Deep Learing - Project: CNN from Scratch

Nguyen Dang Minh - M23.ICT.008

June 16, 2024

1 Introduction

Convolutional Neural Networks (CNNs) are a class of deep learning algorithms that are particularly powerful for analyzing visual data. They are widely used in image and video recognition, medical image analysis, and other computer vision tasks.

For this project, we aim to implement a CNN from scratch without utilizing any external libraries for the computational aspects. Specifically, we will develop the necessary components, such as convolutional layers, pooling layers, and fully connected layers, using only fundamental programming constructs. Subsequently, we will evaluate the implemented CNN's performance on diverse datasets and architectures to analyzing the results.

2 Architecture

A typical CNN architecture consists of several types of layers:

2.1 Convolutional Layer

The convolutional layer is the core of a CNN. It consists of a set of learnable kernels that slide over the input data to produce feature maps.

2.2 Activation Function

After the convolution, an activation function is applied. The most commonly used activation function is ReLU, defined as:

$$\text{ReLU}(x) = \max(0, x)$$

2.3 Pooling Layer

Pooling layers reduce the dimensions of the feature maps to reduce complexity, which improve the training time and predict.

2.4 Fully Connected Layer

We processed in a traditional neural network for simple classfication for the number of unique classes that dataset has.

It works like labwork 5 custom neural network we have done.

3 Implement

Our CNN implementation is based on the TensorFlow Keras style for defining networks and layers. To create any model, we need to define a model, then add subsequent layers to the model, validate the model, and finally train it.

3.1 Layers

At the time of writing this report, there are 5 available layers that can be added to the model:

3.1.1 ConvLayer

This layer creates a convolution kernel that convolves with the layer input over a single spatial dimension to produce a 3D array of outputs.

```
src.Layer.ConvLayer(
    filter_size,
    depth,
    num_filters,
    stride=1,
    padding=0
)
```

Listing 1: ConvLayer Definition

Arguments:

- `filter_size`: int, specifies the size of the convolution window.
- `depth`: int, specifies the number of channels in the input image.
- `num_filters`: int, specifies the number of filters in the convolution.
- `stride`: int, specifies the stride length of the convolution.
- `padding`: int, specifies the number of zero pixels to pad the border.

Input shape:

A 3D list with shape: (channels, height, width)

Output shape:

A 3D list with shape: (filters, new_height, new_width)

Example:

```
y = ConvLayer(10, 3, 16, stride=2)
outputs = y.forward(x)
```

Listing 2: ConvLayer Example

3.1.2 PoolingLayer

Downsamples the input along its spatial dimensions (height and width) by taking the maximum/average value over an input window for each channel of the input. The window is shifted by strides along each dimension.

```
src.Layer.PoolingLayer(
    size=2,
    stride=1,
    mode=['max','avg'])
```

Listing 3: PoolingLayer Definition

Arguments:

- `size`: int, specifies the factors by which to downscale into.
- `stride`: int, specifies the stride length of the downscale.
- `mode`: specifies the pooling operation for spatial data. 'max' using the maximum value over an input window, 'avg' using the average value over an input window.

Input shape:
A 3D list with shape: (channels, height, width)
Output shape:
A 3D list with shape: (channels, smaller_height, smaller_width)

3.1.3 Flatten

Flattens the input.

```
src.Layer.Flatten()
```

Listing 4: Flatten Definition

Input shape:
A 3D list with shape: (channels, height, width)
Output shape:
A 1D list with shape: (channels*height*width)
Example:

```
X = [[[4.25 4.25][4.25 3.5 ]][[4.25 4.25][4.25 3.5 ]][[4.25
    4.25][4.25 3.5 ]]]
y = Layer.Flatten()
outputs = y.forward(x)
>>> outputs = [4.25 4.25 4.25 3.5  4.25 4.25 4.25 3.5  4.25 4.25
    4.25 3.5 ]
```

Listing 5: Flatten Example

3.1.4 Dense

Densely-connected Neural Net layer.

