**Date of Submission:** **[24/03/2025]**

**Roll Number:** **[160122771034 ,160122771041 ,160122771045]**

**Submitted by:** **[Cheemarla Rithesh Reddy, Gudimella Jayanth, Jammula Pavan Kumar]**

**Title:** **Weight Initialization Techniques**

**Year/Sem/Sec:** **[3/6/1]**

# Case Study 4: Weight Initialization Techniques

## 1. Introduction:

Problem Statement: Weight Initialization

In deep learning, weight initialization significantly impacts the training process. If weights are not properly initialized, it can lead to:

- ***Vanishing Gradients***: When weights are too small, gradients shrink exponentially during backpropagation, making it difficult for deeper layers to learn.
- ***Exploding Gradients:*** When weights are too large, gradients grow uncontrollably, causing instability in training.
- ***Slow Convergence:*** Poor initialization results in inefficient updates, requiring more iterations to reach an optimal solution.
- ***Suboptimal Performance:*** Improper initialization may prevent the model from learning meaningful patterns, reducing overall accuracy.

Importance in Deep Learning:

Proper weight initialization is crucial because it:

- ***Enhances Training Stability****:* Prevents exploding or vanishing gradients, ensuring smooth learning.
- ***Speeds Up Convergence****:* Helps reach an optimal solution faster, reducing training time.
- ***Improves Generalization****:* Enables the network to learn meaningful features instead of getting stuck in bad local minima.
- ***Supports Deep Architectures****:* Makes it possible to train very deep networks effectively.

Common initialization techniques like Xavier (Glorot) and He initialization help maintain a balanced flow of gradients throughout layers.

Real-World Applications

- ***Computer Vision****:* Convolutional Neural Networks (CNNs) for tasks like image classification (e.g., ResNet, VGG) rely on proper initialization for efficient training.

- ***Natural Language Processing (NLP)****:* Transformers (e.g., BERT, GPT) use well-initialized weights to handle long-range dependencies effectively.
- ***Autonomous Vehicles****:* Deep learning models in self-driving cars use weight initialization to ensure stable learning of object detection and lane navigation.
- ***Financial Forecasting****:* Neural networks predicting stock prices or fraud detection rely on proper initialization for accurate and robust predictions

## 2. Methods Compared:

Weight Initialization Techniques:

1. **Zero Initialization** – Not recommended; leads to symmetry issues.
2. **Random Initialization** – Prevents symmetry but may cause vanishing/exploding gradients.
3. **Xavier (Glorot) Initialization** – Maintains variance; best for sigmoid and tanh activations.
4. **He (Kaiming) Initialization** – Optimized for ReLU activations; prevents vanishing gradients.
5. **Lecun Initialization** – Designed for sigmoid/tanh; used in LeNet.
6. **Orthogonal Initialization** – Stabilizes training, useful in RNNs/LSTMs.

## 3. Mathematical Formulation:

1. Zero Initialization

All weights are initialized to zeros.

2. Random Initialization

In random initialization, weights are assigned small random values, usually drawn from a uniform or normal distribution.

Equation: $W \sim N(0, \sigma^2)$ or $W \sim U(-a, a)$

where:

- $N(0, \sigma^2)$ represents a normal distribution with mean 0 and variance $\sigma^2$.
- $U(-a, a)$ represents a uniform distribution over the range $(-a, a)$.
- The choice of $\sigma^2$ or a is arbitrary, and improper scaling can lead to vanishing or exploding gradients.

Explanation:

If weights are too large, activations increase exponentially, leading to **exploding gradients**. If too small, activations shrink, causing **vanishing gradients**. This method does not account for network depth, so better techniques like “Xavier” and “He” initialization were developed.

3. Xavier (Glorot) Initialization

Xavier initialization ensures that the variance of activations remains consistent across layers to prevent vanishing/exploding gradients.

Equation:

$$Uniform\ distribution\ (-\sqrt{\frac{6}{n_i + n_0}}, \sqrt{\frac{6}{n_i + n_0}})$$

$$Normal\ Distribution\ with\ \mu = 0\ and\ \sigma = \sqrt{\frac{2}{n_i + n_0}}$$

where:

- $n_i$ is the number of input neurons to the layer.
- $n_o$ is the number of output neurons.

