Observe results before and after applying Transfer Learning.

Transfer learning is a research problem in machine learning & deep learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks.So, in transfer learning your previous learning helps you to understand the new concept or learning. In transfer learning we use pre-trained model & we make some modification on that to make a new model.

So, here we will create a hand writing classifier by MNIST data then we will modify this model to predict a given number is odd or even by the help of transfer learning. Then we will compare them not using transfer learning.

Now first creating a Model which can classify MNIST handwriting:

```
In [ ]: # Importing necessary libraries
        import os
        import tensorflow as tf
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        import time
        plt.style.use("fivethirtyeight")
        %load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

```
In [ ]: # Loading the data of MNIST handwritten
        (X_train_full, y_train_full), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
        X_train_full = X_train_full / 255.0
        X_test = X_test / 255.0
        X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
        y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

In [ ]:
```
# Creating layer of model
tf.random.set_seed(42)  #For getting similar output (optional)
np.random.seed(42)  #For getting similar output (optional)

LAYERS = [ tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(100, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(10, activation="softmax")]


model = tf.keras.models.Sequential(LAYERS)
```

In [ ]:
```
# Compiling the model
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.SGD(learning_rate=1e-3),
              metrics=["accuracy"])
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
flatten_2 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_7 (Dense)             (None, 300)               235500
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)   (None, 300)               0
_________________________________________________________________
dense_8 (Dense)             (None, 100)               30100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)   (None, 100)               0
_________________________________________________________________
dense_9 (Dense)             (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

```
In [ ]: # Lets train the model
        history = model.fit(X_train, y_train, epochs=10,
                            validation_data=(X_valid, y_valid), verbose=2)
```

```
Epoch 1/10
1719/1719 - 3s - loss: 1.5275 - accuracy: 0.5970 - val_loss: 0.9444 - val_accuracy: 0.7
980
Epoch 2/10
1719/1719 - 3s - loss: 0.7465 - accuracy: 0.8287 - val_loss: 0.5868 - val_accuracy: 0.8
596
Epoch 3/10
1719/1719 - 3s - loss: 0.5412 - accuracy: 0.8624 - val_loss: 0.4685 - val_accuracy: 0.8
834
Epoch 4/10
1719/1719 - 3s - loss: 0.4591 - accuracy: 0.8771 - val_loss: 0.4104 - val_accuracy: 0.8
940
Epoch 5/10
1719/1719 - 3s - loss: 0.4142 - accuracy: 0.8869 - val_loss: 0.3758 - val_accuracy: 0.9
006
Epoch 6/10
1719/1719 - 3s - loss: 0.3852 - accuracy: 0.8938 - val_loss: 0.3525 - val_accuracy: 0.9
052
Epoch 7/10
1719/1719 - 3s - loss: 0.3644 - accuracy: 0.8980 - val_loss: 0.3348 - val_accuracy: 0.9
102
Epoch 8/10
1719/1719 - 3s - loss: 0.3485 - accuracy: 0.9021 - val_loss: 0.3209 - val_accuracy: 0.9
138
Epoch 9/10
1719/1719 - 3s - loss: 0.3356 - accuracy: 0.9053 - val_loss: 0.3111 - val_accuracy: 0.9
152
Epoch 10/10
1719/1719 - 3s - loss: 0.3251 - accuracy: 0.9077 - val_loss: 0.3016 - val_accuracy: 0.9
170
```

```
In [ ]: # Saving the model
        model.save("pretrained_mnist_model.h5")
```

Now Lets create a model which can predict a given number is odd or even without having Transfer learning technique.

```
In [ ]: # Making the label as an even or odd category from numbers where even is 1 and odd is 0

        def update_even_odd_labels(labels):
          for idx, label in enumerate(labels):
            labels[idx] = np.where(label % 2 == 0, 1, 0)
          return labels
```

```
In [ ]: y_train_bin, y_test_bin, y_valid_bin = update_even_odd_labels([y_train, y_test, y_valid]
```

In [ ]:
```
# Creating layer of model
tf.random.set_seed(42)  #For getting similar output (optional)
np.random.seed(42)  #For getting similar output (optional)

LAYERS = [ tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(100, kernel_initializer="he_normal"),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Dense(2, activation="softmax")]  # Here I have just used 2 output la


model_1 = tf.keras.models.Sequential(LAYERS)
```

In [ ]:
```
model_1.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_3 (Flatten)          (None, 784)               0         
_________________________________________________________________
dense_10 (Dense)             (None, 300)               235500    
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 300)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 100)               30100     
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 100)               0         
_________________________________________________________________
dense_12 (Dense)             (None, 2)                 202       
=================================================================
Total params: 265,802
Trainable params: 265,802
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# Compiling new model
model_1.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/optimizer_v2/optimizer_v
2.py:375: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instead.")
```

In [ ]:
```python
# now training & calculating the training time.