Dense implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer.

```
src.Layer.Dense(input_dim, output_dim, activation='relu')
```

Listing 6: Dense Definition

Arguments:

- `input_dim`: int , specifies dimensionality of the input space.
- `output_dim,`:int , specifies dimensionality of the output space.
- `activation`:. specifies the Activation function for data. Available as : 'relu' : ReLU 'leakyrelu' : leaky ReLU 'sigmoid' : Sigmoid 'tanh' : tanh

Input shape:

A 1D list with length = input_dim

Output shape:

A 1D list with length = output_dim

Example:

```
X = [0, 0]
v = Dense(input_dim=2, output_dim=4, activation='relu')
output = v.forward(x)
>>> outputs = [1 0 1 0]
```

Listing 7: Dense Example

3.1.5 MultiClassDense

This is the same Dense Layer as above. But using Softmax as activation Function. This layer only for using as last layer as a Multi Class Classification.

3.2 Model

My Network class functions similarly to the Sequential class used in TensorFlow Keras. After defining the model, you can add layers sequentially using the .add(Layer) method. Once all layers are added, you can validate the model with a sample image using the .summary(input) method. If the validation is successful, you can begin training the model using the Train() method.

```
train(self, X, y, epochs, learning_rate, loss_fn,
    loss_fn_derivative, log_file="default_log.txt", val_X=None,
    val_y=None):
```

Listing 8: Train Definition

- `X`: 4D list, the training data.
- `y`: 2D or 3D list, the target values.
- `epochs`: int, the number of epochs to train the model.
- `learning_rate`: float, the learning rate for the training.
- `loss_fn`: function, the loss function used to evaluate the model. We include two types of loss functions in the `act_func.py`:
  - `bce_loss`: for binary classification.
  - `categorical_crossentropy`: for multi-class classification.
- `loss_fn_derivative`: function, the derivative of the loss function used for backpropagation. Use the corresponding derivative with your chosen loss.
- `log_file`: str, optional, the file path to log the training process. Default is "default_log.txt".
- `val_X`: 4D list, optional, the validation data. Default is None.
- `val_y`: 2D or 3D list, optional, the validation target values. Default is None.

```
network = Network()
network.add(ConvLayer(4,1,16))
network.add(ConvLayer(6,16,32))
network.add(PoolingLayer(size=2))
network.add(ConvLayer(8,32,64))
network.add(Flatten())
network.add(Dense(input_dim=576, output_dim=320, activation='relu'))
network.add(Dense(input_dim=320, output_dim=128, activation='relu'))
network.add(MultiClassDense(128,num_classes=10))

network.summary(train_images[0])
network.train(train_images, train_labels, epochs=100, learning_rate=0.0003, loss_fn=categorical_crossentropy, loss_fn_derivative=categorical_crossentropy_derivative,val_X=val_images,val_y=val_labels,log_file="CNN_Simple_log.txt")
```

Listing 9: Example

We also add `save_model(filepath)` and `load_model(filepath)` for saving the model architecture and weights to a file and loading them back, respectively. These functions help in preserving the model state and enable resuming training or making predictions without redefining the model. I use pickle here for quick save. I swear this is the only package in the src are used in this model :).

4 Results

We test with 4 different dataset. 2 only using FullyConnectedLayer, and 2 using CNN. we also test with binary classfication and multi classfication.

4.1 Binary Fully Connected NN

The dataset we use for this category is XOR of 3 binary variables. We get random 15 sets. The network comprises 3 dense layers with sizes 3, 4, and 1 respectively. The output will be XOR of 3 binary variables.

Layer name	Input shape	Output shape
Dense	(3,)	(4,)
Dense	(4,)	(3,)
Dense	(3,)	(1,)

After training with epochs=4000 and learning_rate=0.01, we get:

- Loss: 0.0022
- Accuracy: 1.0000

Example predictions:

- Input: [1, 1, 0], True: [0], Predicted: [0.00028948259382792637]
- Input: [0, 1, 0], True: [1], Predicted: [0.999848693992993]
- Input: [0, 1, 1], True: [0], Predicted: [0.0002865771116965438]

This indicates the data have been learned.

4.2 Multi Classes Fully Connected NN

The dataset we use for this category is a synthetic dataset with 4 binary variables. We generate 25 random samples. The network comprises 3 layers:

- Dense layer with 4 inputs and 4 outputs, using ReLU activation.
- Dense layer with 4 inputs and 3 outputs, using ReLU activation.
- MultiClassDense layer with 3 inputs and 3 outputs.

The output will be one of three classes, encoded in one-hot format.

