MAIN PROBLEM

The main problem was the implementation of Convolutional Neural Network.

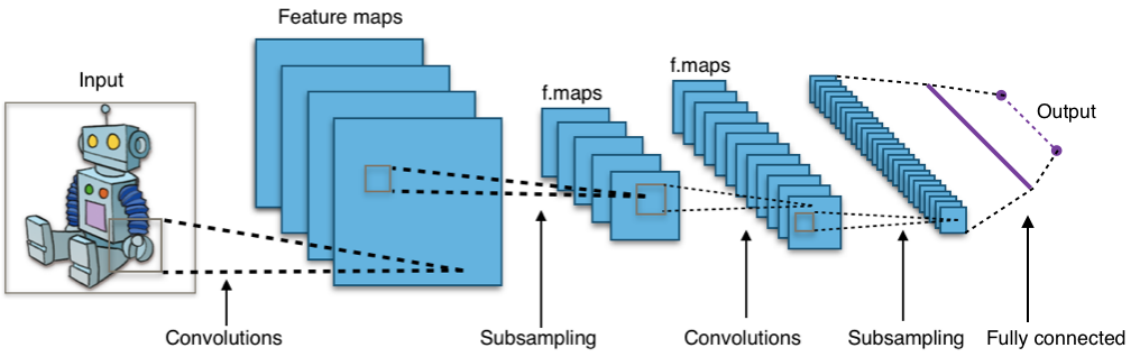


Figure 1 *Convolutional Neural Networks*

A convolutional neural network is a type of artificial neural network (ANN) that uses filters to extract features from the input data. These filters are applied with sliding window process and the outputs are feature maps. Thus, they can learn the spatial relationships between pixels and identify patterns and objects in input data.

The key features of CNNs:

- *Convolutional layers:* They extract features from the input data by applying a series of filters or kernels.
- *Pooling layers:* These layers reduce the dimensionality of the data by subsampling the output of the convolutional layers. This helps to reduce the computational cost of the network and to prevent overfitting.
- *Fully connected layers:* These layers are similar to the fully connected layers of a regular neural network. They combine the features extracted by the convolutional layers and pooling layers to produce a final output.

CNN is used for various tasks such as image recognition, object detection, image segmentation and video analysis.