# starting time
start = time.time()

history = model_1.fit(X_train, y_train_bin, epochs=10,
                    validation_data=(X_valid, y_valid_bin), verbose=2)

#ending time
end = time.time()

# total time taken
print(f"Runtime of the program is {end - start}")
```

```
Epoch 1/10
1719/1719 - 3s - loss: 0.4357 - accuracy: 0.8088 - val_loss: 0.3279 - val_accuracy: 0.8
664
Epoch 2/10
1719/1719 - 3s - loss: 0.3124 - accuracy: 0.8699 - val_loss: 0.2763 - val_accuracy: 0.8
900
Epoch 3/10
1719/1719 - 3s - loss: 0.2767 - accuracy: 0.8885 - val_loss: 0.2474 - val_accuracy: 0.9
046
Epoch 4/10
1719/1719 - 3s - loss: 0.2530 - accuracy: 0.9005 - val_loss: 0.2262 - val_accuracy: 0.9
160
Epoch 5/10
1719/1719 - 3s - loss: 0.2333 - accuracy: 0.9101 - val_loss: 0.2083 - val_accuracy: 0.9
228
Epoch 6/10
1719/1719 - 3s - loss: 0.2167 - accuracy: 0.9183 - val_loss: 0.1927 - val_accuracy: 0.9
300
Epoch 7/10
1719/1719 - 3s - loss: 0.2018 - accuracy: 0.9252 - val_loss: 0.1795 - val_accuracy: 0.9
364
Epoch 8/10
1719/1719 - 3s - loss: 0.1888 - accuracy: 0.9319 - val_loss: 0.1676 - val_accuracy: 0.9
434
Epoch 9/10
1719/1719 - 3s - loss: 0.1774 - accuracy: 0.9375 - val_loss: 0.1581 - val_accuracy: 0.9
458
Epoch 10/10
1719/1719 - 3s - loss: 0.1675 - accuracy: 0.9409 - val_loss: 0.1494 - val_accuracy: 0.9
506
Runtime of the program is 41.243441104888916
```

Conclusion:

Runtime of the program is 41.24 sec & val_accuracy: 0.9506

Now Let's create the same model which can predict a given number is odd or even with having Transfer learning technique.

In [ ]:
```
# Loading pre-trained model
pretrained_mnist_model = tf.keras.models.load_model("pretrained_mnist_model.h5")
```

In [ ]:
```
pretrained_mnist_model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_7 (Dense)              (None, 300)               235500
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 300)               0
_________________________________________________________________
dense_8 (Dense)              (None, 100)               30100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# Checking layers are trainable or not
for layer in pretrained_mnist_model.layers:
  print(f"{layer.name}: {layer.trainable}")
```

```
flatten_2: True
dense_7: True
leaky_re_lu_4: True
dense_8: True
leaky_re_lu_5: True
dense_9: True
```

In [ ]:
```
# Lets make them false or non trainable except last one
for layer in pretrained_mnist_model.layers[:-1]:
  layer.trainable = False
  print(f"{layer.name}: {layer.trainable}")
```

```
flatten_2: False
dense_7: False
leaky_re_lu_4: False
dense_8: False
leaky_re_lu_5: False
```

In [ ]:
```python
for layer in pretrained_mnist_model.layers:
  print(f"{layer.name}: {layer.trainable}")
```

```
flatten_2: False
dense_7: False
leaky_re_lu_4: False
dense_8: False
leaky_re_lu_5: False
dense_9: True
```

In [ ]:
```python
# Now make a model using that one
lower_pretrained_layers = pretrained_mnist_model.layers[:-1]

new_model = tf.keras.models.Sequential(lower_pretrained_layers)
new_model.add(
    tf.keras.layers.Dense(2, activation="softmax")
)
```

In [ ]:
```python
new_model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_7 (Dense)              (None, 300)               235500
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 300)               0
_________________________________________________________________
dense_8 (Dense)              (None, 100)               30100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_13 (Dense)             (None, 2)                 202
=================================================================
Total params: 265,802
Trainable params: 202
Non-trainable params: 265,600
_________________________________________________________________
```

In [ ]:
```python
# Making the label as an even or odd category from numbers where even is 1 and odd is 0

def update_even_odd_labels(labels):
  for idx, label in enumerate(labels):
    labels[idx] = np.where(label % 2 == 0, 1, 0)
  return labels
```

In [ ]:
```python
y_train_bin, y_test_bin, y_valid_bin = update_even_odd_labels([y_train, y_test, y_valid]
```

In [ ]:
```
# Compiling new model
new_model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/optimizer_v2/optimizer_v
2.py:375: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instead.")
```

In [ ]:
```
# now train and calculating the time

