Understanding Activation Functions in Deep Learning

A Practical Tutorial on ReLU, Leaky ReLU, and ELU with Fashion-MNIST

Author	Md Shahedur Rahman
StudentID	23036883
Course Module	Machine Learning and Neural Networks
Assignment	Individual assignment: Machine learning tutorial

Tutorial Outline

- Introduction
- Dataset Preparation
- Building Models with Different Activation Functions
- Training and Comparing Models

Tutorial Outline

- **Advanced Visualization: Decision Boundaries**
- **Observations and Insights**
- **Conclusion**
- **References**

Introduction

What Are Activation Functions?

- Activation functions introduce non-linearity into neural networks, enabling them to learn complex patterns (Goodfellow et al., 2016).
- They determine how input signals are transformed into output signals at each layer of the network.

Introduction

Why Are Activation Functions Crucial in Deep Learning?

- Enable networks to approximate non-linear relationships between input and output.
- Allow stacking of multiple layers to form deep architectures.
- Examples of activation functions:
  - **ReLU**: Simplicity and efficiency.
  - **Leaky ReLU**: Gradient flow improvements.
  - **ELU**: Smooth convergence for deeper models.

Understanding Activation Functions

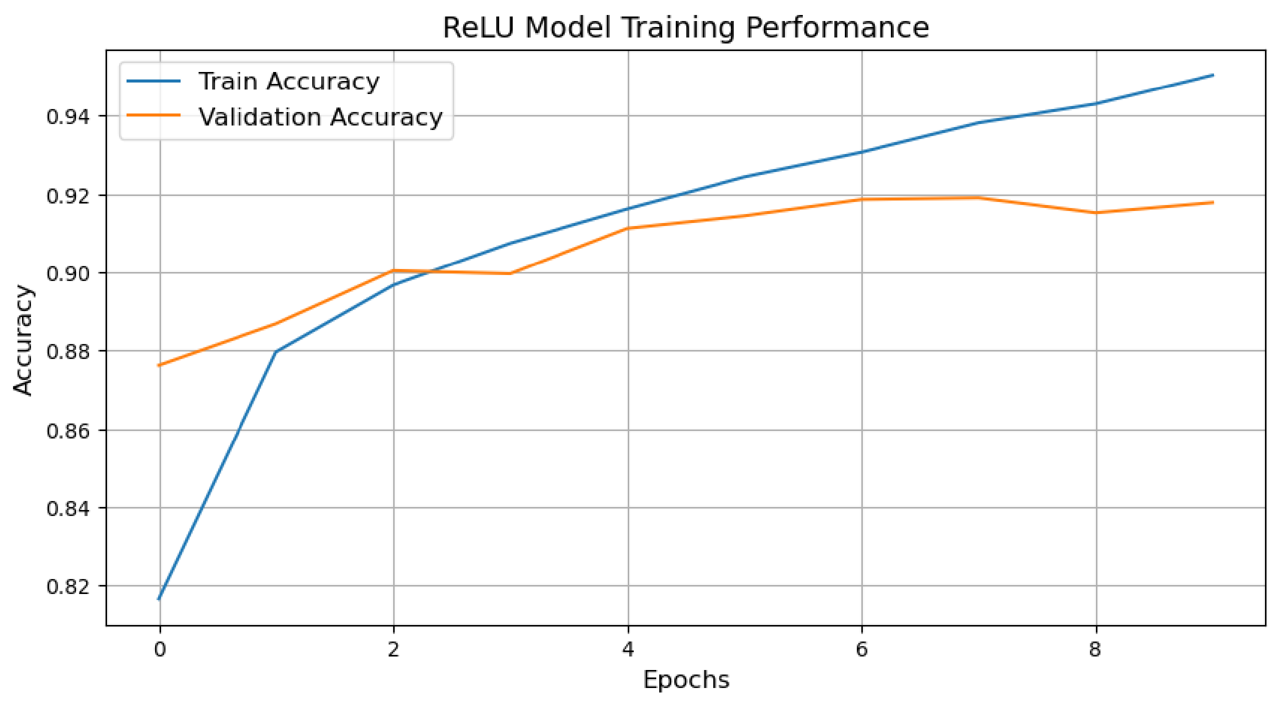
ReLU: The Most Widely Used Activation Function

- **Definition**:
  - Rectified Linear Unit (ReLU) is defined as:

    $f(x) = \max(0, x)$
  - Introduced in the context of deep learning to improve gradient flow and computational efficiency (He et al., 2015).
- **Advantages**:
  - Computational efficiency.
  - Introduces sparsity into activations.
- **Limitations**:
  - Dying ReLU problem: Neurons become inactive for negative inputs.

