Final Project: CNN from scratch

Phan Thanh Binh

I. Introduction

Convolutional Neural Networks (CNNs) is a special type of deep learning network that designed for extracting features from grid-like structure such as images. The concept probably originated from the fusion of image processing techniques (e.g. applying filters to images) and deep neural networks, where optimal weights are learnt for specific tasks. Image processing techniques, specifically applying filters to extract meaningful features from images, faces challenges of choosing the right kernel size and parameters. Deep neural networks, if utilizing traditionally for images, would require a tremendous amount of weights and fail to account for the local spatial relationships between neighboring pixels.

The goal of this project is to implement this revolutionary concept from scratch, which helps strengthen my understanding of CNN. The term *from scratch* literally means no external frameworks or libraries are allowed, just builtin features of a programming languages. I chose Python for its simplicity and readability. Its syntax closely resembles pseudocode, making the implementation easier to understand and follow. Some basic components will be explored: convolution, maxpool, flatten, dense, ReLU, softmax; then the model will be applied to a image classification task.

II. Implementation

A. General Idea

My implementation are based off these ideas:

- Every component implements the *Module* interface, which has 3 core methods: forward, backward and update.
- A model itself is also a *Module*, specifically, one that contains a sequence of other modules.
- In the model's forward method, each module receives the output from the previous module, applies its own forward operation, and passes the result to the next module. The input to the first module is the input data.
- Similarly, in the backward method, each module receives the gradient from the next module, performs its own backward operation, and passes the gradient to the previous module. The first gradient that the model receives is given by the loss function.
- The update method simply calls the update method of each module, allowing them to adjust their internal parameters accordingly.
- Due to my not so optimal implementation, the components actually implements 3 different subclasses of *Module*. This will be detailed in later sections.

B. Dense

This section is about the *Dense* subclass of *Module*, which the forward and backward method both receives and return an array of floats (Figure 1). The *Array* class here simply is my wrapper for the Python list which support elements-wise operations.

```
class Module:
    def __call__(self, xs: Array[float]) -> Array[float]: ...
    def back(self, grads: Array[float]) -> Array[float]: ...
    def update(self): return None
```

Fig. 1: Dense Module Interface

1) Linear Layer: The linear layer is the most important module of this section since it is the only module that has weights, which allow the model the be trained and updated. In my mind, the linear layer is just a container of many linear nodes (or neurons), and the input and output of each node are based on this really simple Figure 2 that we all learn in the machine learning course about neural networks. With the input $x_1, x_2, ..., x_m$ (array of floats), the node multiply each of the input with its corresponding weights, take the sum, add its bias to give the output (a float). Notice the activation function for each node is not mentioned since it will also be a module that applied the activation function to all the outputs of the linear layer.

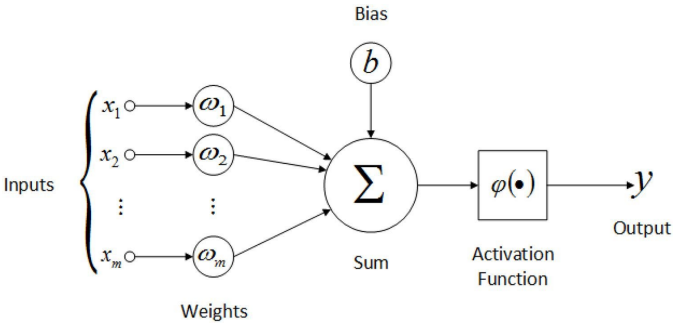

Fig. 2: Neural Network Linear Neuron

My implementation for the node is shown in Figure 3. In the forward method, as stated, the output will be the dot product of the input vector with the node's weights vector plus the bias. The input vector is remembered to compute the gradients later in the backward method. The backward method receive a float because the whole linear layer receive a float array of gradients and each gradient is given to each node accordingly. If we takes the partial derivatives for the gradients:

- the bias gradient is the same as the received gradient

- the weights gradients is computed by multiply grad with the remembered input.
- the gradients that the node passed onto the input is computed by multiply grad with its weights.

I stored the gradients in a list for the ability to updated by batch of inputs.

```
class _Node:
    def __init__(self, n_in: int):
        self.ws = Array(rng.rand() for _ in range(n_in))
        self.b = 0.0

        self.ws_grads: list[Array[float]] = []
        self.b_grads : list[float]= []

    def __call__(self, xs: Array[float]) -> float:
        self.xs = xs
        return self.ws@xs + self.b

    def back(self, grad: float) -> Array[float]:
        self.b_grads.append(grad)
        self.ws_grads.append(grad * self.xs)
        return grad * self.ws

    def update(self):
        rate = -1/len(self.b_grads)
        self.b = self.b + rate * sum(self.b_grads)
        self.ws: Array[float] = self.ws + rate * sum(self.ws_grads)

        self.b_grads = []
        self.ws_grads = []
```

