Electrical Engineering Department
Sharif University of Technology

Title:

Project

Neural Networks

Author:
Mohammad Hossein Najafi (97103938)

Supervisor:
Dr. S. Bagheri

Summer 2024

Part One

Model

We use the train directory for training, and for evaluating the training process, we show the loss for each epoch. We make 10 epochs. We use the test directory for testing, and on the test directory, we got 58 percent accuracy.
Also, fps means frames per second. For finding the model's FPS, we run it on the test dataset, which contains 6588 images, and we got 59.21 FPS.

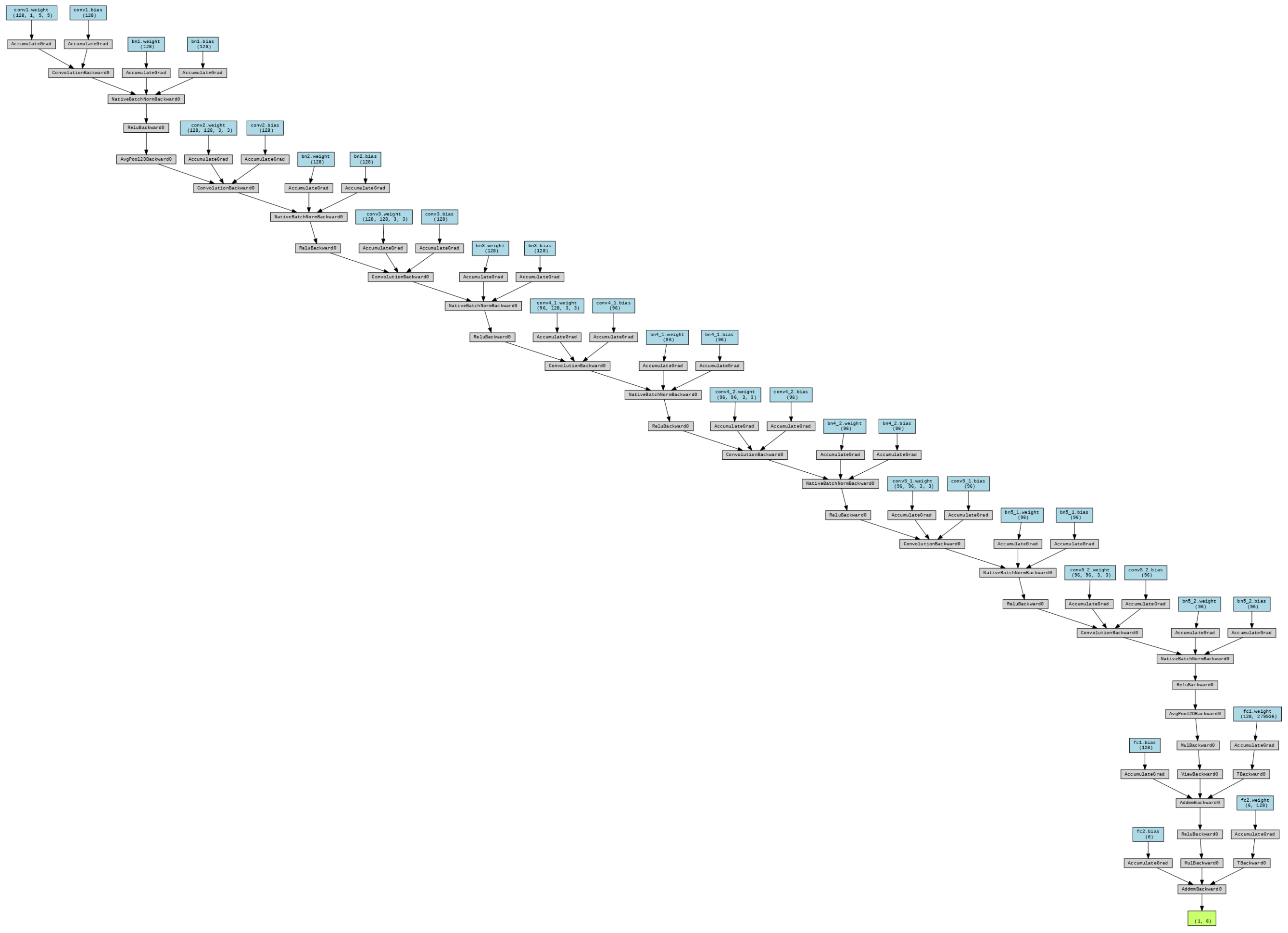

Figure 1: Model

Questions

1

The image resolution is 800x947. For improved learning, we transform and resize the images to 256x256.
Yes, it is possible to change the number of channels of a grayscale image. A grayscale image typically has 1 channel, but it can be converted to 3 channels by replicating the single channel across the three RGB channels.
The number of parameters in the first convolutional layer will triple. This is because each filter will now have three times as many weights (one for each channel).
However, the number of feature maps depends on the number of filters used in the convolutional layers, not the number of input channels.

for 1-channel or 3-channel: total feature map = 768

Layer	Output Size
Conv1	128 feature maps
BatchNorm1	128 feature maps
AvgPool1	128 feature maps
Conv2	128 feature maps
BatchNorm2	128 feature maps
Conv3	128 feature maps
BatchNorm3	128 feature maps
Conv4_1	96 feature maps
BatchNorm4_1	96 feature maps
Conv4_2	96 feature maps
BatchNorm4_2	96 feature maps
Conv5_1	96 feature maps
BatchNorm5_1	96 feature maps
Conv5_2	96 feature maps
BatchNorm5_2	96 feature maps
AvgPool2	96 feature maps
Dropout	96 feature maps
FC1	128 units
Dropout_FC1	128 units
FC2	6 units

Table 1: CNN Architecture

1-channel parameters:36492550

3-channel parameters:36498950

2

- Convolution Layer (Conv): The convolution layers are used to apply filters to the input data, extracting relevant features and creating feature maps. This is a fundamental component of

CNNs, as it allows the network to learn local patterns and representations from the input data.

- Batch Normalization Layer (BatchNorm): The batch normalization layers are used to improve the stability and performance of the network by normalizing the inputs to each layer. This helps to reduce the internal covariate shift, which can lead to faster convergence and better generalization.