Understanding Activation Functions

ReLU: The Most Widely Used Activation Function



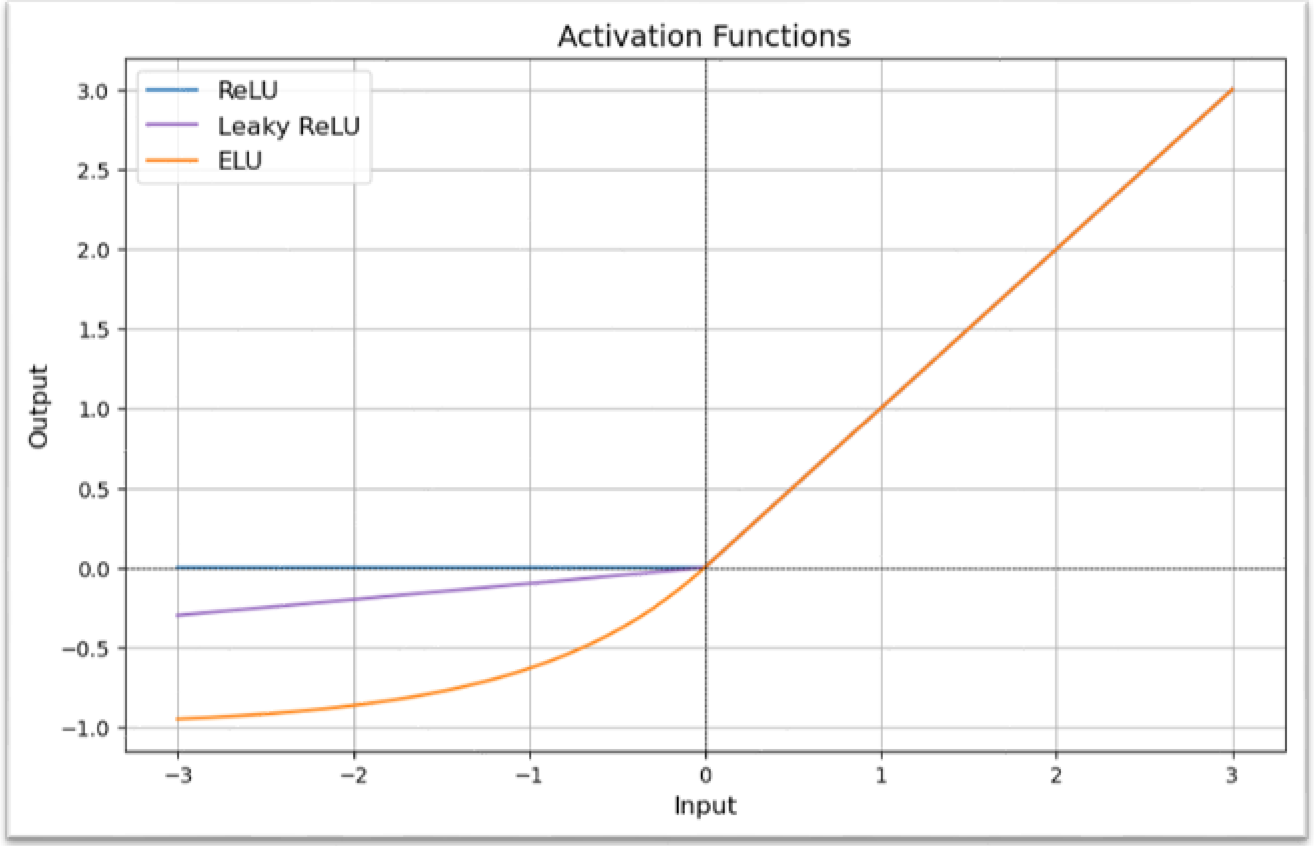
Understanding Activation Functions

Leaky ReLU: Addressing the "Dying ReLU" Problem

- **Definition**:
  - Leaky ReLU allows a small gradient for negative inputs: $f(x)=x$ if $x>0$, otherwise $f(x)=\alpha x$, where α is a small positive constant (Maas et al., 2013).
- **Advantages**:
  - Resolves the dying ReLU issue by allowing gradient flow for negative inputs.
- **Limitations**:
  - The small slope in negative regions can still result in slow training in some cases.