Layer name	Input shape	Output shape
Dense	(4,)	(4,)
Dense	(4,)	(3,)
MultiClassDense	(3,)	(3,)

After training with epochs=8000 and learning_rate=0.2, we evaluate the network.
Example predictions:

- Input: [1, 0, 0, 0],True [0, 0, 1], Predicted: [3.67157840922965e-11, 1.9779628520267672e-19, 0.9999999999632843]
- Input: [1, 1, 1, 0],True [0, 1, 0], Predicted: [0.003347154077492902, 0.9963500473830836, 0.0003027985394234498]
- Input: [0, 1, 1, 1],True [0, 1, 0], Predicted: [0.003347154077492902, 0.9963500473830836, 0.0003027985394234498]

This indicates the data have been learned.

4.3 Convolutional Neural Network for MNIST Classification

First, we load the MNIST dataset with 30 samples per class for training and validation. We use a special dataset loader in `Dataset.py` to load the MNIST dataset in list format. This loader use TF and numpy just because the loading image into list part, no calculation. The dataset is divided into training, testing, and validation sets with a ratio of 0.7, 0.2, and 0.1 respectively.

X will be a 4D list that has the shape $(\text{num\_sample}, \text{channel} = 1, \text{height} = 28, \text{width} = 28)$, as MNIST images are grayscale.

Y will be a 2D list that has the shape $(\text{num\_sample}, \text{classes} = 10)$, as there are 10 classes in one-hot encoding.

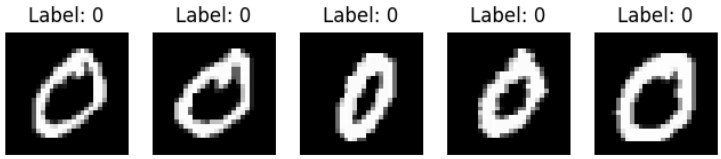

Figure 1: Sample Training image in MNIST

```
train_images, train_labels, test_images, test_labels , val_images,
val_labels = load_mnist_dataset(num_samples_per_class=30)
```

We verify the shape and label of the first training image:

```
First training image shape:  (300, 1, 28, 28) First training image
label:  [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Next, we initialize our network and add the following layers:

- `ConvLayer(6, 1, 16)`
- `PoolingLayer()`

- `Flatten()`
- `Dense(input_dim=1936, output_dim=160, activation='relu')`
- `MultiClassDense(160, num_classes=10)`

`network = Network()`

```
network.add(ConvLayer(6, 1, 16))
network.add(PoolingLayer())
network.add(Flatten())
network.add(Dense(input_dim=1936, output_dim=160, activation='relu'
    ))
network.add(MultiClassDense(160, num_classes=10))
```

Listing 10: Example

We summarize the network:

Layer name	Input shape	Output shape
ConvLayer	(1, 28, 28)	(16, 23, 23)
PoolingLayer	(16, 23, 23)	(16, 11, 11)
Flatten	(16, 11, 11)	(1936,)
Dense	(1936,)	(160,)
MultiClassDense	(160,)	(10,)

We train the network with 100 epochs, a learning rate of 0.0003, and validate it using the validation set. We use Categorical Cross-entropy for 10 classes classfication.

```
network.train(train\_images, train\_labels, epochs=100, learning\
    _rate=0.0003, loss\_fn=categorical\_crossentropy, loss\_fn\
    _derivative=categorical\_crossentropy\_derivative, val\_X=val\
    _images, val\_y=val\_labels, log\_file="CNN\_Simple\_log.txt")}
```

Listing 11: Example

Based on the log of the training process: Total Training Time: 5:09:17

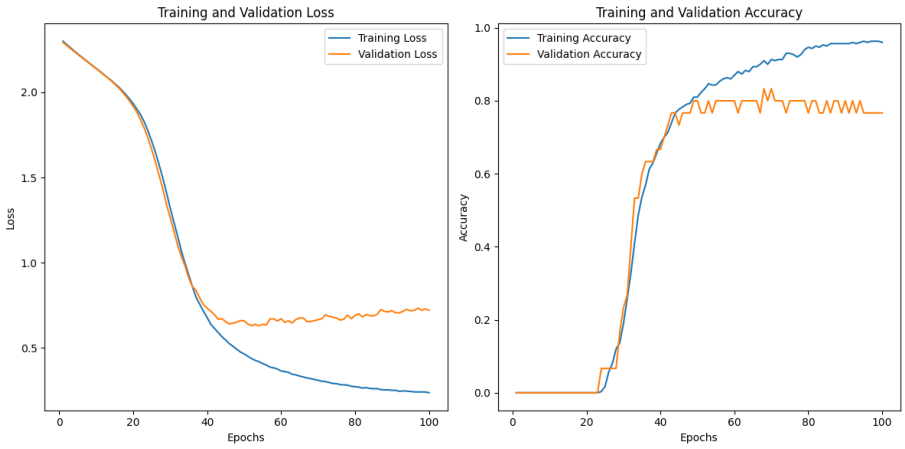

Figure 2: Training and Validation Loss and Accuracy over Epochs

After training, we save the model:
`network.save_model("CNN_Simple.pkl")`
Finally, we evaluate the network on the test set:

```
out, test_loss, test_accuracy = network.evaluate( test_images,
    test_labels,categorical_crossentropy)
print(f"Test Loss: {test_loss}, Test Accuracy: {test_accuracy}")
```

Listing 12: Example

This part uses external libraries for generate Classification Report and Confusion Matrix.

precision	recall	f1-score	support	
0	0.86	1.00	0.92	6
1	1.00	1.00	1.00	6
2	0.50	0.83	0.62	6
3	1.00	1.00	1.00	6
4	0.71	0.83	0.77	6
5	0.83	0.83	0.83	6
6	1.00	0.33	0.50	6
7	1.00	0.83	0.91	6
8	1.00	0.83	0.91	6
9	0.67	0.67	0.67	6
accuracy	0.82	60		
macro avg	0.86	0.82	0.81	60
weighted avg	0.86	0.82	0.81	60

Table 1: Classification Report of Simple CNN

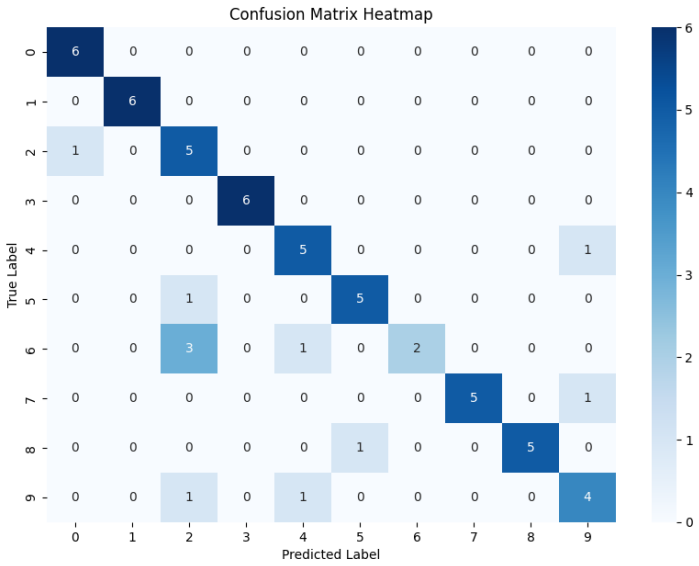

Figure 3: Confusion Matrix Heatmap

With an accuracy of 0.82 achieved using only 100 epochs and a simple 5-layer network, we believe there is potential for even better results by increasing the model's complexity. To explore this, we included a more complex model in `CNN_Test.py`. However, due to suboptimal code for CPU execution, the complex model frequently crashed on our laptop. Nevertheless, based on the log of the epochs that did run, we observed:

```
03:14:17  :Epoch 1/20, Loss: 2.4031, Accuracy: 0.0
03:21:11  :Epoch 2/20, Loss: 2.0304, Accuracy: 0.06
03:28:02  :Epoch 3/20, Loss: 2.0340, Accuracy: 0.27
03:34:56  :Epoch 4/20, Loss: 3.6728, Accuracy: 0.355
```

From this, we can see that the complex model converges much faster.

4.4 Convolutional Neural Network for Cat/Dog Classification

First, we load the Cat/Dog dataset with 70 samples per class for training, testing, and validation. The dataset can be found at *https://www.kaggle.com/datasets/tongpython/cat-and-dog*. The dataset is divided into training, testing, and validation sets with a ratio of 0.7, 0.2, and 0.1 respectively.

X will be a 4D list that has the shape $(\text{num\_sample}, \text{channel} = 3, \text{height} = 50, \text{width} = 50)$, as Cat/Dog images are big and RGB. We also reshape the images to 50×50 for faster computation.

Y will be a 2D list that has the shape (num_sample, classes = 1), with 0 and 1 representing cats and dogs, respectively.

Figure 4: Sample Training Image in Cat/Dog Dataset

We verify the shape and label of the first training image:
`First training image shape: (140, 3, 50, 50) First training image label: [0]`
Next, we initialize our network and add the following layers:

- `ConvLayer((10 ,3,16,stride=2)`
- `PoolingLayer()`
- `Flatten()`
- `Dense(input_dim=1600, output_dim=160, activation='relu')`
- `Dense(input_dim=160, output_dim=1, activation='sigmoid')`