- Average Pooling Layer (AvgPool): The average pooling layers are used to reduce the spatial dimensions of the feature maps, while preserving the most important information. This helps to reduce the computational complexity of the network and introduces a certain level of translation invariance.
- Dropout Layer (Dropout): The dropout layers are used to prevent overfitting by randomly disabling a proportion of the units in the network during training. This encourages the network to learn more robust and generalized features, improving its performance on unseen data.
- Fully Connected Layer (FC): The fully connected layers are used to transform the learned features into the final output of the network. These layers are responsible for making the final predictions or classifications based on the output of the previous layers.

3

In the implementation of the CNN model presented, the initial assignment of the weights is done during the construction of the model and the definition of its layers. Most pre-trained models use appropriate distributions for the initial weight initialization to improve the convergence of training. In this specific implementation, the initial weight initialization is done by PyTorch by default. In other words, in the ___init___() method, where the various convolutional layers, batch normalization, and all the fully connected layers are constructed, the initial weight assignment takes place. This initial assignment is based on standard methods such as Xavier initialization or He initialization, which are applied by default in PyTorch.

4

total parameters:36492550
The layers with the highest number of parameters are:
conv2.weight: 147,456 parameters
conv3.weight: 147,456 parameters
conv4_1.weight: 110,592 parameters
conv5_1.weight: 82,944 parameters
conv5_2.weight: 82,944 parameters
These convolutional layers have the most parameters due to their large kernel sizes and the number of input/output channels.

5

The loss function used is the CrossEntropyLoss from PyTorch, which is common for multi-class classification. While the loss function guides training, evaluating the model with additional metrics like accuracy is necessary to assess its performance comprehensively. The loss function and accuracy are related but distinct - the loss quantifies the prediction-label difference, while accuracy measures the proportion of correct predictions.