# starting time
start = time.time()

history = new_model.fit(X_train, y_train_bin, epochs=10,
                    validation_data=(X_valid, y_valid_bin), verbose=2)

#ending time
end = time.time()
print(f"Runtime of the program is {end - start}")
```

```
Epoch 1/10
1719/1719 - 3s - loss: 0.3898 - accuracy: 0.8288 - val_loss: 0.3247 - val_accuracy: 0.8
676
Epoch 2/10
1719/1719 - 3s - loss: 0.3300 - accuracy: 0.8602 - val_loss: 0.3049 - val_accuracy: 0.8
752
Epoch 3/10
1719/1719 - 3s - loss: 0.3163 - accuracy: 0.8660 - val_loss: 0.2948 - val_accuracy: 0.8
796
Epoch 4/10
1719/1719 - 3s - loss: 0.3083 - accuracy: 0.8701 - val_loss: 0.2884 - val_accuracy: 0.8
832
Epoch 5/10
1719/1719 - 2s - loss: 0.3023 - accuracy: 0.8725 - val_loss: 0.2834 - val_accuracy: 0.8
846
Epoch 6/10
1719/1719 - 2s - loss: 0.2978 - accuracy: 0.8752 - val_loss: 0.2792 - val_accuracy: 0.8
874
Epoch 7/10
1719/1719 - 3s - loss: 0.2939 - accuracy: 0.8772 - val_loss: 0.2758 - val_accuracy: 0.8
872
Epoch 8/10
1719/1719 - 3s - loss: 0.2907 - accuracy: 0.8788 - val_loss: 0.2728 - val_accuracy: 0.8
890
Epoch 9/10
1719/1719 - 3s - loss: 0.2876 - accuracy: 0.8797 - val_loss: 0.2708 - val_accuracy: 0.8
906
Epoch 10/10
1719/1719 - 3s - loss: 0.2851 - accuracy: 0.8817 - val_loss: 0.2678 - val_accuracy: 0.8
930
Runtime of the program is 41.2496874332428
```

Conclusion:

Runtime of the program is 41.24 sec & val_accuracy: 0.8930

Comparison:

Without Transfer learning:

- Runtime of the program is 41.24 sec
- val_accuracy: 0.9506

With Transfer Learning:

- Runtime of the program is 41.24 sec
- val_accuracy: 0.8930

Here we can we have transfer learning output is pretty close to actual accuracy, although we are just training 202 parameters. So, if we increase the epochs then the accuracy would be high. Now it is taking same time but in big problem it may take less time with compare to Without Transfer learning.

Q1:Explain the importance of weight initialization in artificial neural networks. Why is it necessary to initialize the weights carefully?

Answer: Weight initialization is crucial in artificial neural networks because it sets the initial values for the model's parameters (weights). Proper weight initialization ensures that the neural network starts with reasonable initial values, which significantly impacts its training and convergence. If the weights are initialized poorly, the model may struggle to learn effectively, leading to slow convergence or even complete failure to learn.

Q2:Describe the challenges associated with improper weight initialization. How do these issues affect model training and convergence?

Answer: Improper weight initialization can lead to several challenges during model training. If weights are initialized too large or small, the gradients during backpropagation can become vanishingly small or exploding, respectively, resulting in slow convergence or instability. Additionally, improper weight initialization may cause the model to get stuck in local minima, leading to suboptimal solutions. It can also result in slow learning or oscillating behaviors during training, making it difficult to reach a global minimum.

Q3: Discuss the concept of variance and how it relates to weight initialization. Why is it crucial to consider the variance of weights during initialization?

Answer: Variance is a measure of the spread or dispersion of data points. In the context of weight initialization, it refers to the range of values that weights can take. A very high or very low variance can cause gradients to explode or vanish during backpropagation, respectively. Controlling the variance during weight initialization is essential to ensure stable and effective learning. Properly balanced variance helps prevent training issues, allowing the model to learn more efficiently and converge faster.

Part 2: Weight Initialization Techniques

Q1: Explain the concept of zero initialization. Discuss its potential limitations and when it can be appropriate to use.

Answer: Zero initialization involves setting all weights to zero at the beginning of training. However, this approach has limitations as all neurons will have the same output, leading to symmetrical gradients during backpropagation. This means that each neuron will learn the same features, making the learning process ineffective. Zero initialization is generally not recommended for most neural network architectures.

Q2: Describe the process of random initialization. How can random initialization be adjusted to mitigate potential issues like saturation or vanishing/exploding gradients?

Answer: Random initialization sets the weights to random values, often drawn from a normal or uniform distribution. This introduces diversity among neurons, allowing them to learn different features. To mitigate saturation or vanishing/exploding gradients, it is crucial to adjust the scale of random initialization based on the activation function. For example, Xavier/Glorot initialization scales the random weights based on the number of input and output neurons for a layer, which helps balance the variance and improve convergence.

Q3: Discuss the concept of Xavier/Glorot initialization. Explain how it addresses the challenges of improper weight initialization and the underlying theory behind it.

Answer: Xavier/Glorot initialization is a weight initialization technique that sets the initial weights based on the size of the layer's input and output. The goal is to ensure that the variance of the activations and gradients remains consistent across layers. By balancing the variance, it mitigates the vanishing and exploding gradient problems, promoting stable training and faster convergence. The initialization formula takes into account the number of input and output neurons and follows a specific distribution based on the activation function.

Q4: Explain the concept of He initialization. How does it differ from Xavier initialization, and when is it preferred?

Answer: He initialization is similar to Xavier/Glorot initialization, but it scales the weights based only on the number of input neurons. It is specifically designed for activation functions like ReLU and its variants, which introduce non-linearity and may cause the vanishing gradient problem. He initialization allows ReLU-based activations to maintain a balanced variance during training, making it a preferred choice when using ReLU or similar activation functions.

In [1]:
```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.initializers import Zeros, RandomNormal, GlorotUniform, HeNormal

