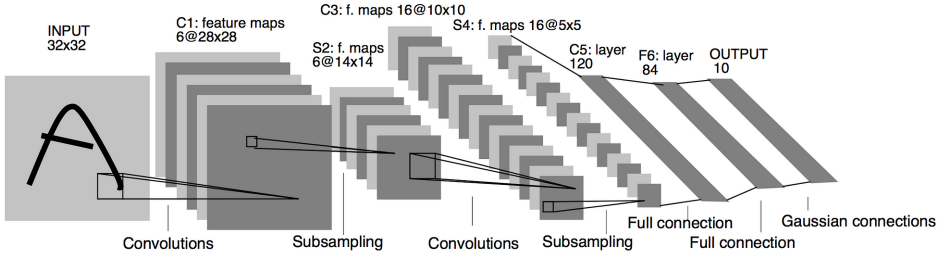
CS461 HW4

Sumanta Das Krishaan Chaudhary Zuhayr Rashid

1 LeNet-5 Implementation

The LeNet-5 architecture is a historically innovative convolutional neural network designed to recognize handwritten characters and digits. Its original implementation is made of seven layers:



The network receives a 32x32 input image of a handwritten digit. It then passes four convolutional layers reducing the input to 16 maps of 5x5 grids. The next layer (C5) is labelled as a convolutional layer but acts exactly as a fully connected layer. Hence, the next two layers (C5 and F6) are fully connected layers reducing the 16 maps to just 84 neurons. This layer is then compared with predefined constant 7x12 drawings of digits, and the final classification is the predefined drawing most similar to the 84 neuron-wide layer.

The parameter initialization for the convolutional and fully connected layers are determined by their fan-in number:

$$F_i = \begin{cases} (\text{Kernel Size})^2 \times \#\text{ of Input Channels} & \text{If Convolutional} \\ \#\text{ of Input Neurons} & \text{If Fully Connected} \end{cases}$$

The parameters for each layer are then initialized by a uniform distribution from $-\frac{2.4}{F_i}$ to $\frac{2.4}{F_i}$.

Our implementation is done with `pytorch`, which provides implementations for convolutional (`torch.nn.Conv2d`) and fully connected (`torch.nn.Linear`) layers, but we must implement custom layers to specify this non-default uniform parameter initialization distribution:

```
# Uniform Convolutional Layer
class LeNetConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        super(LeNetConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride)

        f_in = kernel_size * kernel_size * in_channels
        nn.init.uniform_(self.conv.weight, a=-2.4/f_in, b=2.4/f_in)

    def forward(self, x):
        return self.conv(x)

# Uniform Fully Connected Layer
class LeNetLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super(LeNetLinear, self).__init__()
        self.linear = nn.Linear(in_features, out_features)

        f_in = in_features
        nn.init.uniform_(self.linear.weight, a=-2.4/f_in, b=2.4/f_in)

    def forward(self, x):
        return self.linear(x)
```

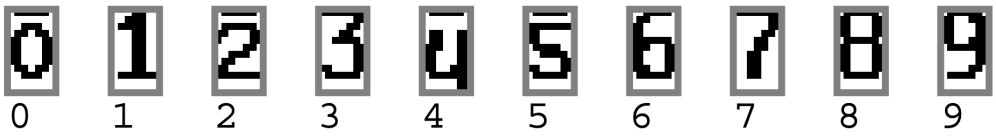
The original architecture begins with four convolutional layers:

1. The first layer (C1) has a kernel size of 5x5 and produces 6 output channels, reducing the input image to 6 channels of 28x28 feature maps.
2. The second layer (S2) has a kernel size of 2x2 and a stride of 2, essentially halfing the size of each feature map to 14x14.
3. The third layer (C3) has a kernel size of 5x5 and produces 16 output channels, reducing the previous layer to 16 channels of 10x10 feature maps.
4. The fourth and final convolutional layer (S4) is identical to S2, having a kernel size of 2x2 and a stride of 2. This layer also halfs the size of each feature map to 5x5.

The 16 layers of 5x5 feature maps are unwrapped to a linear sequence of

$16 \times 5 \times 5$ neurons which are passed through two fully connected layers (C5 and F6), reducing the layer to 120 and then 84 neurons respectively.