6

The optimizer used is the Adam optimizer from PyTorch, with a learning rate of 1e-2 and a weight decay of 1e-4.
The Adam optimizer is a popular choice for training neural network models as it combines the benefits of momentum and adaptive learning rates to efficiently update the model parameters during the optimization process.

Part Two

Chain Rule

The chain rule in backpropagation helps us determine how much each weight in a neural network affects the overall error. By breaking down the error step-by-step through each layer of the network, it allows us to calculate and adjust the weights to reduce the error, making the network learn and improve its predictions.

Suppose we have a simple neural network with an input layer, one neuron in the hidden layer, and one neuron in the output layer. Weights connect these neurons:

1. Feedforward

First, inputs pass through the network and the output is computed. Suppose we have inputs x_1 and x_2 connected to the hidden neuron with weights w_1 and w_2. The output of the hidden neuron h is:

$$h = f(w_1 \cdot x_1 + w_2 \cdot x_2)$$

where f is the activation function. Then, the output of the hidden neuron is connected to the output neuron with weight w_3, and the final output y is computed:

$$y = g(w_3 \cdot h)$$

where g is also an activation function.

2. Error Calculation

The error between the network output and the target value t is calculated. Suppose the loss function is:

$$E = \frac{1}{2}(y-t)^2$$

3. Backpropagation

The goal is to compute the gradients of the error for the weights to update the weights.
To compute the gradient $\frac{\partial E}{\partial w_3}$, we use the Chain Rule:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial w_3}$$

Here:

$$\frac{\partial E}{\partial y} = y - t$$

and

$$\frac{\partial y}{\partial w_3} = \frac{\partial g(w_3 \cdot h)}{\partial w_3} = g'(w_3 \cdot h) \cdot h$$

Therefore:

$$\frac{\partial E}{\partial w_3} = (y-t) \cdot g'(w_3 \cdot h) \cdot h$$

Now, to compute the gradients $\frac{\partial E}{\partial w_1}$ and $\frac{\partial E}{\partial w_2}$, we first need to compute the gradient with respect to h:

$$\frac{\partial E}{\partial h} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial h} = (y-t) \cdot g'(w_3 \cdot h) \cdot w_3$$

Then:

$$\frac{\partial h}{\partial w_1} = f'(w_1 \cdot x_1 + w_2 \cdot x_2) \cdot x_1$$

Therefore:

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial h} \cdot \frac{\partial h}{\partial w_1} = (y-t) \cdot g'(w_3 \cdot h) \cdot w_3 \cdot f'(w_1 \cdot x_1 + w_2 \cdot x_2) \cdot x_1$$

Similarly, we can compute the gradient with respect to w_2:

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial h} \cdot \frac{\partial h}{\partial w_2} = (y-t) \cdot g'(w_3 \cdot h) \cdot w_3 \cdot f'(w_1 \cdot x_1 + w_2 \cdot x_2) \cdot x_2$$

4. Updating Weights

The weights are updated using the computed gradients. Suppose the learning rate is η:

$$w_1 \leftarrow w_1 - \eta \frac{\partial E}{\partial w_1}$$

$$w_2 \leftarrow w_2 - \eta \frac{\partial E}{\partial w_2}$$

$$w_3 \leftarrow w_3 - \eta \frac{\partial E}{\partial w_3}$$

This process is repeated iteratively until the network error is minimized, allowing the network to gradually learn how to map inputs to the desired outputs. The Chain Rule helps transfer the error gradient step by step from the output layer to the previous layers, facilitating the optimization of weights.