Understanding Activation Functions

Leaky ReLU: Addressing the "Dying ReLU" Problem



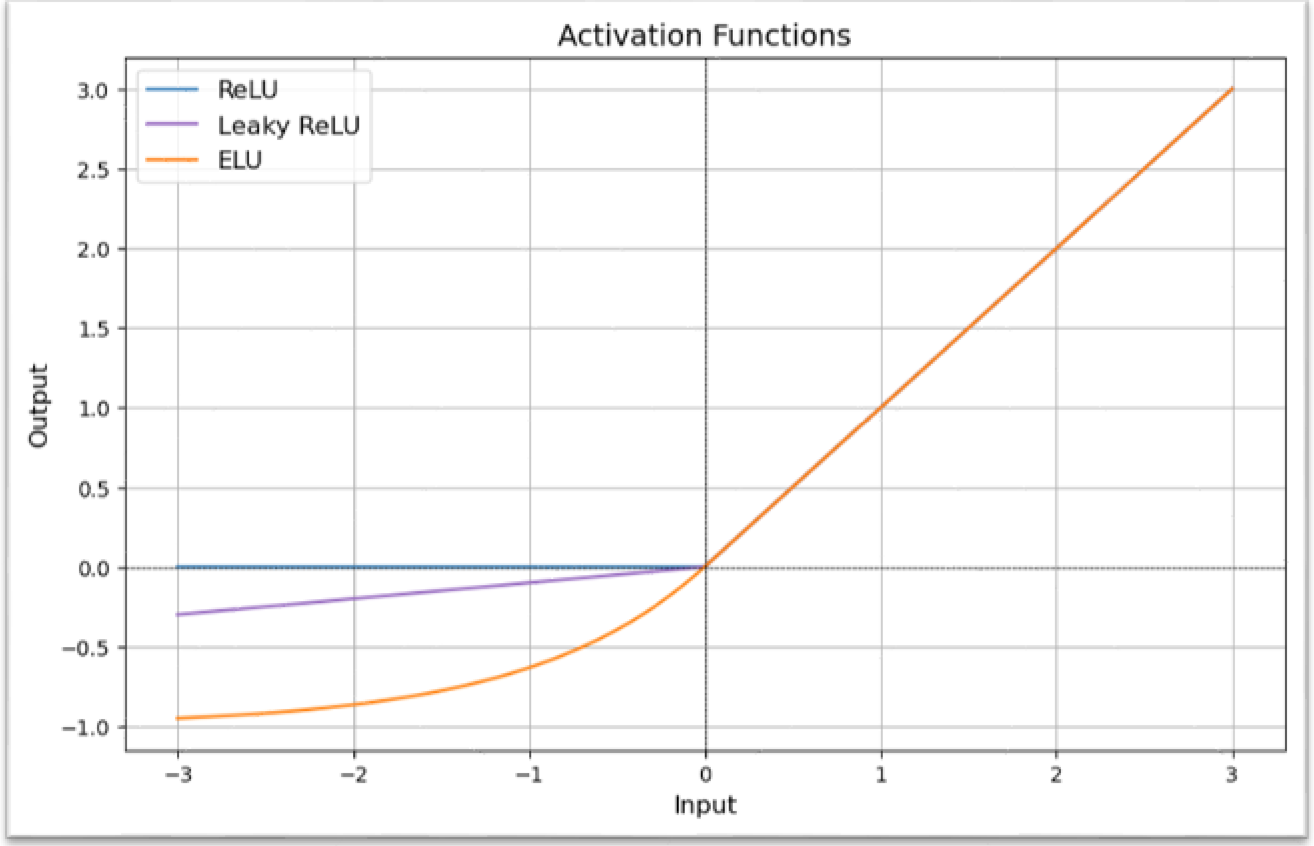
Understanding Activation Functions

ELU: Smoothing Convergence for Deep Networks

- **Definition**:
  - ELU (Exponential Linear Unit) outputs: $f(x)=x$ if $x>0$, otherwise $f(x)=\alpha(\exp(x)-1)$ (Clevert et al., 2015).
- **Advantages**:
  - Smooth transition between negative and positive inputs improves optimization.
- **Limitations**:
  - Slightly more computationally expensive due to the exponential operation.