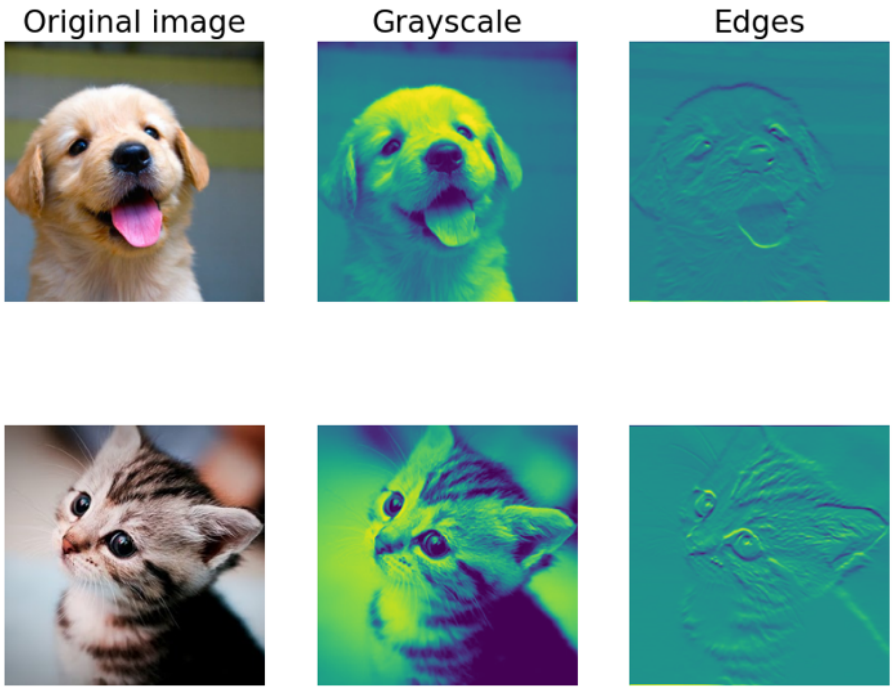
IMPLEMENTATION

1. Convolutional Layer

First, the forward pass function was implemented for a convolutional layer. This code implements the convolution operation using explicit nested loops over the batch size, filter dimension, and output height and width dimensions.

```
def forward(x, w, b):
    ...
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape

    # Calculate output dimensions
    stride, pad = conv_param['stride'], conv_param['pad']
    H_out = 1 + (H + 2 * pad - HH) // stride
    W_out = 1 + (W + 2 * pad - WW) // stride

    # zero-padding for input
    padded_x = torch.nn.functional.pad(x, (pad, pad, pad, pad))

    out = torch.zeros((N, F, H_out, W_out)).to(x)

    # convolution
    for i in range(N):
        for j in range(F):
        for h in range(H_out):
            for k in range(W_out):
                block = padded_x[i, :, h * stride:h * stride + HH,
                                       k * stride:k * stride + WW]
                out[i,j,h,k] = (block*w[j]).sum() + b[j]
    ...
```

The result error is 1.01e-9 which is less than 1e-7 so, it is acceptable.



As we can see in images above, textures and outlines of objects are more visible, and objects are now independent from backgrounds. These are important features for object detection and classification.

Then, the backward pass function was implemented for a convolutional layer. This code implements backpropagation with the convolution operation using explicit nested loops over the batch size, filter dimension, and output height and width dimensions.

```
def backward(dout, cache):
    ...
    x, w, b, conv_param = cache
    N, C, H, W = x.shape
    F, wC, HH, WW = w.shape

    # Calculate output dimensions
    stride, pad = conv_param['stride'], conv_param['pad']
    H_out = 1 + (H + 2 * pad - HH) // stride
    W_out = 1 + (W + 2 * pad - WW) // stride

    # zero-padding for input
    padded_x = torch.nn.functional.pad(x, (pad, pad, pad, pad))

    dx_pad = torch.zeros_like(padded_x).to(x.dtype).to(x.device)

    # Initialize gradients
    dx = torch.zeros_like(x).to(x.dtype).to(x.device)
    dw = torch.zeros_like(w).to(x.dtype).to(x.device)
    db = torch.zeros_like(b).to(x.dtype).to(x.device)

    for i in range(N):
      for j in range(F):
        db[j] += torch.sum(dout[i, j])
        for h in range(0, H_out):
          for k in range(0, W_out):
            block = padded_x[i, :, h * stride:h * stride + HH,
                                   k * stride:k * stride + WW]
            dw[j] += block * dout[i, j, h, k]
            dx_pad[i, :, h * stride:h * stride + HH,
                         k * stride:k * stride + WW] +=
                                                  w[j] * dout[i, j, h, k]
    dx = dx_pad[:, :, pad:pad + H, pad:pad + W]
    ...
```

The result errors of testing are dw= 1.34e-9, dw=1.00e-9 and db=7.01e-10 which are less than 1e-8 so, it is acceptable.

2. Max-Pooling

Max-pooling is a downsampling operation which is used to reduce the spatial dimensions of a feature map while keeping the most important information. It operates on each feature map independently and involves selecting the maximum value from a kernel and moving that kernel across the input feature map with a certain stride.

In this part of homework, the forward pass and backward pass for the max-pooling operation was implemented.