Derivation:

To ensure variance is preserved, we analyze the forward propagation process:

1. The activation of a neuron at layer l is: $a^l=f(W^la^{l-1}+b^l)$
2. If W is too large, activations grow exponentially. If too small, they shrink.
3. To maintain variance consistency, we set:

$$Var(W^l)=1/n_i+n_0$$

This initialization works best for **sigmoid** and **tanh** activations, as these functions are sensitive to variance changes.

4. He (Kaiming) Initialization

He initialization is designed for **ReLU** and **Leaky ReLU** activations, which do not have symmetric outputs like sigmoid/tanh.

**Equation:**

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

$$W \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}}}}\right)$$

Derivation:

For ReLU activations, only **half** of the values pass through (as negative values are set to zero). This reduces the effective variance, so we compensate by doubling the variance:

1. Without ReLU, we would set:

$$\text{Var}(W) = \frac{1}{n_{\text{in}}}$$

2. Since ReLU discards half of the values, we adjust to:

$$\mathrm{Var}(W) = \frac{2}{n_{\text{in}}}$$

This method prevents vanishing gradients and helps deep networks train effectively with ReLU.

5. Lecun Initialization

LeCun initialization is a weight initialization technique designed to optimize training in networks using **sigmoid** or **tanh** activations. It ensures that the variance of activations remains stable across layers, preventing vanishing or exploding gradients.

Equation: For a weight matrix W of a layer with $n_{in}$ input neurons, the initialization is:

$$W \sim \mathcal{N}\left(0, \frac{1}{n_{in}}\right)$$

Where: $\mathcal{N}$ represents a normal distribution with mean **0** and variance $\frac{1}{n_{in}}$.

6. Orthogonal Initialization

Orthogonal initialization sets the weight matrix to an **orthogonal matrix**, ensuring that gradients do not shrink or explode. This helps preserve variance in deep networks, making training more stable.

Equation: Given a random weight matrix $W_0$ of shape $(n_{out}, n_{in})$, we perform a **QR decomposition**:

$$W, R = QR(W_0)$$

where **QR** is the **Gram-Schmidt orthogonalization** process. The final initialized weight matrix is: $W = Q$ where **Q** is an orthogonal matrix.

## 4. Impact Analysis:

**Zero Initialization**

Zero initialization completely blocks gradient flow since all neurons in a layer update symmetrically, preventing learning. Training is highly unstable because no meaningful weight updates occur. Computational cost is minimal, but it is entirely ineffective. This method is rarely used in practical applications.

**Random Initialization**

Gradient flow is inconsistent, as randomly assigned values can lead to vanishing or exploding gradients. Training stability is unpredictable, especially in deep networks. Computational cost is low,

but it often requires additional techniques like batch normalization to stabilize learning. It is mainly used as a naive baseline for comparisons.

**Xavier (Glorot) Initialization**

Gradient flow is well-maintained as Xavier initialization balances variance across layers, reducing vanishing gradients. Training is stable for shallow to moderately deep networks but struggles with deep ReLU-based architectures. Computational cost is slightly higher due to the scaling factor. This method is ideal for networks with **sigmoid/tanh** activations.

**He (Kaiming) Initialization**

Gradient flow is optimized for **ReLU-based** networks, allowing deep models to train efficiently without vanishing gradients. Training stability is high, making it the preferred choice for deep learning architectures. Computational cost is similar to Xavier initialization, as it only requires variance scaling. This method is widely used in deep **CNNs and MLPs**.

**Lecun Initialization**

Gradient flow is well-maintained for **sigmoid/tanh** networks, preventing exploding gradients. Training stability is good for shallow networks but may require batch normalization for deeper models. Computational cost is low, as the initialization is straightforward. This method is used in networks like **LeNet** and is effective for **shallow architectures**.

**Orthogonal Initialization**

Gradient flow is well-preserved, even in deep networks, as orthogonality prevents variance collapse. Training stability is high, particularly for **RNNs, LSTMs, and deep fully connected networks**, but convergence can be slower. Computational cost is higher due to the need for **QR decomposition**. This method is useful for **deep networks and recurrent architectures**.