Understanding Activation Functions

ELU: Smoothing Convergence for Deep Networks



Dataset Preparation

Introduction to the Fashion-MNIST Dataset

- **Fashion-MNIST**:
  - A dataset of 70,000 grayscale images of fashion items, split into 10 classes (e.g., shirts, shoes) (Xiao et al., 2017).
  - Challenges: Diverse and ambiguous patterns, making it ideal for testing neural networks.

Dataset Preparation

Data Preprocessing Steps

- Normalize pixel values to a range of [0, 1].
- Add a channel dimension for compatibility with Conv2D layers.
- Split into training, validation, and test sets.

Dataset Preparation

Data Preprocessing Steps

```
# Load Fashion-MNIST dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize the data
x_train = x_train / 255.0  # Scale images to [0, 1]
x_test = x_test / 255.0

# Add a channel dimension (required for Conv2D)
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

# Convert labels to categorical (one-hot encoding)
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Split validation data from training
x_val, y_val = x_train[-5000:], y_train[-5000:]
x_train, y_train = x_train[:-5000], y_train[:-5000]

print(f"Training set: {x_train.shape}, Validation set: {x_val.shape}, Test set: {x_test.shape}")
```

Building Models with Different Activation Functions

Model Architecture for ReLU

- A simple convolutional neural network with:
  - 2 Conv2D layers using ReLU activation.
  - 2 MaxPooling2D layers for downsampling.
  - 1 Dense layer with 128 units and ReLU.
  - A final Dense layer with softmax for classification.

Building Models with Different Activation Functions

Model Architecture for ReLU

```
def create_relu_model():
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')  # 10 classes for Fashion-MNIST
    ])
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

relu_model = create_relu_model()
relu_model.summary()
```

Building Models with Different Activation Functions

Model Architecture for Leaky ReLU

- Similar to the ReLU model but replaces ReLU with Leaky ReLU:
  - Conv2D layers with Leaky ReLU activation (alpha=0.1).
  - Dense layers with Leaky ReLU activation.

Building Models with Different Activation Functions

Model Architecture for Leaky ReLU

```
def create_leaky_relu_model():
    model = Sequential([
        Conv2D(32, (3, 3)),
        LeakyReLU(alpha=0.1),  # Leaky ReLU activation
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3)),
        LeakyReLU(alpha=0.1),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128),
        LeakyReLU(alpha=0.1),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

leaky_relu_model = create_leaky_relu_model()
```

Building Models with Different Activation Functions

Model Architecture for ELU

- Similar to the previous models but replaces activation functions with **ELU**:
  - Conv2D layers with ELU activation.
  - Dense layers with ELU activation for smooth gradients.

Building Models with Different Activation Functions

Model Architecture for ELU

```
def create_elu_model():
    model = Sequential([
        Conv2D(32, (3, 3)),
        ELU(alpha=1.0),  # ELU activation
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3)),
        ELU(alpha=1.0),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128),
        ELU(alpha=1.0),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

elu_model = create_elu_model()
```

Training Models with Different Activation Functions

- **Training Parameters**:
  - Optimizer: Adam
  - Loss Function: Categorical Crossentropy
  - Batch Size: 64
  - Epochs: 10
- Trained models:
  - ReLU Model
  - Leaky ReLU Model
  - ELU Model

Training Models with Different Activation Functions

```
history_relu = relu_model.fit(
    x_train, y_train,
    validation_data=(x_val, y_val),
    epochs=10,
    batch_size=64
)

# Evaluate on test set
relu_test_loss, relu_test_accuracy = relu_model.evaluate(x_test, y_test)
print(f"ReLU Model - Test Loss: {relu_test_loss}, Test Accuracy: {relu_test_accuracy}")
```