Convolution

Calculating

At first, we complete the feature map and the output of the last layer. For calculating the convolution output and the sigmoid output, we write a Python code. The results are:
Convolved Output:

$$\begin{bmatrix} 0.25 & -0.16 & 0.27 & -0.34 \\ 0.11 & 0.31 & -0.30 & 0.01 \\ -0.22 & 0.61 & -0.09 & -0.28 \\ -0.33 & 0.48 & -0.27 & 0.19 \end{bmatrix}$$

Output after applying sigmoid:

$$\begin{bmatrix} 0.58 & 0.57 \\ 0.65 & 0.55 \end{bmatrix}$$

Backpropagation for Fully Connected Layer, Sigmoid, Max Pooling, and Convolution Layer

- Fully Connected Layer

  Forward Pass:

$$\begin{aligned} \text{Input: } & x \\ \text{Weights: } & W \\ \text{Bias: } & b \\ \text{Output: } & z = W \cdot x + b \end{aligned}$$

  Backward Pass:

$$\begin{aligned} \text{Gradient of loss w.r.t. output: } & \frac{\partial L}{\partial z} \\ \text{Gradients of loss w.r.t. weights: } & \frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot x^T \\ \text{Gradients of loss w.r.t. bias: } & \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \\ \text{Gradients of loss w.r.t. input: } & \frac{\partial L}{\partial x} = W^T \cdot \frac{\partial L}{\partial z} \end{aligned}$$

- Sigmoid Activation

  Forward Pass:

$$\begin{aligned} \text{Input: } & z \\ \text{Output: } & a = \sigma(z) = \frac{1}{1+e^{-z}} \end{aligned}$$

  Backward Pass:

$$\begin{aligned} \text{Gradient of loss w.r.t. output: } & \frac{\partial L}{\partial a} \\ \text{Gradient of loss w.r.t. input: } & \frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot a \cdot (1-a) \end{aligned}$$

- Max Pooling Layer

  Forward Pass:

$$\begin{aligned} \text{Input: } & x \quad \text{(assuming a 2x2 pooling window)} \\ \text{Output: } & y \quad \text{where } y_{i,j} = \max(x_{i:i+2,j:j+2}) \end{aligned}$$

Backward Pass:

$$\text{Gradient of loss w.r.t. output: } \frac{\partial L}{\partial y}$$

$$\text{Gradient of loss w.r.t. input: } \begin{cases} \text{For each } y_{i,j}, \text{ find the position of the max value in the corresponding} \\ \text{Propagate } \frac{\partial L}{\partial y_{i,j}} \text{ back to this position.} \\ \text{All other positions in the window get a gradient of 0.} \end{cases}$$

- Convolutional Layer

  Forward Pass:

$$\begin{aligned} \text{Input: } & x \\ \text{Filters: } & W \\ \text{Biases: } & b \\ \text{Output: } & y \quad \text{where } y_{i,j,k} = \sum_{m,n,c} W_{m,n,c,k} \cdot x_{i+m,j+n,c} + b_k \end{aligned}$$

  Backward Pass:

$$\begin{aligned} \text{Gradient of loss w.r.t. output: } & \frac{\partial L}{\partial y} \\ \text{Gradients of loss w.r.t. weights: } & \frac{\partial L}{\partial W_{m,n,c,k}} = \sum_{i,j} \frac{\partial L}{\partial y_{i,j,k}} \cdot x_{i+m,j+n,c} \\ \text{Gradients of loss w.r.t. bias: } & \frac{\partial L}{\partial b_k} = \sum_{i,j} \frac{\partial L}{\partial y_{i,j,k}} \\ \text{Gradients of loss w.r.t. input: } & \frac{\partial L}{\partial x_{i,j,c}} = \sum_{m,n,k} \frac{\partial L}{\partial y_{i-m,j-n,k}} \cdot W_{m,n,c,k} \end{aligned}$$

Result

Fully connected delta input:

$$\begin{bmatrix} 0.1495 \\ 0.28 \\ 0.2055 \\ -0.2755 \end{bmatrix}$$

Fully connected delta weights:

$$\begin{bmatrix} 0.14417249 & 0.14147853 & 0.16209922 & 0.13678985 \\ -0.0865035 & -0.08488712 & -0.09725953 & -0.08207391 \end{bmatrix}$$