The last layer compares the output 84 neurons with predefined constant 7x12 drawings of digits called *parameter vectors*, with each element being either 1 or -1. The paper uses the following constant parameter vectors in their implementation:



Implementing this final layer includes hardcoding these parameter vectors:

```
class ParamVecs:
    ZERO = [
         1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,
         1,  1, -1, -1, -1,  1,  1,
         1, -1, -1,  1, -1, -1,  1,
        -1, -1,  1,  1,  1, -1, -1,
        -1, -1,  1,  1,  1, -1, -1,
        -1, -1,  1,  1,  1, -1, -1,
        -1, -1,  1,  1,  1, -1, -1,
         1, -1, -1,  1, -1, -1,  1,
         1,  1, -1, -1, -1,  1,  1,
         1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  1,  1,  1
    ]
    # define other constant parameter vectors from one to nine
```

The final layer outputs 10 neurons, representing each of the ten digits. The value of each neuron (y_i) is the Euclidean distance (sum of squared differences between corresponding elements) between the previous layer's 84 neurons (x) and the corresponding constant parameter vector (w_i):

$$y_i = \sum_j (x_j - w_{ij})^2$$

The final layer is implemented below:

```
class RbfLayer(nn.Module):
    def __init__(self):
        super(Gaussian, self).__init__()
        self.param_vecs = nn.Parameter(torch.tensor([
            ParamVecs.ZERO,
            ParamVecs.ONE,
```

```
            ParamVecs.TWO,
            ParamVecs.THREE,
            ParamVecs.FOUR,
            ParamVecs.FIVE,
            ParamVecs.SIX,
            ParamVecs.SEVEN,
            ParamVecs.EIGHT,
            ParamVecs.NINE
        ], dtype=torch.float32))

        # parameter vectors must stay constant
        self.param_vecs.requires_grad = False

    def forward(self, x):
        x_expanded = x.unsqueeze(1).expand(-1, 10, -1)
        return torch.norm(x_expanded - self.param_vecs, dim=2)
```

We now have all the prerequisite layers to define the overall neural network:

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.C1 = LeNetConv2d(1, 6, kernel_size=5, stride=1)
        self.S2 = LeNetConv2d(6, 6, kernel_size=2, stride=2)
        self.C3 = LeNetConv2d(6, 16, kernel_size=5, stride=1)
        self.S4 = LeNetConv2d(16, 16, kernel_size=2, stride=2)
        self.C5 = LeNetLinear(16*5*5, 120)
        self.F6 = LeNetLinear(120, 84)
        self.Rbf = RbfLayer()
```

The original paper's activation function of choice for both its convolutional and fully connected layers is:

$$f(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$$

We can now show the full forward propagation step for the network:

```
class LeNet(nn.Module):
    def __init__(self, num_classes=10):
        ...

    def forward(self, x):
        x = 1.7159 * torch.tanh(self.C1(x) * 2/3)
        x = 1.7159 * torch.tanh(self.S2(x) * 2/3)
        x = 1.7159 * torch.tanh(self.C3(x) * 2/3)
        x = 1.7159 * torch.tanh(self.S4(x) * 2/3)
        x = x.view(-1, 16*5*5)
        x = 1.7159 * torch.tanh(self.C5(x) * 2/3)
        x = 1.7159 * torch.tanh(self.F6(x) * 2/3)
        x = self.Rbf(x)
        return x
```

Typically, the neuron with the highest activation is the predicted label, but this is not the case with this network. The more similar the 84 neurons are to a parameter vector, the lower the Euclidean distance between the two are. In other words, the neuron with the *lowest* activation is the predicted label for LeNet-5.

The back propagation step requires a loss function that handles this unusual activation scheme. The paper presents the following loss function for some sample batch B. Let $\vec{o}$ be the final vector of the ten output neuron activations:

$$L(B, j) = \frac{1}{|B|} \sum_{(X,y) \in B} \vec{o}_y + \log(e^{-j} + \sum_{y \neq i \in \{1,2,\ldots,|\vec{o}|\}} e^{-\vec{o}_i})$$

In other words, for every sample in the batch, find the activation of the output neuron corresponding to the actual label ($\vec{o}_y$). Then, add a regularization term, $\log(\ldots)$ that penalizes low activation of incorrect classes and prevents overfitting. This is the loss value of this particular sample. The overall loss of the batch is then the average of all of the individual sample losses in the batch.

It's worth mentioning that j is a regularization hyperparameter that must be defined before the model's training, which the assignment sets to $j = 0.1$.

This loss function is implemented below:

```
class LeNetLoss(nn.Module):
    def __init__(self):
        super(LeNetLoss, self).__init__()

    def forward(self, output, labels):
        j = torch.tensor([0.1])
        oy = output[range(output.size(0)), labels]

        e_incorrect = torch.exp(-output)
        e_mask = torch.arange(10).unsqueeze(0).expand(output.size(0), -1) !=
    labels.unsqueeze(1)
        e_incorrect = e_incorrect * e_mask

        reg = torch.log(torch.exp(-j) + e_incorrect.sum(dim=1))
        return (oy + reg).mean()
```

The assignment asks us to implement a constant step size on the network's parameters θ as its optimization algorithm:

$$\theta_{t+1} = \theta_t + \eta \nabla_\theta L$$

This algorithm is directly implemented in PyTorch's `torch.optim.SGD`:

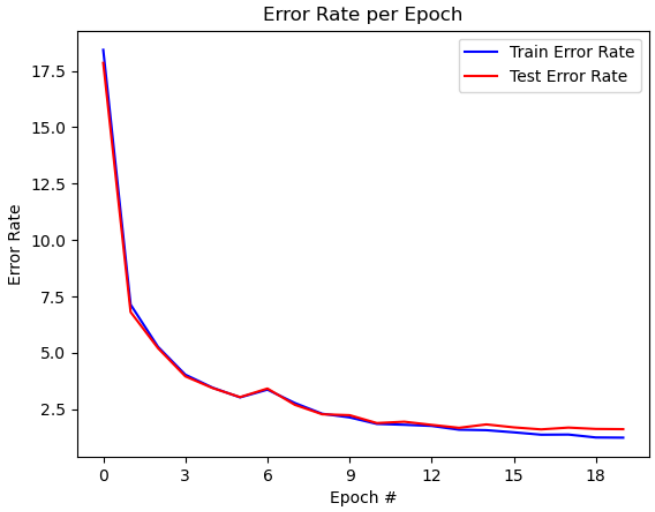
```
model = LeNet()