Training Models with Different Activation Functions

```
history_leaky_relu = leaky_relu_model.fit(
    x_train, y_train,
    validation_data=(x_val, y_val),
    epochs=10,
    batch_size=64
)

history_elu = elu_model.fit(
    x_train, y_train,
    validation_data=(x_val, y_val),
    epochs=10,
    batch_size=64
)
```

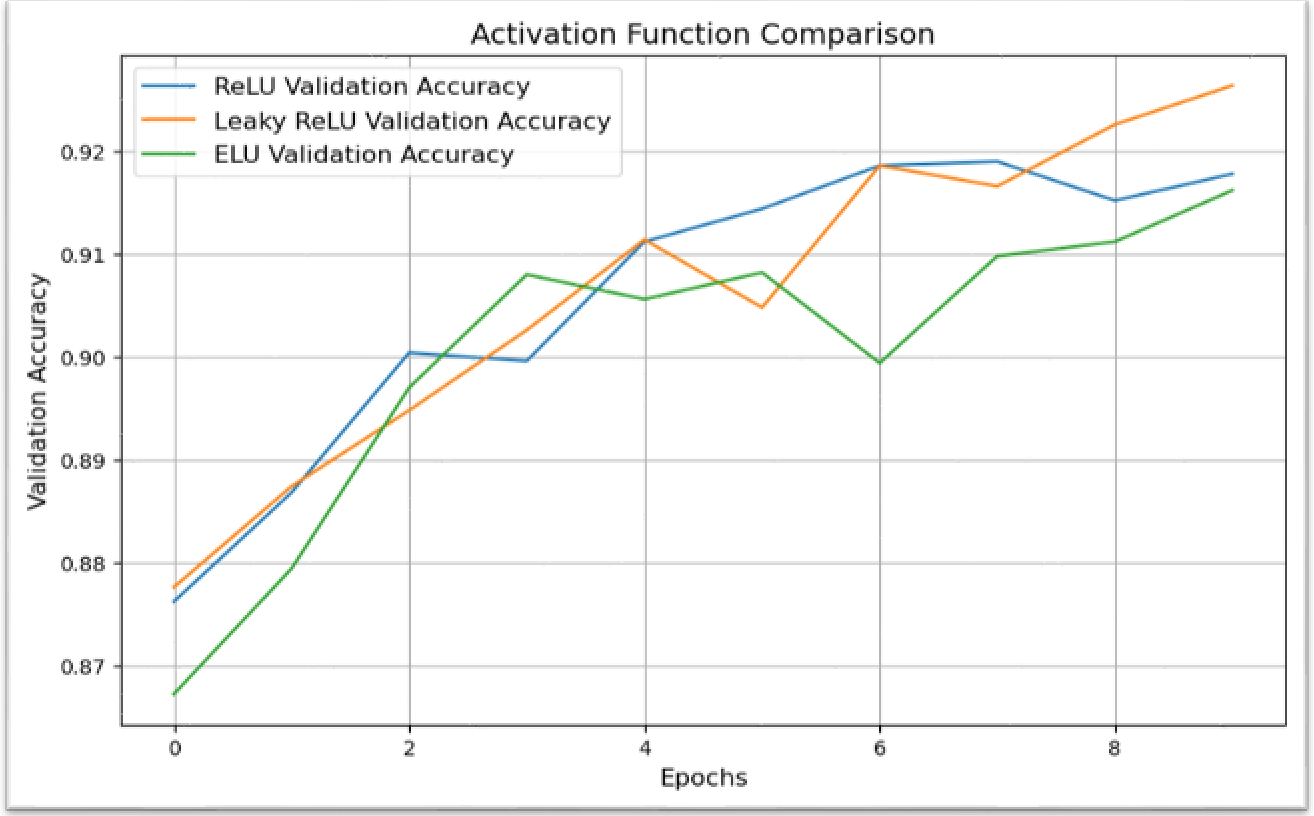
Training Results

Observations:

- All three models converged within 10 epochs.
- Slight differences in validation loss and accuracy, depending on the activation function.