```
network = Network()
network.add(ConvLayer(10,3,16,stride=2))
network.add(PoolingLayer())
network.add(Flatten())
network.add(Dense(input_dim=1600, output_dim=160, activation='relu'
    ))
network.add(Dense(input_dim=160, output_dim=1, activation='sigmoid'
    ))
```

Listing 13: Example

We summarize the network:

Layer name	Input shape	Output shape
ConvLayer	(3, 50, 50)	(16, 21, 21)
PoolingLayer	(16, 21, 21)	(16, 10, 10)
Flatten	(16, 10, 10)	(1600,)
Dense	(1600,)	(160,)
Dense	(160,)	(1,)

We train the network with 50 epochs, a learning rate of 0.0003, and validate it using the validation set. We use Binary Cross-entropy for Binary classfication.

```
network.train(train_images, train_labels, epochs=50, learning_rate
    =0.0003, loss_fn=bce_loss, loss_fn_derivative=
    bce_loss_derivative,val_X=val_images,val_y=val_labels,log_file=
    "CNN_Binary_log.txt")
}
```

Listing 14: Example

Based on the log of the training process: Total Training Time: 8:21:01

Figure 5: Training and Validation Loss and Accuracy over Epochs

After training, we save the model:
`network.save_model("CNN_Binary.pkl")`
Finally, we evaluate the network on the test set:

This part uses external libraries for generate Classification Report and Confusion Matrix.

precision	recall	f1-score	support	
0	0.50	0.36	0.42	14
1	0.50	0.64	0.56	14
accuracy	0.50	28		
macro avg	0.50	0.50	0.49	28
weighted avg	0.50	0.50	0.49	28

Table 2: Classification Report of Binary CNN

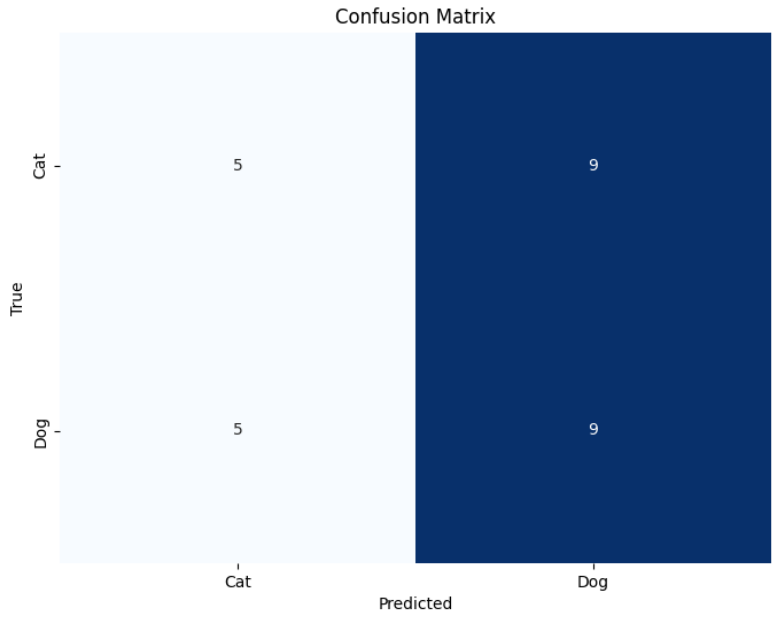

Figure 6: Confusion Matrix Heatmap

As observed, the model achieves low accuracy primarily due to its tendency to misclassify cat images as dog images. This issue can be attributed to several factors: the model's simplicity, insufficient epoch training for a CNN designed to handle RGB images, and the lack of convolutional layers. The need for a more complex architecture with additional convolutional layers becomes apparent to effectively capture and differentiate the intricate features inherent in RGB images like those of cats and dogs.

5 Conclusion

In conclusion, while we were able to successfully train a CNN model, its performance is currently constrained by limited computational resources relying solely on CPU. To achieve better results, future improvements should focus on adding more layers to the network, optimizing activation functions and utilities, and use GPU acceleration for faster computations. These enhancements are crucial for handling more complex tasks and datasets effectively