This code uses nested loops for max-pooling. The loop iterates over the input feature map (x) with a specified stride. For each pooling region, it applies max pooling along both height and width dimensions and assigns the maximum value to the corresponding location in the output tensor.

```
    N, C, H, W = x.shape
    pool_height, pool_width, stride = pool_param['pool_height'],
                                      pool_param['pool_width'],
                                      pool_param['stride']
    H_out = 1 + (H - pool_height) // stride
    W_out = 1 + (W - pool_width ) // stride

    out = torch.zeros((N, C, H_out, W_out)).to(x)

    for i in range(H_out):
      for j in range(W_out):
        # Apply max pooling along both width and height and
          assign the result
        block = x[:, :, i * stride:i * stride + pool_height,
                        j * stride:j * stride + pool_width]
        out[:, :, i, j] = torch.amax(block, dim=(2, 3))
```

The backward pass code implements the convolution operation using explicit nested loops over the batch size, filter dimension, and output height and width dimensions. In the loop, fins the indices of max value then calculates the gradients with these indices.

```
    x, pool_param = cache
    N, C, H, W = x.shape
    pool_height, pool_width, stride = pool_param['pool_height'],
pool_param['pool_width'], pool_param['stride']

    H_out = 1 + (H - pool_height) // stride
    W_out = 1 + (W - pool_width ) // stride

    dx = torch.zeros_like(x)

    # Iterate over each element in the input data
    for i in range(N):
      for j in range(C):
        for h in range(H_out):
          for k in range(W_out):
            block = x[i, j, h * stride:h * stride + pool_height, k * stride:k *
stride + pool_width]

            # Find the indices of the maximum value in the block
            max_h = torch.argmax(block).item()//pool_width,
            max_w = torch.argmax(block).item()%pool_width

            dx[i, j, h*stride + max_h, k * stride + max_w] += dout[i, j, h, k]
```

3. Fast Layers

The implementation of fast versions of convolution and max-pooling layers using PyTorch's built-in functions.

The results of comparisons are shown in below:

Testing FastConv

```
Testing FastConv.forward:
Naive: 4.573389s
Fast: 0.405314s
Fast CUDA: 0.000612s
Speedup: 11.283564x
Speedup CUDA: 7472.607324x
Difference:  2.1928544370986248e-16
Difference CUDA:  2.1928544370986248e-16

Testing FastConv.backward:
Naive: 5.216513s
Fast: 0.142354s
Fast CUDA: 0.000522s
Speedup: 36.644711x
Speedup CUDA: 9995.267702x
dx difference:  3.8774834488572664e-16
dw difference:  1.0684549953610693e-15
db difference:  1.6798889889341262e-16
dx diff CUDA:3.8774834488572664e-16
dw diff CUDA:1.0684549953610695e-15
db diff CUDA:1.6798889889341262e-16
```

Testing FastMaxPool

```
Testing FastMaxPool.forward:
Naive: 0.010955s
Fast: 0.000234s
Fast CUDA: 0.000156s
Speedup: 46.789206x
Speedup CUDA: 70.255352x
Difference:  0.0
Difference CUDA:  0.0

Testing FastMaxPool.backward:
Naive: 4.208053s
Fast: 0.000528s
Fast CUDA: 0.000310s
Speedup: 7968.330474x
Speedup CUDA: 13587.260970x
dx difference:  0.0
dx difference CUDA:  0.0
```

These implementations are more efficient than the naive ones implemented earlier.

4. Convolutional Sandwich Layers

The sandwich layers combine convolution, ReLU activation, and optional pooling operations. The relative errors for the gradients are on the order of e-9, which is well within an acceptable range and suggests that the implemented backward passes are accurate.

```
Testing Conv_ReLU:
dx error:  1.8037001458781077e-09
dw error:  1.2470995998634857e-09
db error:  1.1230402096612364e-09

Testing Conv_ReLU_Pool
dx error:  1.5915037060449427e-09
dw error:  1.8962680214651407e-09
db error:  5.05984212319748e-09
```

5. Three Layer Convolutional Network

A three-layer convolutional network with the following architecture:

1. Convolutional layer with ReLU activation and 2x2 max pooling
2. Linear layer with ReLU activation
3. Linear output layer with Softmax activation

In the initialization part, weights are initialized from a Gaussian distribution with a mean of 0 and a standard deviation of weight_scale. Biases are initialized to zero.