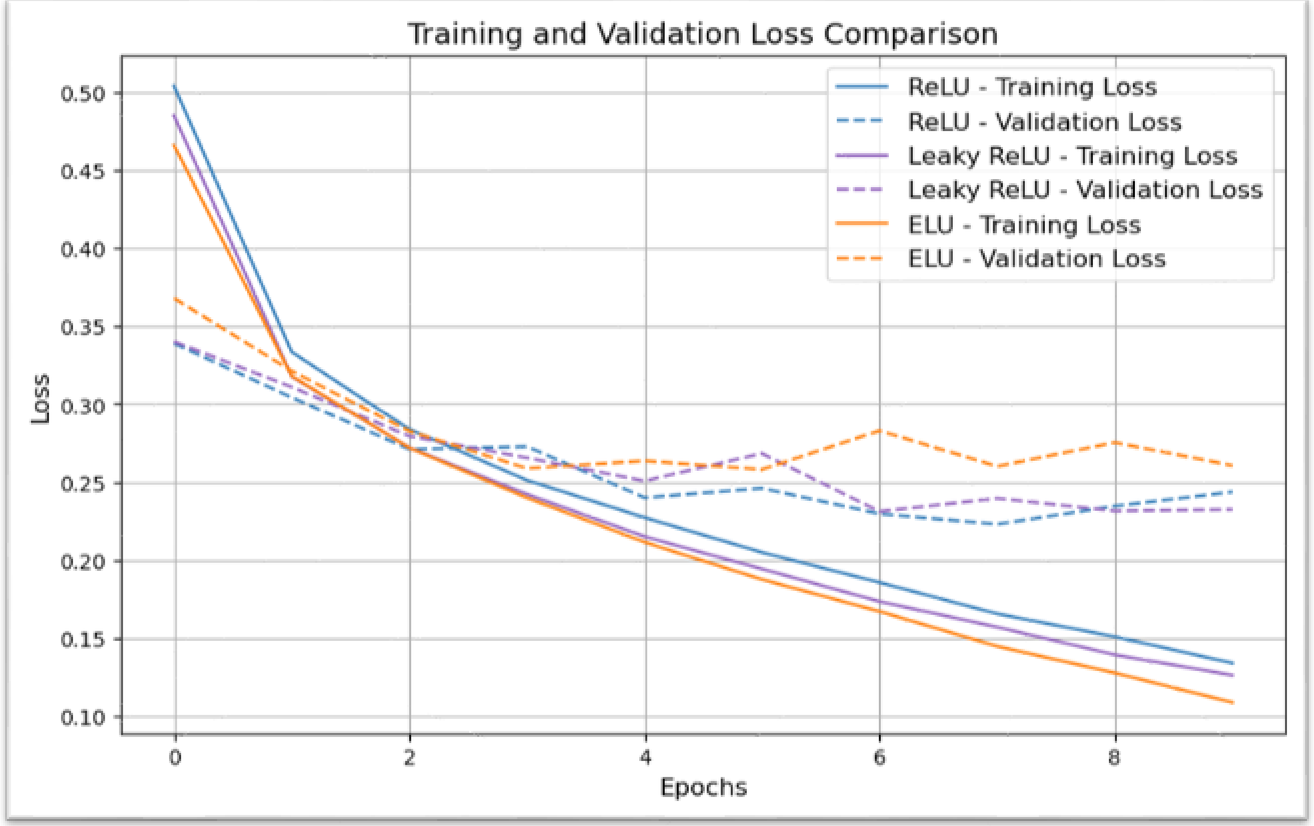
Training Results

The training and validation accuracy comparison plot



Training Results

The training and validation loss comparison plot



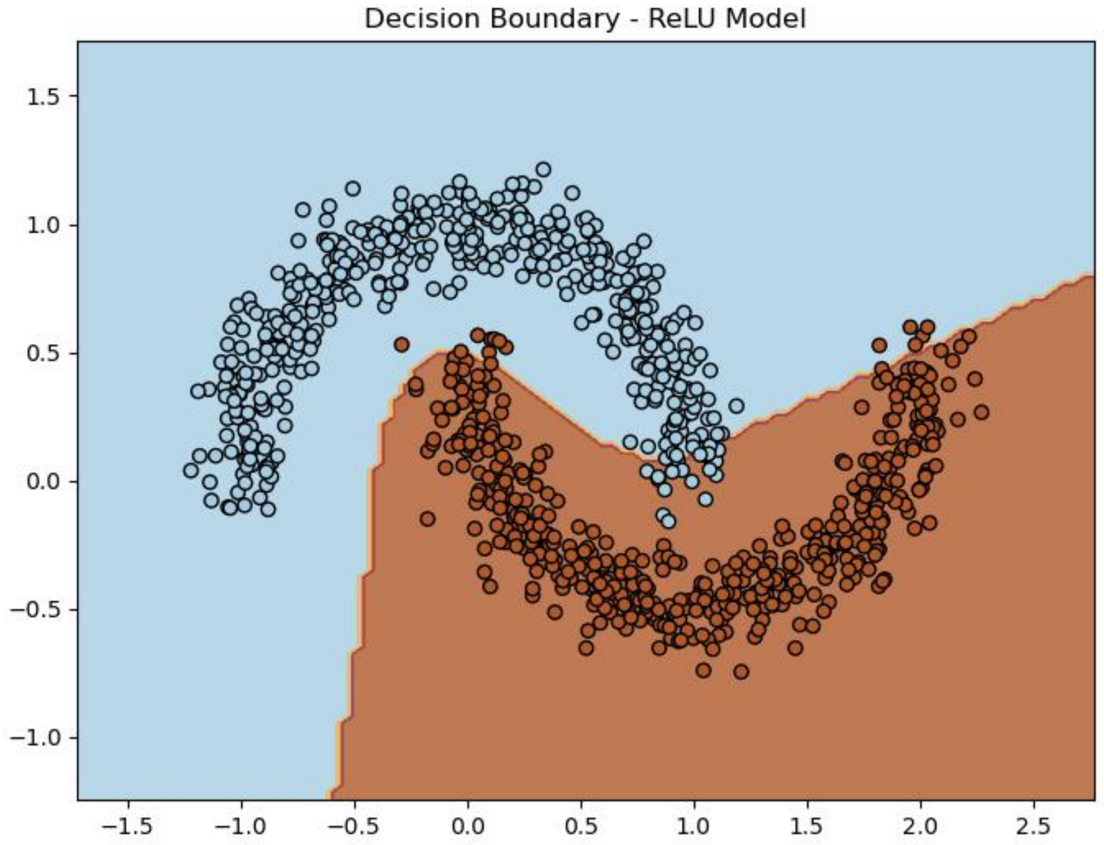
Decision Boundaries

Visualizing Decision Boundaries

- Decision boundaries illustrate how models classify data in input space.
- Differences in boundary smoothness reflect the activation functions used.

Decision Boundaries

Visualizing Decision Boundaries



Observations and Insights

- **Performance Comparison:**
  - ReLU:
    - Fast convergence but prone to dying ReLU for deeper layers.
  - Leaky ReLU:
    - Improved gradient flow but slower convergence.
  - ELU:
    - Smooth convergence but higher computational cost.

Observations and Insights

- **Recommendations:**
  - Use ReLU for general-purpose tasks.
  - Consider Leaky ReLU for avoiding dying ReLU issues.
  - Use ELU for deeper networks requiring smoother optimization.