## 5. Experimental Evaluation

**Dataset Used**

The following datasets were used for evaluating different weight initialization techniques:

- **MNIST** (28×28 grayscale images of handwritten digits, 10 classes)
- **Fashion-MNIST** (28×28 grayscale images of clothing items, 10 classes)
- **CIFAR-10** (32×32 color images of objects like cars, birds, etc., 10 classes)
- **CIFAR-100** (32×32 color images of 100 object categories)
- **California Housing Dataset** (Regression task predicting housing prices based on 8 numerical features)

**Model Architecture:** We experimented with three types of models:

- **MLP (Multi-Layer Perceptron)**: Fully connected neural network with two hidden layers (256, 128 units)

- **CNN (Convolutional Neural Network)**: Two convolutional layers followed by max-pooling and dense layers
- **Regression Model**: Fully connected network for predicting housing prices, with normalization applied to inputs

**Implementation Details**

- **Optimizer**: Adam (Adaptive Moment Estimation)
- **Learning Rate**: 0.001 (default for Adam)
- **Epochs**: 50 (with early stopping based on validation loss)
- **Batch Size**: 64
- **Loss Function**:
  - **Classification**: Sparse Categorical Crossentropy
  - **Regression**: Mean Squared Error (MSE)
- **Metrics**:
  - **Classification**: Accuracy
  - **Regression**: Mean Absolute Error (MAE)

**Experimentation Code:**

Executed on Google Colab in a TPU environment. (T4)

Full code can be found here: https://github.com/gjaynir0508/DL-A1

(or) https://colab.research.google.com/drive/1Gqu0J92pbP05HYzeYrA39mlSLUHBDjbL?usp=sharing

**Snippets:**

**Functions to create models:**

```
def create_mlp(input_shape, output_units, initializer, activation="softmax"):
   inputs = Input(shape=input_shape)
   x = Flatten()(inputs)
   x = Dense(256, activation='sigmoid', kernel_initializer=initializer)(x)
   x = Dense(128, activation='tanh', kernel_initializer=initializer)(x)
   outputs = Dense(output_units, activation=activation,
                        kernel_initializer=initializer)(x)
   return Model(inputs, outputs)

def create_cnn(input_shape, output_units, initializer, activation="softmax"):
   inputs = Input(shape=input_shape)
   x = Conv2D(32, (3, 3), activation='relu',
                        kernel_initializer=initializer)(inputs)
   x = MaxPooling2D((2, 2))(x)
   x = Conv2D(64, (3, 3), activation='relu',
                        kernel_initializer=initializer)(x)
   x = MaxPooling2D((2, 2))(x)
   x = Flatten()(x)
   x = Dense(128, activation='relu', kernel_initializer=initializer)(x)
   x = Dropout(0.5)(x)
```

```
    outputs = Dense(output_units, activation=activation,
                        kernel_initializer=initializer)(x)
    return Model(inputs, outputs)

def create_regression_model(input_shape, initializer):
    inputs = Input(shape=(input_shape,))
    x = Normalization()(inputs) # Apply Normalization layer
    x = Dense(64, activation='relu', kernel_initializer=initializer)(x)
    x = Dense(32, activation='relu', kernel_initializer=initializer)(x)
    outputs = Dense(1, kernel_initializer=initializer)(x)
    return Model(inputs, outputs)
```

**Function to train models**

```
def train_model(model, dataset_name, initializer_name, model_type, epochs=50):
    (x_train, y_train), (x_test, y_test) = load_dataset(dataset_name)

    loss_fn = 'sparse_categorical_crossentropy' if model_type != 'Regression'
              else 'mse'
    metrics = ['accuracy'] if model_type != 'Regression' else ['mae']
    model.compile(optimizer='adam', loss=loss_fn, metrics=metrics)

    model_dir = f'models/{model_type}/{dataset_name}/{initializer_name}/'
    os.makedirs(model_dir, exist_ok=True)
    checkpoint_path = os.path.join(model_dir, 'best_model.keras')

    callbacks = [
        EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True),
        ModelCheckpoint(filepath=checkpoint_path, save_best_only=True,
monitor='val_loss')
    ]

    history = model.fit(x_train, y_train, epochs=epochs, batch_size=64,
validation_data=(x_test, y_test), callbacks=callbacks, verbose=0)

    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
    return history.history, test_acc, test_loss
```