# replace 'lr' with constant step size of choice
optimizer = torch.optim.SGD(model.parameters(), lr=0.02)
```

A step size of $\eta = 0.02$ seems to give us the best results, with an inference accuracy rate of $\approx 98.5\%$ after 20 epochs of training.

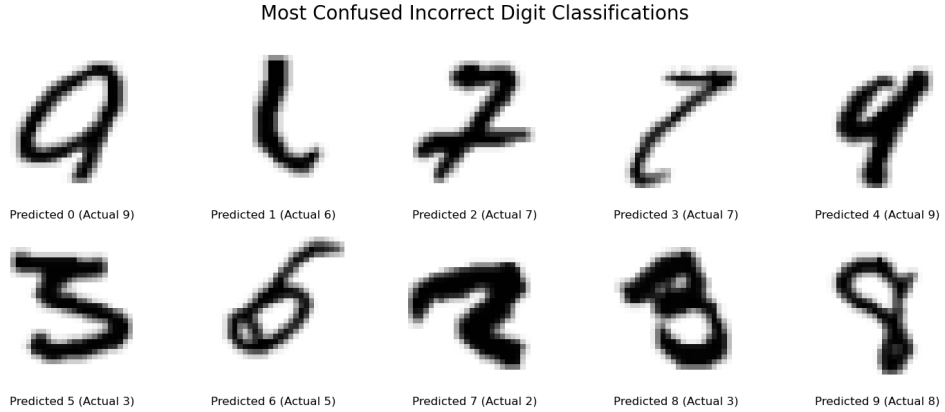
The 10x10 confusion matrix for this model after 20 epochs of training is shown below:

$$\begin{bmatrix} 972 & 0 & 4 & 0 & 1 & 2 & 2 & 0 & 1 & 1 \\ 0 & 1128 & 0 & 0 & 2 & 0 & 3 & 3 & 0 & 0 \\ 0 & 3 & 1015 & 1 & 1 & 0 & 2 & 6 & 1 & 0 \\ 0 & 2 & 2 & 993 & 0 & 5 & 1 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 & 962 & 0 & 0 & 0 & 2 & 3 \\ 0 & 2 & 0 & 3 & 0 & 878 & 2 & 0 & 2 & 5 \\ 4 & 0 & 0 & 0 & 2 & 1 & 944 & 0 & 2 & 1 \\ 2 & 0 & 6 & 8 & 0 & 1 & 0 & 1016 & 2 & 6 \\ 2 & 0 & 3 & 4 & 2 & 5 & 3 & 0 & 953 & 8 \\ 0 & 0 & 2 & 1 & 12 & 0 & 1 & 0 & 1 & 984 \end{bmatrix}$$

The graph comparing error rates over n epochs of training up to 20 is shown below:



The input images corresponding to the most confused incorrect classifications for each digit are shown below:



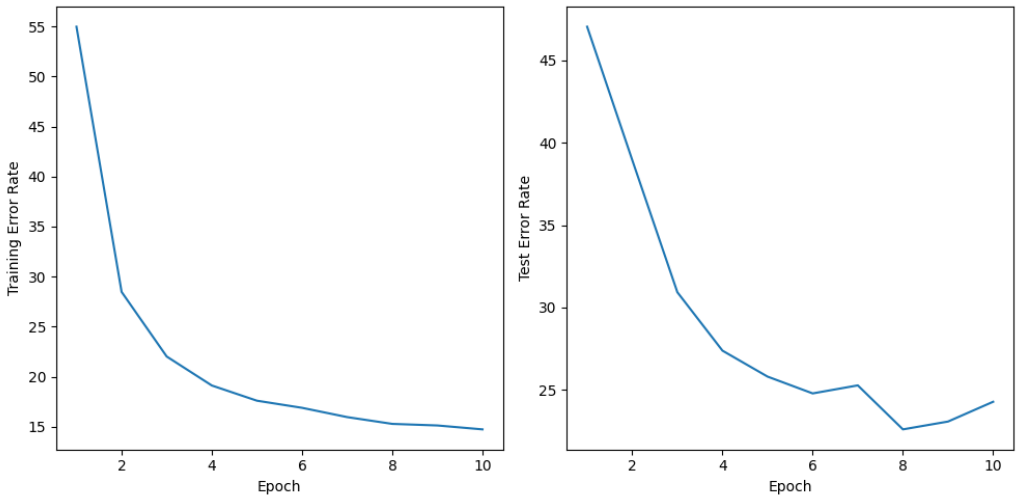
After 20 epochs, the model outputs a 98.61% accuracy on the training dataset and a 98.38% accuracy on the test dataset.

2 Modification of LeNet-5 to Handle Unseen Data

2.1 Plan:

Here are the modifications to the original LeNet Architecture implemented in our plan:

1. **Modified Training Dataset**: Modified the training data set by taking a certain portion of the images in the original training data set and applied rotations, scalings, translations, and shearings to simulate unseen dataset as described in the affNIST dataset.
2. **Activation Function**: replaced tanh activation function with ReLU function for all layers to learn faster and achieving overall better computational efficiency.
3. **Downsampling**: We used Max pooling in a separate layer instead of strided convolutions to focus on the most dominant features in a region and ensure translational invariance.
4. **Output Layer**: Replace the RBF layer with a fully connected layer for simplicity.
5. **Loss Function**: We replaced our original loss function with Cross Entropy Loss for simplicity and computational efficiency. It is also the standard for multi-class classification problems.
6. **Momentum**: We incorporated momentum into our optimizer to reduce oscillations and reach optimial solution quicker.



2.2 Results and Discussion:

We were able to achieve approximately 97% accuracy on the original MNIST dataset when training on a combination of transformed images and untransformed images. During the training phase of the model, the training and test accuracies reached approximately 80% on the dataset which consisted of a combination of transformed and untransformed images.