Fully connected delta biases:

$$\begin{bmatrix} 0.25 & -0.15 \end{bmatrix}$$

Sigmoid delta:

$$\begin{bmatrix} 0.03649574 & 0.06878349 \\ 0.04684955 & -0.06826229 \end{bmatrix}$$

Max-pooling delta:

$$\begin{bmatrix} 0 & 0 & 0.06878349 & 0 \\ 0 & 0.03649574 & 0 & 0 \\ 0 & 0.04684955 & 0 & 0 \\ 0 & 0 & 0 & -0.06826229 \end{bmatrix}$$

Convolution delta input:

$$\begin{bmatrix} 0 & 0 & -0.00894185 & 0.01031752 & 0 \\ 0 & -0.00474445 & -0.02960522 & 0.04264576 & 0 \\ 0 & -0.02470327 & 0.02965479 & 0 & 0 \\ 0 & -0.02389327 & 0.02904672 & 0.0088741 & -0.01023934 \\ 0 & 0 & 0 & 0.03481377 & -0.04232262 \end{bmatrix}$$

Convolution delta kernel:

$$\begin{bmatrix} 0.04474314 & 0.09433226 \\ 0.01082693 & 0.06850172 \end{bmatrix}$$

Updated Convolution Kernel:

$$\begin{bmatrix} -0.15237157 & 0.10283387 \\ -0.51541346 & 0.58574914 \end{bmatrix}$$

Updated Fully Connected Layer Weights:

$$\begin{bmatrix} 0.53791375 & 0.74926073 & 0.87895039 & -1.06839493 \\ 0.06325175 & -0.45755644 & 0.27862977 & 0.21103696 \end{bmatrix}$$

Updated Fully Connected Layer Biases:

$$\begin{bmatrix} -0.125 & 0.075 \end{bmatrix}$$

Unfold (im2col)

The unfold operation transforms the input image into a series of columns, where each column represents a flattened receptive field of the input image.
Consider an input image of shape (C, H, W), where C is the number of channels, H is the height, and W is the width. Let the convolution kernel have a shape (K_h, K_w), where K_h is the height and K_w is the width of the kernel.
Steps:

1. Slide the kernel over the image with a specified stride and padding.
2. For each position of the kernel, extract the corresponding window from the image and flatten it into a column vector.
3. Stack these column vectors to form a matrix where each column corresponds to one of the receptive fields of the input.

Let S be the stride and P be the padding. The output dimensions after convolution (height H_{out} and width W_{out}) are calculated as:

$$H_{\text{out}} = \left\lfloor \frac{H + 2P - K_h}{S} \right\rfloor + 1$$

$$W_{\text{out}} = \left\lfloor \frac{W + 2P - K_w}{S} \right\rfloor + 1$$

The unfolding operation transforms the input tensor into a matrix where each column is a flattened receptive field of the input image. The dimensions of this matrix are:

$$\text{Unfolded Matrix Dimensions} = (C \times K_h \times K_w, H_{\text{out}} \times W_{\text{out}})$$

For example, for a 3×3 kernel sliding over a 5×5 image with no padding and a stride of 1, the unfold operation results in a matrix with 9 rows (flattened 3×3 kernel) and 9 columns (number of positions the kernel fits into).

Fold (col2im)

The fold operation is the reverse of unfold. It takes the matrix of column vectors (output of the unfold operation) and reconstructs the original input image or an appropriately transformed version of it.
Steps:

1. Reshape each column back into the original window shape.
2. Place these windows back into their original positions in the image, summing overlapping elements appropriately.

Advantages

- Efficiency: Matrix multiplication is highly optimized in modern libraries and hardware (GPUs and specialized hardware like TPUs).
- Parallelism: Unfolding allows for exploiting parallel processing capabilities effectively.
- Flexibility: This method can handle various convolution parameters like stride and padding uniformly.