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(-1, 784) / 255.0
x_test = x_test.reshape(-1, 784) / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Define the neural network architecture
def create_model(initializer):
    model = Sequential()
    model.add(Dense(256, input_shape=(784,), kernel_initializer=initializer))
    model.add(Activation('relu'))
    model.add(Dense(128, kernel_initializer=initializer))
    model.add(Activation('relu'))
    model.add(Dense(10, kernel_initializer=initializer))
    model.add(Activation('softmax'))
    return model

# Define the weight initialization techniques
zero_initializer = Zeros()
random_initializer = RandomNormal(mean=0, stddev=0.01)
xavier_initializer = GlorotUniform()
he_initializer = HeNormal()

# Train and evaluate the models with different initializers
initializers = [zero_initializer, random_initializer, xavier_initializer, he_initializer
for initializer in initializers:
    model = create_model(initializer)
    model.compile(loss='categorical_crossentropy', optimizer=SGD(learning_rate=0.01), me
    history = model.fit(x_train, y_train, batch_size=32, epochs=10, validation_split=0.1
    test_loss, test_accuracy = model.evaluate(x_test, y_test)
    print(f"\nModel with {initializer} initialization:")
    print(f"Test Loss: {test_loss}, Test Accuracy: {test_accuracy}")
```

```
Epoch 2/10
1688/1688 [==============================] - 8s 5ms/step - loss: 0.2773 - accuracy: 0.9202 - val_loss: 0.2001 - val_accuracy: 0.9427
Epoch 3/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.2253 - accuracy: 0.9356 - val_loss: 0.1750 - val_accuracy: 0.9505
Epoch 4/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.1925 - accuracy: 0.9451 - val_loss: 0.1538 - val_accuracy: 0.9578
Epoch 5/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.1675 - accuracy: 0.9515 - val_loss: 0.1349 - val_accuracy: 0.9628
Epoch 6/10
1688/1688 [==============================] - 8s 5ms/step - loss: 0.1490 - accuracy: 0.9570 - val_loss: 0.1232 - val_accuracy: 0.9662
Epoch 7/10
1688/1688 [==============================] - 7s 4ms/step - loss: 0.1335 - accuracy: 0.9621 - val_loss: 0.1137 - val_accuracy: 0.9693
Epoch 8/10
```

So , HE Normalization Perform better because we use Relu activation function

In this code, we define a neural network with three layers (two hidden layers and one output layer) and use different initializers for each layer. We then train the model with each initializer and evaluate their performance.

Considerations and tradeoffs when choosing weight initialization techniques:

- Activation Function: Different weight initializers work better with specific activation functions. For example, He initialization is suitable for ReLU-based activations, while Xavier initialization works well with tanh or sigmoid activations.
- Layer Size: The number of neurons in each layer can impact the choice of weight initialization. Smaller layers might work well with zero or random initialization, while larger layers might benefit from techniques like Xavier or He initialization.
- Convergence Speed: Proper weight initialization can lead to faster convergence during training. Techniques like Xavier and He initialization are known to speed up convergence.
- Avoiding Vanishing/Exploding Gradients: Weight initialization can help prevent vanishing or exploding gradients, which can affect the stability of training.
- Model Performance: It's essential to evaluate the model's performance on the validation set and test set to choose the best weight initialization technique for a specific task.
- Experimentation: Experimenting with different weight initialization techniques is crucial to find the one that works best for a given neural network architecture and task.

```
In [ ]:
```