**Training models for all possible combinations**

```
results = {}
for model_type in ["MLP", "CNN", "Regression"]:
    for dataset_name in ["MNIST", "Fashion-MNIST", "CIFAR-10", "CIFAR-100"] if
              model_type != "Regression" else ["California Housing"]:
        output_units = 10 if dataset_name in ["MNIST", "Fashion-MNIST", "CIFAR-10"]
                    else 100 if dataset_name == "CIFAR-100" else 1
        activation = "softmax" if model_type != "Regression" else None

        for initializer_name, initializer in initializers.items():
            print(f"Training {model_type} on {dataset_name} with
                                            {initializer_name}...")

            if model_type == "MLP":
                model = create_mlp((28, 28) if "MNIST" in dataset_name else (32, 32,
3), output_units, initializer, activation)
            elif model_type == "CNN":
```

```
            model = create_cnn((28, 28, 1) if "MNIST" in dataset_name else (32,
32, 3), output_units, initializer, activation)
        else:
            model = create_regression_model(8, initializer)

        history, test_acc, test_loss = train_model(model, dataset_name,
initializer_name, model_type)
        results[(model_type, dataset_name, initializer_name)] = {
            "history": history,
            "test_acc": test_acc,
            "test_loss": test_loss
        }
        print(f"{model_type} on {dataset_name} with {initializer_name}: Test
Accuracy = {test_acc:.4f}")
```

**Plotting**

```
def plot_training_curves(history, model_type, dataset_name,
initializer_name):
    plt.figure(figsize=(12, 5))

    # Plot loss curve
    plt.subplot(1, 2, 1)
    plt.plot(history['loss'], label='Train Loss')
    plt.plot(history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(
        f'Loss Curve - {model_type} on {dataset_name} ({initializer_name})')
    plt.legend()

    # Plot accuracy curve if available
    if 'accuracy' in history:
        plt.subplot(1, 2, 2)
        plt.plot(history['accuracy'], label='Train Accuracy')
        plt.plot(history['val_accuracy'], label='Validation Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.title(
            f'Accuracy Curve - {model_type} on {dataset_name}
({initializer_name})')
        plt.legend()

    plt.show()

for (model_type, dataset_name, initializer_name), result in results.items():
    plot_training_curves(result["history"], model_type,
                         dataset_name, initializer_name)
```

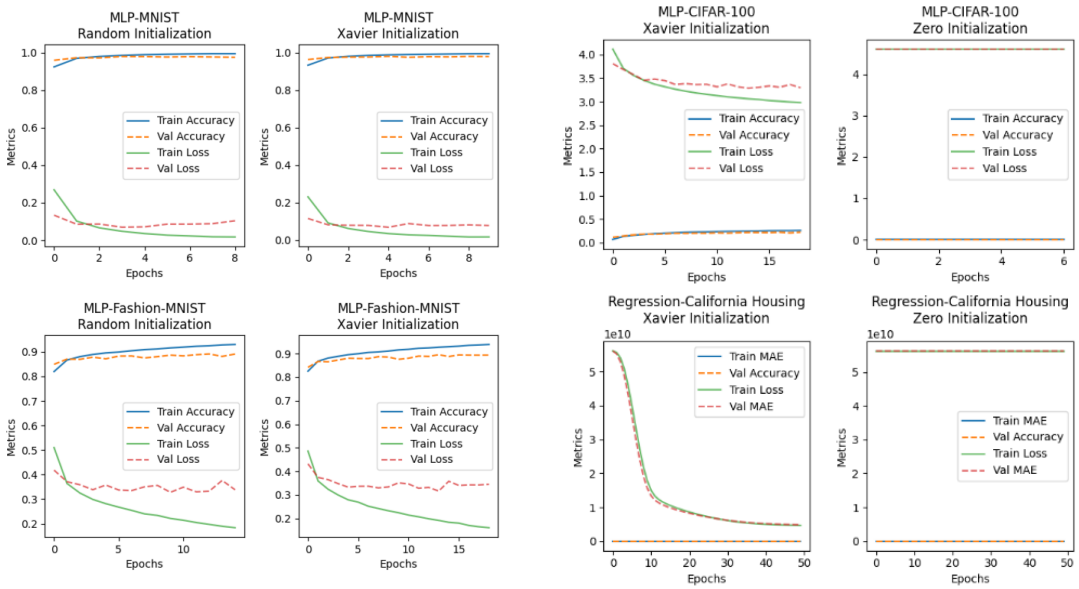
**Results:**

We have obtained the results for performance of different models on different datasets for 2 different tasks for each initialization technique. They are expressed in the following table of results and the image of a plot of training and validation – loss and accuracy curves.