After the implementation of initialization, the loss function was implemented.

Forward pass:

```
    out_1,  cache_1 = Conv_ReLU_Pool.forward(X, W1, b1, conv_param, pool_param)
    out_2,  cache_2 = Linear_ReLU.forward(out_1, W2, b2)
    scores, cache_3 = Linear.forward(out_2, W3, b3)
```

Backward pass:

```
loss, soft_max  = softmax_loss(scores, y)

loss += self.reg * (torch.sum(W1 ** 2) + torch.sum(W2 **2) + torch.sum(W3 **2))

dX3, grads['W3'], grads['b3'] = Linear.backward(soft_max, cache_3)
dX2, grads['W2'], grads['b2'] = Linear_ReLU.backward(dX3, cache_2)
dX1, grads['W1'], grads['b1'] = Conv_ReLU_Pool.backward(dX2, cache_1)

grads['W3'] = grads['W3'] + 2 * self.reg * W3
grads['W2'] = grads['W2'] + 2 * self.reg * W2
grads['W1'] = grads['W1'] + 2 * self.reg * W1
```

As you can see in above, just used previously implemented functions.

Sanity Check Loss

The loss values are reasonable. No regularization is approximately equal to -log(1/number of classes:10). When regularization added to loss, it penalizes large weights, and this increases the overall loss. As result, loss calculations are behaving as expected.

```
Initial loss (no regularization):  2.3025846093194815
Initial loss (with regularization):  2.7148746692349603
```

Gradient Check

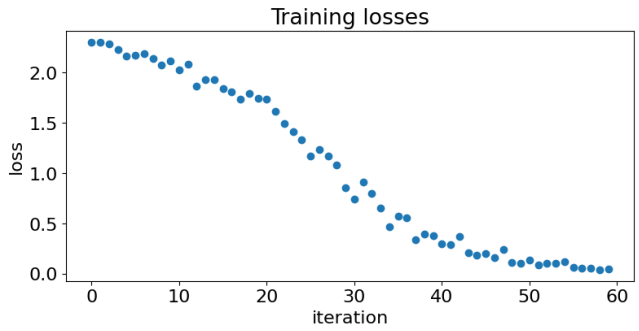
The errors are all less than 1e-5. Backward pass is likely implemented correctly.

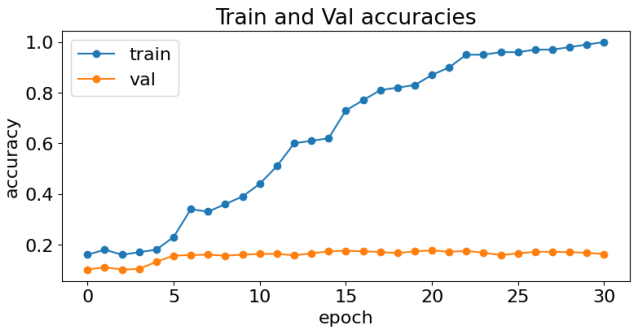
```
W1 max relative error: 3.111514e-08
W2 max relative error: 6.098729e-08
W3 max relative error: 8.701769e-09
```

```
b1 max relative error: 1.123378e-07
b2 max relative error: 2.290706e-08
b3 max relative error: 1.990082e-09
```

Overfit Small Data

When tested with a few training samples, overfit was observed as expected, with very high training accuracy and low validation accuracy.





Train The Net

Greater than 50% accuracy on the training set was achieved by training the three-layer convolutional network for one epoch.