Fig. 3: Linear Node Implementation

Onto the linear layer, the implementation can be seen in Figure 4. The configuration for the layer has 2 parameters: n_{in} and n_{out} specified the shape of layer's input and output respectively. The forward method simply returns an array of the nodes' outputs. The backward method is done similarly, but the gradients that the layer passed onto the previous module is the sum of the backward gradients from its nodes.

```
class Layer(Module):
    def __init__(self, n_in: int, n_out: int):
        self.nodes = [_Node(n_in) for _ in range(n_out)]

    def __call__(self, xs: Array[float]) -> Array[float]:
        return Array(node(xs) for node in self.nodes)

    def back(self, grads: Array[float]) -> Array[float]:
        return Array(
            node.back(grad)
            for node, grad in zip(self.nodes, grads)
        ).sum()

    def update(self):
        for n in self.nodes: n.update()
```

Fig. 4: Linear Layer Implementation

2) Activations: As mentioned earlier, the activation function is also implemented as a module. These modules does not have weights therefore not trainable, but help transform the output of the previous layer in forward and the gradient of the next layer in backward. I implements 2 most popular activations, *ReLU* and *Sigmoid*, as show in Figure 5.

```
class Sigmoid(Module):
    def __call__(self, xs: Array[float]) -> Array[float]:
        self.out = Array(sigmoid(x) for x in xs)
        return self.out

    def back(self, grads: Array[float]) -> Array[float]:
        return grads * Array(o*(1-o) for o in self.out)

class ReLU(Module):
    def __call__(self, xs: Array[float]) -> Array[float]:
        self.out = Array(max(0, x) for x in xs)
        return self.out

    def back(self, grads: Array[float]) -> Array[float]:
        return grads * Array(float(o > 0) for o in self.out)
```

Fig. 5: Dense Activations

3) Notes: With these first implementations, the idea to implement every module is shown very clear:

- Each module will perform a transformation to the input (a function), this will be applied in the forward method
- During backward, the module will transform the gradients given by its next module. This transformation will be calculated based on the partial derivatives of its transformation function.

C. Convolution

This section is about the *Convolution* subclass of *Module*, which the forward and backward method both receives and return an array of matrices (Figure 6). My logic is that, matrices in the *Convolution* subclass is somewhat similar to floats in the *Dense* subclass.

```
class Module:
    def __call__(self, xs: Array[Matrix]) -> Array[Matrix]: ...
    def back(self, grads: Array[Matrix]) -> Array[Matrix]: ...
    def update(self): return None
```

Fig. 6: Convolution Module

1) Linear Layer: The linear layer for convolution will be implemented exactly the same as in dense. The only modification is that, with dense, the weights and the inputs are multiplied together, but with convolution, the inputs will perform convolution with the weights. This also leads to some changes in backward. The linear node implementation is shown in Figure 7. Compared to dense linear node, it has 2 additional configurations: kernel and padding size, which are all foundational for the convolution operation. In the forward method, the inputs are padded, remembered, then get convolution-ed with the node's weights. The backward is quite tricky, but after some hours of carefully computing the partial derivatives, i come up with the backward solution:

- the bias gradients is the sum of all entries of the backward gradient matrix
- the weights gradients is computed by perform the convolution of the remembered inputs with the gradient matrix
- the gradients that is passed onto the previous module is computed by first flip the gradient matrix, the nodes weights is padded and performs convolution with the

flipped gradient matrix, start from the last entry of the gradient matrix with the first entry of the weights.

```
class _Node:
    def __init__(self, n_in: int, ksize: int, pad: int = 1):
        self.ws = Array(Matrix.fill(None, ksize, ksize) for _ in range(n_in))
        self.b = 0.0

        self.ws_grads: list[Array[Matrix]] = []
        self.b_grads : list[float]= []

        self.pad = pad

    def __call__(self, xs: Array[Matrix]) -> Matrix:
        self.xs = Array(m.pad(self.pad, self.pad) for m in xs)
        out = Array(x.conv(w) for x, w in zip(self.xs, self.ws)).sum()
        return out

    def back(self, grad: Matrix) -> Array[Matrix]:
        self.b_grads.append(sum(grad))
        self.ws_grads.append(Array(m.conv(grad) for m in self.xs))

        grad = grad.flip()
        return Array(
            w.pad(grad.nrow - 1, grad.ncol - 1).conv(grad)[
                self.pad : -self.pad, self.pad : -self.pad
            ]
            for w in self.ws
        )

    def update(self):
        rate = -1/len(self.b_grads)
        self.b = self.b + rate * sum(self.b_grads)
        self.ws: Array[Matrix] = self.ws + rate * sum(self.ws_grads)

        self.b_grads = []
        self.ws_grads = []
```

Fig. 7: Convolution Node Implementation

2) Activations: The activations are literally the same but each activations is applied on all entries of the matrices. I only implemented *ReLU*