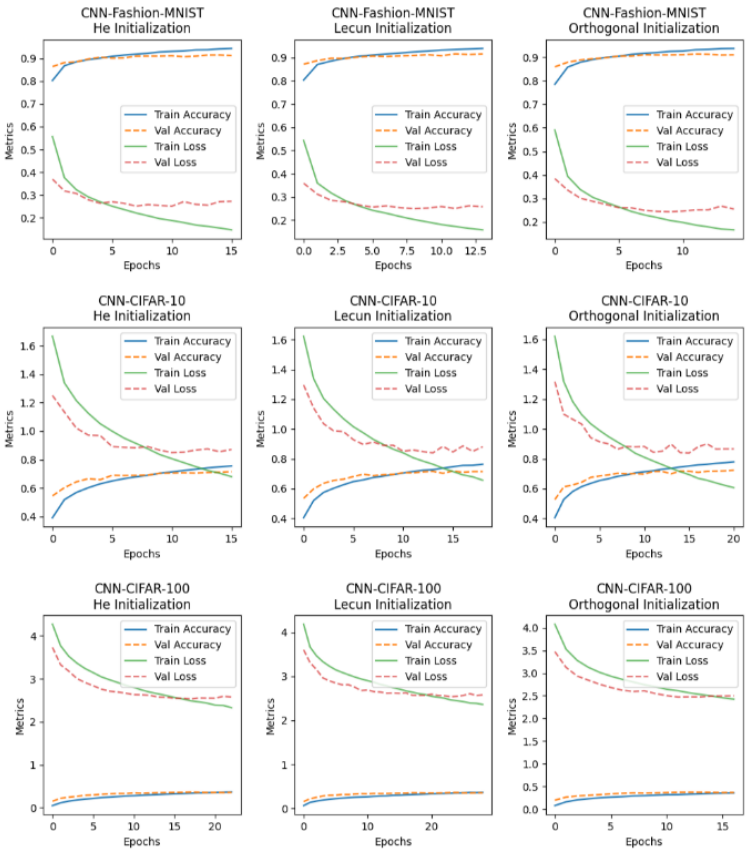
| | Model | Dataset | Initializer | Val Acc | Test Acc | Test Loss | Epochs |
|---|---|---|---|---|---|---|---|
| 0 | MLP | MNIST | Zero Initialization | 0.1135 | 0.1135 | 2.301 | 23 |
| 1 | MLP | MNIST | Random Initialization | 0.9789 | 0.9789 | 0.0699 | 9 |
| 2 | MLP | MNIST | Xavier Initialization | 0.981 | 0.981 | 0.0675 | 10 |
| 3 | MLP | MNIST | He Initialization | 0.9796 | 0.9787 | 0.0666 | 9 |
| 4 | MLP | MNIST | Lecun Initialization | 0.9794 | 0.9794 | 0.0717 | 11 |
| 5 | MLP | MNIST | Orthogonal Initialization | 0.979 | 0.9784 | 0.073 | 10 |
| 6 | MLP | Fashion-MNIST | Zero Initialization | 0.1 | 0.1 | 2.3026 | 6 |
| 7 | MLP | Fashion-MNIST | Random Initialization | 0.8911 | 0.8863 | 0.3278 | 15 |
| 8 | MLP | Fashion-MNIST | Xavier Initialization | 0.8952 | 0.8952 | 0.3165 | 19 |
| 9 | MLP | Fashion-MNIST | He Initialization | 0.8919 | 0.8919 | 0.3212 | 16 |
| 10 | MLP | Fashion-MNIST | Lecun Initialization | 0.8917 | 0.889 | 0.3195 | 15 |
| 11 | MLP | Fashion-MNIST | Orthogonal Initialization | 0.8903 | 0.8887 | 0.3164 | 14 |
| 12 | MLP | CIFAR-10 | Zero Initialization | 0.1 | 0.1 | 2.3026 | 7 |
| 13 | MLP | CIFAR-10 | Random Initialization | 0.5124 | 0.5097 | 1.3954 | 20 |
| 14 | MLP | CIFAR-10 | Xavier Initialization | 0.5035 | 0.4969 | 1.4163 | 18 |
| 15 | MLP | CIFAR-10 | He Initialization | 0.5051 | 0.5051 | 1.3942 | 23 |
| 16 | MLP | CIFAR-10 | Lecun Initialization | 0.5114 | 0.5089 | 1.4026 | 28 |
| 17 | MLP | CIFAR-10 | Orthogonal Initialization | 0.5056 | 0.5041 | 1.3995 | 20 |
| 18 | MLP | CIFAR-100 | Zero Initialization | 0.01 | 0.01 | 4.6052 | 7 |
| 19 | MLP | CIFAR-100 | Random Initialization | 0.2291 | 0.225 | 3.2791 | 18 |
| 20 | MLP | CIFAR-100 | Xavier Initialization | 0.2266 | 0.2215 | 3.2848 | 19 |
| 21 | MLP | CIFAR-100 | He Initialization | 0.224 | 0.224 | 3.2947 | 22 |
| 22 | MLP | CIFAR-100 | Lecun Initialization | 0.22 | 0.2172 | 3.3064 | 18 |