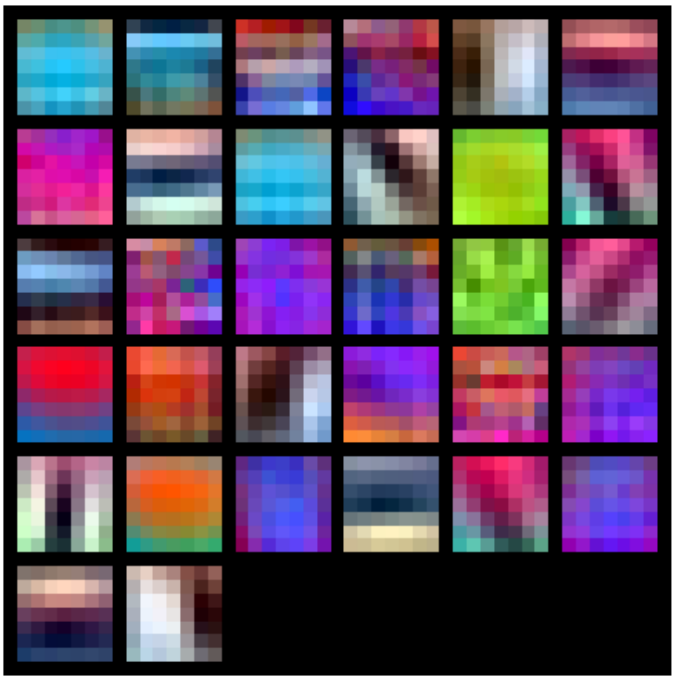
```
(Time 0.09 sec; Iteration 1 / 625) loss: 2.306690
(Epoch 0 / 1) train acc: 0.091000; val_acc: 0.102500
(Time 0.68 sec; Iteration 51 / 625) loss: 1.927303
(Time 1.02 sec; Iteration 101 / 625) loss: 1.783371
(Time 1.35 sec; Iteration 151 / 625) loss: 1.690697
(Time 1.68 sec; Iteration 201 / 625) loss: 1.860708
(Time 2.02 sec; Iteration 251 / 625) loss: 1.672931
(Time 2.35 sec; Iteration 301 / 625) loss: 1.702446
(Time 2.68 sec; Iteration 351 / 625) loss: 1.535979
(Time 3.01 sec; Iteration 401 / 625) loss: 1.764157
(Time 3.35 sec; Iteration 451 / 625) loss: 1.625230
(Time 3.68 sec; Iteration 501 / 625) loss: 1.638446
(Time 4.02 sec; Iteration 551 / 625) loss: 1.491076
(Time 4.36 sec; Iteration 601 / 625) loss: 1.532202
(Epoch 1 / 1) train acc: 0.511000; val_acc: 0.498200
```

Visualize Filters

It is apparent that they have noisy patterns.

In fact, “well-trained networks usually display nice and smooth filters without any noisy patterns. Noisy patterns can be an indicator of a network that hasn’t been trained for long enough, or possibly a very low regularization strength that may have led to overfitting.”

In our case, the cause of overfitting is most likely small data.



CONCLUSION

- Convolutional layers are a key component of convolutional neural networks (CNNs) and are responsible for extracting features from input data. In this report, we successfully implemented the forward and backward pass functions for a convolutional layer, demonstrating a clear understanding of its operation and achieving accurate results.
- Max pooling is a downsampling operation used in CNNs to reduce the spatial dimensions of feature maps while preserving important information.
- The performance of both naive and fast implementations of convolutional and max-pooling layers are investigated, finding that the fast versions significantly outperformed the naive ones in terms of speed.
- Convolutional sandwich layers combine Convolutional layer with ReLU activation and 2x2 max pooling, Linear layer with ReLU activation, and Linear output layer with Softmax activation, forming a fundamental building block for CNN architectures.
- A three-layer convolutional network was implemented, employing appropriate initialization techniques, a loss function, and forward and backward passes. The network achieved greater than 50% accuracy on the training set after just one epoch, demonstrating its ability to learn from data.