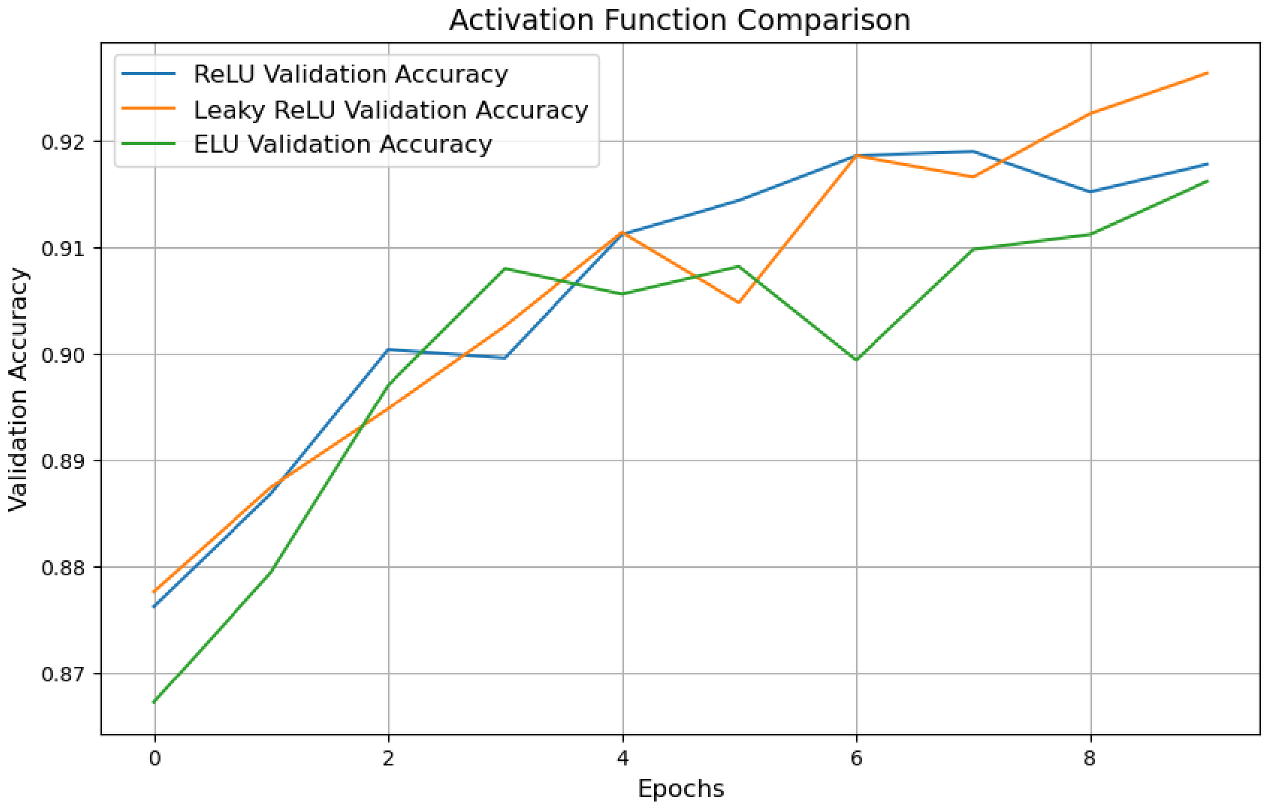
Performance Comparison

Validation Accuracy Comparison

- **ReLU**:
  - Validation accuracy stabilizes around **91%** after 10 epochs.
  - Slight fluctuations in later epochs indicate possible overfitting.
- **Leaky ReLU:**
  - Validation accuracy reaches ~92%, higher than ReLU in the later epochs.
  - Smoothest convergence with reduced overfitting tendencies.
- **ELU:**
  - Validation accuracy starts lower than ReLU and Leaky ReLU but catches up to reach around 91%.
  - Demonstrates smooth convergence but slightly lags in early epochs.

Performance Comparison

Validation Accuracy Comparison



Key Insights

- **When to Use Each Activation Function:**
  - ReLU: Best for shallow or moderately deep networks.
  - Leaky ReLU: Use when avoiding the "dying ReLU" problem.
  - ELU: Ideal for deep networks requiring smoother optimization.
- **Considerations:**
  - Computational cost increases from ReLU to ELU.
  - Choice of activation can significantly impact training dynamics and accuracy.

Conclusion

- Activation functions are crucial for introducing non-linearity in neural networks.
- Each activation function (ReLU, Leaky ReLU, ELU) has distinct advantages and trade-offs.
- Optimal selection depends on:
  - Network depth.
  - Task complexity.
  - Desired computational efficiency.

Reference lists

- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.
- Maas, A.L., Hannun, A.Y., and Ng, A.Y. (2013). *Rectifier Nonlinearities Improve Neural Network Acoustic Models*. Proceedings of ICML.
- Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. arXiv:1511.07289.
- Xiao, H., Rasul, K., and Vollgraf, R. (2017). *Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv:1708.07747.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. arXiv:1502.01852.