| 23 | MLP | CIFAR-100 | Orthogonal Initialization | 0.2217 | 0.2152 | 3.3005 | 19 |
| 24 | CNN | MNIST | Zero Initialization | 0.1135 | 0.1135 | 2.301 | 6 |
| 25 | CNN | MNIST | Random Initialization | 0.9929 | 0.9928 | 0.0221 | 13 |
| 26 | CNN | MNIST | Xavier Initialization | 0.9943 | 0.9943 | 0.0208 | 20 |
| 27 | CNN | MNIST | He Initialization | 0.9934 | 0.9934 | 0.0236 | 14 |
| 28 | CNN | MNIST | Lecun Initialization | 0.9935 | 0.9931 | 0.0214 | 12 |
| 29 | CNN | MNIST | Orthogonal Initialization | 0.9935 | 0.9931 | 0.0225 | 16 |
| 30 | CNN | Fashion-MNIST | Zero Initialization | 0.1 | 0.1 | 2.3026 | 6 |
| 31 | CNN | Fashion-MNIST | Random Initialization | 0.9168 | 0.9104 | 0.243 | 16 |
| 32 | CNN | Fashion-MNIST | Xavier Initialization | 0.912 | 0.9094 | 0.2536 | 13 |
| 33 | CNN | Fashion-MNIST | He Initialization | 0.9149 | 0.9116 | 0.2508 | 16 |
| 34 | CNN | Fashion-MNIST | Lecun Initialization | 0.9161 | 0.9093 | 0.2498 | 14 |
| 35 | CNN | Fashion-MNIST | Orthogonal Initialization | 0.9131 | 0.9093 | 0.244 | 15 |
| 36 | CNN | CIFAR-10 | Zero Initialization | 0.1 | 0.1 | 2.3026 | 14 |
| 37 | CNN | CIFAR-10 | Random Initialization | 0.7119 | 0.7119 | 0.8482 | 22 |
| 38 | CNN | CIFAR-10 | Xavier Initialization | 0.7205 | 0.717 | 0.8268 | 20 |
| 39 | CNN | CIFAR-10 | He Initialization | 0.7158 | 0.7055 | 0.8495 | 16 |
| 40 | CNN | CIFAR-10 | Lecun Initialization | 0.718 | 0.7156 | 0.8398 | 19 |
| 41 | CNN | CIFAR-10 | Orthogonal Initialization | 0.7236 | 0.7171 | 0.8392 | 21 |
| 42 | CNN | CIFAR-100 | Zero Initialization | 0.01 | 0.01 | 4.6052 | 6 |
| 43 | CNN | CIFAR-100 | Random Initialization | 0.3714 | 0.3664 | 2.5034 | 25 |
| 44 | CNN | CIFAR-100 | Xavier Initialization | 0.3655 | 0.3654 | 2.5138 | 28 |
| 45 | CNN | CIFAR-100 | He Initialization | 0.3666 | 0.3666 | 2.5341 | 23 |
| 46 | CNN | CIFAR-100 | Lecun Initialization | 0.3598 | 0.3587 | 2.536 | 29 |
| 47 | CNN | CIFAR-100 | Orthogonal Initialization | 0.3729 | 0.3719 | 2.4717 | 17 |