3) Pooling: I implemented *MaxPool* since it is the most popular used. This convolution module is quite special this the shape of the input and output matrix decrase by half. When forward, the key points is to remembered the entry indices what was chosen to be the max in the neighbor hood. The MaxPool Node is shown in Figure 8

```
class _MP:
    def __init__(self, ksize: int = 2):
        self.k = ksize

    def __call__(self, x: Matrix) -> Matrix:
        assert x.nrow % self.k == 0 and x.ncol % self.k == 0
        rs = range(0, x.ncol, self.k)
        cs = range(0, x.nrow, self.k)

        data: list[float] = []
        rcs: list[tuple[int, int]] = []
        for r in rs:
            for c in cs:
                v, (mr, mc) = x[r:r+self.k, c:c+self.k].max()
                data.append(v)
                rcs.append((mr+r, mc+c))

        self.shape = x.shape
        self.rcs = rcs
        self.out = Matrix(data).on(len(rs), len(cs))
        return self.out

    def back(self, grad: Matrix):
        assert grad.shape == self.out.shape
        out = Matrix.fill(0, *self.shape)
        for rc, v in zip(self.rcs, grad): out[rc] = v
        return out
```

Fig. 8: MaxPool Node implementation

D. Other

This section I implemented some special modules that cannot be classified in to *Dense* or *Convolution* such as the *Flatten* module and the *MCE* loss (Figure 9

```
class MCE:
    def __call__(self, preds: Array[float], label: int) -> float:
        assert label in range(len(preds))
        self.label = label
        preds = Array(exp(p) for p in preds)
        self.preds: Array[float] = preds / preds.sum()
        return -log(self.preds[self.label])

    def back(self) -> Array[float]:
        grad = Array(self.preds)
        grad[self.label] = grad[self.label]-1
        return grad


class Flatten:
    def __call__(self, xs: Array[Matrix]) -> Array[float]:
        self.shape = xs[0].shape
        out = Array(d for x in xs for d in x)
        self.n_out = len(out)
        return out

    def back(self, grads: Array[float]) -> Array[Matrix]:
        assert len(grads) == self.n_out
        k = self.shape[0] * self.shape[1]
        loop = range(0, self.n_out, k)
        out = Array(
            Matrix(grads[i+j] for j in range(k)).on(*self.shape)
            for i in loop
        )
        return out
```

Fig. 9: Misc Modules

III. Experiments

A. Dataset

For the experiments of the model, I have chosen Optical Recognition of Handwritten Digits dataset from UCI Repository. It is a data for digit recognition problem where given an image, the task would be to classify that image into the exact number it represents. The data has a total of 5620 instances, each of which is an 8x8 image with pixel values ranging from 0 to 16. There are total of 10 classes where the labels are the numbers from 0-9.

B. Model

TABLE I: Neural Network Architecture

Layer	Type	Parameters
1	Convolutional	Input: 1, Output: 2, Kernel: 3
2	ReLU Activation	-
3	Convolutional	Input: 2, Output: 4, Kernel: 3
4	ReLU Activation	-
5	Max Pooling	Pool size: 4
6	Flatten	-
7	Dense	Input: 64, Output: 32
8	ReLU Activation	-
9	Dense	Input: 32, Output: 10

Table I details my CNN model's architecture. I build a simple and traditional CNN model with two Convolutional layers, a Max Pooling, and then two Linear layers to map to the final vector with 10 elements corresponding to 10 classes in the dataset.

Apart from the model building, I also need to define some hyperparameters for the final setup of our training. For this experiment, I use a learning rate of 0.1, a batch size of 128 and I will train it for 10 epochs

C. Results

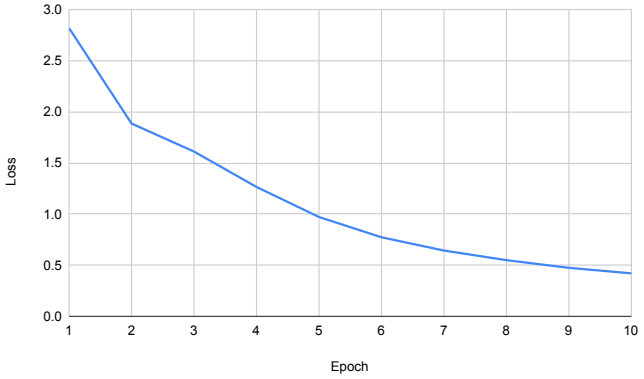

Fig. 10: Loss value per epoch

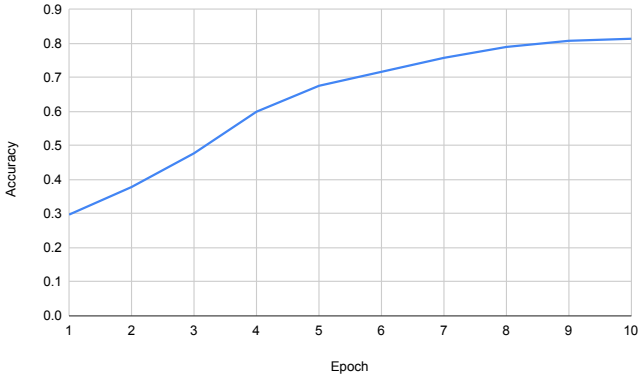

Fig. 11: Accuracy per epoch

The results are shown in Figure 10 and Figure 11. It can be seen that the loss gradually decreases each epoch while the accuracy increases after each update which is the expected trend in this training process. These results show that our model has successfully learned and updated through each epoch. Additionally, we can see that the loss value has not yet completely converge meaning as we continue the training for more epochs, the model can still continue to learn and update.