NANDHA ENGINEERING COLLEGE, AUTONOMOUS, ERODE -52

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ASSIGNMENT -1

ACADEMIC YEAR: 2024-2025

TEAM NUMBER	**: 15**
REGISTER NUMBER	**: 22CS088** **22CS087**
TEAM MEMBERS NAME	**: SHARMILA B** **SATHIYASUDHAN M**
COURSE CODE	**: 22CSX01**
COURSE NAME	**: DEEP LEARNING**
YEAR / CLASS	**: III CSE / 5th SEM – ‘B’ SECTION**

Faculty Signature

What are the differences between ELU and SELU activation functions, and in what scenario would you prefer using SELU in a deep CNN model?

Difference between ELU and SELU Activation Function:
The ELU (Exponential Linear Unit) and SELU (Scaled Exponential Linear Unit) are activation functions commonly used in neural networks, particularly in deep learning. Here's a breakdown of the differences and use cases for each:

1. ELU (Exponential Linear Unit)

- **Formula:**

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0 \end{cases}$$

where α is a hyperparameter that controls the saturation point for negative values.

- **Characteristics:**
  - ELU introduces nonlinearity like ReLU but has an exponential decay for negative values instead of zeroing them out.
  - Helps mitigate the "dying ReLU" problem (where neurons become inactive for all inputs) by allowing negative outputs.
  - Gradient Smoothness: Because it's continuous, it has smoother gradients than ReLU and Leaky ReLU, which can improve training.
  - Hyperparameter: α\alphaα is often set to 1, though it can be tuned.
- **Use Cases:**
  - ELU is often used when you want faster convergence in neural networks, particularly in shallow or moderately deep networks.

2. SELU (Scaled Exponential Linear Unit)

- **Formula:**

$$\mathrm{SELU}(x) = \lambda \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0 \end{cases}$$

where $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$ are fixed constants.

- **Characteristics:**
  - SELU is a scaled version of ELU and is specifically designed to enable self-normalizing properties in neural networks.

- Self-Normalizing Networks: SELU was introduced to keep activations in a normalized range automatically, which stabilizes training.
- Fixed Constants: The constants α\alphaα and λ\lambdaλ are determined to enable self-normalization, and unlike ELU, they're not typically modified.

- **Use Cases:**
    - SELU is commonly used in Self-Normalizing Neural Networks (SNNs) for deep learning tasks.
    - Works best with fully connected layers with normalized inputs, and usually requires specific weight initialization and no dropout layers.

Key Differences

Features	ELU	SELU
Formula	Includes α	Includes α and γ
Self-Normalization	No	Yes
Parameters	α (typically 1)	Fixed $\alpha \approx 1.6733$, $\gamma \approx 1.0507$
Use in Networks	General neural networks	Self-Normalizing Neural Networks (SNNs)

CODE:

Segment 1:

Import Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

Segment 2:

Define the CNN Model

```
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
        self.se = nn.SELU()

    def forward(self, x):
        x = self.se(self.conv1(x))
        x = nn.MaxPool2d(2)(x)
        x = self.se(self.conv2(x))
        x = nn.MaxPool2d(2)(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.se(self.fc1(x))
        x = self.fc2(x)
        return x
```

Segment 3:

Load the MNIST Dataset

```
transform = transforms.Compose([transforms.ToTensor()])
trainset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
shuffle=True)
```

Segment 4:

Initialize Model, Loss Function, and Optimizer

```
model = CNNModel()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Segment 5:

Train the Model

```
num_epochs = 5
losses = []

for epoch in range(num_epochs):
    running_loss = 0.0
    for images, labels in trainloader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    epoch_loss = running_loss / len(trainloader)
    losses.append(epoch_loss)
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}')
```

Segment 6:

Plot the Training Loss

```
plt.plot(losses)
plt.title('Training Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```

Output

```
Epoch [1/5], Loss: 0.2154
Epoch [2/5], Loss: 0.0901
Epoch [3/5], Loss: 0.0602
Epoch [4/5], Loss: 0.0453
Epoch [5/5], Loss: 0.0351
```

```
Training Loss over Epochs
|
|          *
|         *
|        *
|       *
|      *
|     *
|    *
|   *
|  *
| *
|________________________
  1   2   3   4   5
```