| | Model | Dataset | Initializer | MAE | Test Loss | Epochs |
|---|---|---|---|---|---|---|
| 48 | Regression | California Housing | Zero Initialization | 207080.5312 | 56176668672 | 50 |
| 49 | Regression | California Housing | Random Initialization | 51472.9844 | 4966854656 | 50 |
| 50 | Regression | California Housing | Xavier Initialization | 50577.8125 | 4844045312 | 50 |
| 51 | Regression | California Housing | He Initialization | 51037.6016 | 4911141376 | 50 |
| 52 | Regression | California Housing | Lecun Initialization | 51841.793 | 5041774592 | 50 |
| 53 | Regression | California Housing | Orthogonal Initialization | 50863.5586 | 4884531712 | 50 |

Sample plots:




All the plots can be found in the GitHub Repository (https://github.com/gjaynir0508/DL-A1/blob/main/results_plot.png)



## 6. Discussion & Insights

**Key Takeaways from the Comparisons**

- **Zero Initialization** generally leads to poor performance as it causes neurons in each layer to learn the same features, preventing effective learning.
- **Random Initialization** helps break symmetry and allows models to learn different representations, though it does not guarantee controlled variance.
- **Xavier (Glorot) Initialization** is effective for models using sigmoid or tanh activation functions as it ensures proper signal propagation without vanishing or exploding gradients.
- **He Initialization** is optimized for ReLU-based networks, allowing better variance control and faster convergence in deep architectures.
- **Lecun Initialization** is beneficial for networks using activation functions like tanh, ensuring stable learning dynamics.
- **Uniform and Normal Distributions** are often used for random weight initialization, but their performance varies depending on the network depth and activation functions.
- **Orthogonal Initialization** helps maintain variance in deep networks, reducing gradient-related issues.
- **Sparse Initialization** sets most weights to zero, which can be useful in certain architectures like deep sparse networks.

**Why Does a Certain Technique/Model Perform Better?**

- **Proper initialization prevents gradient-related issues** such as vanishing or exploding gradients, which are critical for training deep networks efficiently.
- **Xavier, He, and Lecun Initializations** provide structured variance control, ensuring stable signal propagation and better generalization.
- **Random and Uniform Initializations work but lack structure**, leading to fluctuations in performance across different models.
- **Zero Initialization should be avoided** as it leads to symmetry problems and prevents learning.
- **Orthogonal Initialization can be beneficial in deep networks** as it maintains stability during training.
- **Sparse Initialization is useful in specialized models**, particularly when network sparsity is desired, e.g.: Neural Networks for Edge Devices, Graph Neural Networks etc.,

**Limitations and Possible Improvements**

- **Impact on different architectures:** Initializations should be tested across various architectures such as CNNs, Transformers, and RNNs to understand their broader implications.
- **Effect of learning rate tuning:** The performance of initialization techniques can be further optimized by adjusting the learning rate dynamically.
- **Combination with normalization techniques:** Batch normalization or layer normalization could enhance the effectiveness of initialization methods.
- **Impact of optimizers:** Different optimizers (SGD, Adam, RMSProp) interact with initialization in unique ways, influencing model convergence and stability.

## 7. Conclusion

**Final Summary of Findings**

- **Effective weight initialization is crucial** for stable training, faster convergence, and better generalization across machine learning models.
- **Xavier, He, and Lecun Initializations** are widely used due to their ability to prevent gradient-related issues and maintain stable learning.
- **Random and Uniform Initializations** can work but are less structured, while **Zero Initialization is ineffective** due to symmetry breaking problems.
- **Orthogonal and Sparse Initializations have niche use cases**, particularly in deep networks and specialized models.

**Best Technique/Model Based on Results**

- **There is no one-size-fits-all approach**; the best initialization depends on the architecture, activation functions, and problem domain.
- **Xavier Initialization is suitable for sigmoid/tanh activations**, while **He Initialization works best with ReLU-based models.**
- **Orthogonal Initialization may be beneficial for very deep networks**, while **Sparse Initialization is useful for models requiring structured sparsity.**

**Future Work and Scope for Improvements**

- Expand studies to include **modern deep learning architectures** such as CNNs, Transformers, and RNNs.
- Investigate how **different activation functions** (ReLU, GELU, Swish) interact with various initialization strategies.
- Explore the **impact of weight initialization on training efficiency and generalization** across large-scale datasets.
- Analyze the **combined effect of initialization, normalization, and optimization techniques** to find the best synergy for deep learning models.

These insights can help optimize neural network training strategies and enhance model performance across diverse applications.