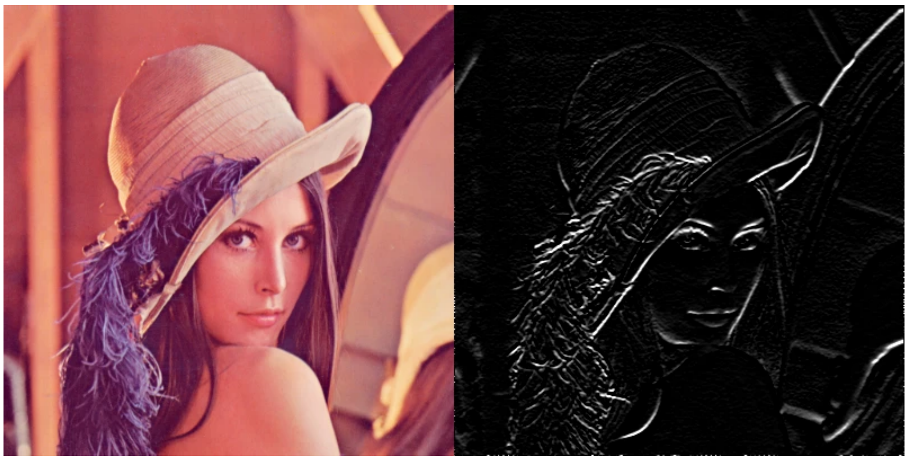

Figure 5: An image convolved with horizontal sobel filter

3.1 Implementing Convolutions

Below is the implementation of a conv class using 3×3 filters using numpy

```python
import numpy as np

class Conv3x3:
    """
    A custom Convolutional Layer that applies multiple 3x3 filters
    to a 2D grayscale image using valid padding (no padding).
    """

    def __init__(self, num_filters):
        """
        Initialize the convolutional layer.

        Parameters:
        - num_filters: Number of 3x3 filters to apply (output depth)
        """
        self.num_filters = num_filters

        # Initialize filters with random values.
        # Shape: (num_filters, 3, 3)
        # Divided by 9 to reduce the variance of initial weights.
        self.filters = np.random.randn(num_filters, 3, 3) / 9

    def iterate_regions(self, image):
        """
        Generator that yields all possible 3x3 regions of the image.

        Parameters:
        - image: A 2D numpy array (grayscale image)

        Yields:
        - A tuple (image_patch, row_index, col_index)
        """
        height, width = image.shape

        # Slide a 3x3 window over every valid region
        for i in range(height - 2):
            for j in range(width - 2):
                image_patch = image[i:i+3, j:j+3]
                yield image_patch, i, j

    def forward(self, input):
        """
        Perform the forward pass of the convolutional layer.

        Parameters:
        - input: A 2D numpy array representing the input image

        Returns:
        - A 3D numpy array of shape (height-2, width-2, num_filters)
        """
        height, width = input.shape

        # Initialize output array (valid padding reduces size by 2)
        output = np.zeros((height - 2, width - 2, self.num_filters))

        # Apply each filter to every 3x3 region
        for image_patch, i, j in self.iterate_regions(input):
            # Element-wise multiplication and summation for all filters
            output[i, j] = np.sum(image_patch * self.filters, axis=(1, 2))

        return output
```

4 Pooling

The neighboring pixels in an image tend to have similar values, so the conv layers will also produce similar outputs for the neighboring pixels. As a result, we have a lot of redundant data in the output of the conv layer, which does not provide any extra information.

So to overcome this we use pooling. Pooling reduces the size of the input by the *pooling* value. Suppose the size of an image is 4×4 and we use a pool size of say 2, then after pooling the size of the image will reduce to 2×2. The pooling can be done using simple operations such as max, min, and avg.

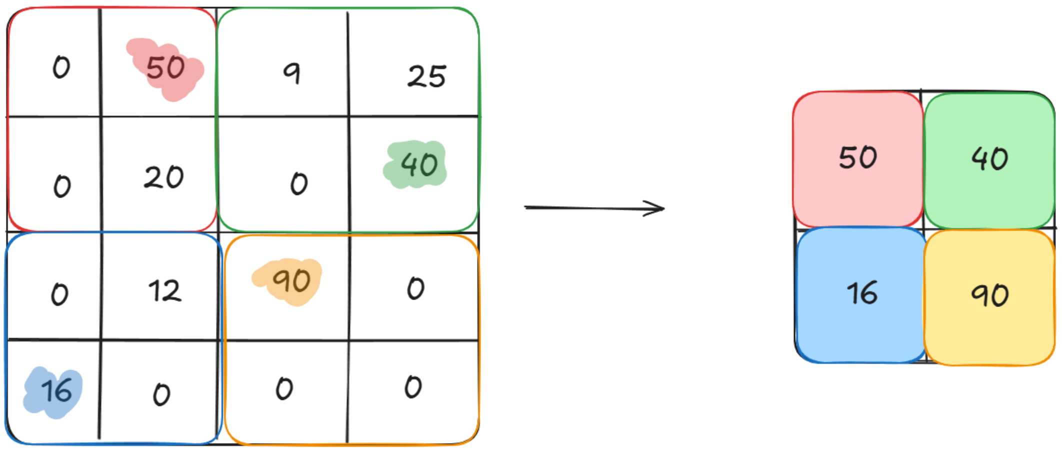

Figure 6: Max Pooling (pool size 2) on a 4x4 image to produce a 2x2 output

The code below shows an example of implementing *max pooling* from scratch using numpy

```python
import numpy as np

```

```python
class MaxPool2:
    """
    A max pooling layer with a 2x2 window and a stride of 2.
    It reduces the input spatial dimensions by half.
    """

    def iterate_regions(self, image):
        """
        Generates non-overlapping 2x2 regions from the input feature map.

        Parameters:
        - image: 3D NumPy array of shape (height, width, num_filters)

        Yields:
        - A tuple (image_patch, i, j), where:
          - image_patch is a 2x2xnum_filters region
          - i, j are the top-left indices in the output
        """
        h, w, _ = image.shape
        new_h, new_w = h // 2, w // 2

        for i in range(new_h):
            for j in range(new_w):
                # Select a 2x2 patch for all filters
                image_patch = image[i*2:i*2+2, j*2:j*2+2, :]
                yield image_patch, i, j

    def forward(self, input):
        """
        Performs the forward pass of max pooling.

        Parameters:
        - input: 3D NumPy array of shape (height, width, num_filters)

        Returns:
        - A 3D NumPy array of shape (height//2, width//2, num_filters)
        """
        h, w, num_filters = input.shape

        # Initialize the output array (dimensions reduced by 2)
        output = np.zeros((h//2, w//2, num_filters))

        # Iterate over 2x2 patches and apply max pooling
        for image_patch, i, j in self.iterate_regions(input):
            # Take max over height and width (axes 0 and 1)
            output[i, j] = np.amax(image_patch, axis=(0, 1))

        return output
```

5 Softmax

Softmax layer is a standard layer that I used in the multiclass classification problems, like the one that I am going to develop during this training. This layer is what gives a neural network the ability to make predictions. The **softmax** layer is a *fully connected (dense)* layer that uses *softmax* function as its activation to make predictions.

5.1 Cross Entropy Loss

The cross-entropy loss tells us how sure the neural network is about each prediction. The formula for calculating the cross-entropy loss is

$$L = -\ln(p_c)$$

- c: correct class
- p_c: predicted probability of the correct class
- $\ln$: natural log
- L: loss

We will try to minimize this loss while training the model.