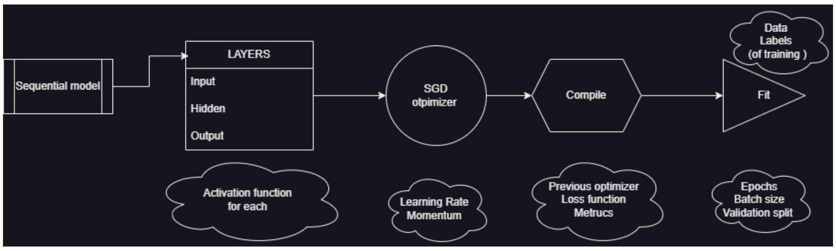
LINK : https://github.com/kos00pas/ECE661.git # To RUN - Uncomment the function to the corresponding exercise you want to run

```python
if __name__ == "__main__":
    # Load the data once using DataLoader
    data_loader = DataLoader()
    X_train, X_test, y_train, y_test =
data_loader.get_data()
    """# Method: each exercise will create its own DNNs"""

    # exercise_1_a(X_train, X_test, y_train, y_test)
    # exercise_1_b(X_train, X_test, y_train, y_test)
    # exercise_1_c(X_train, X_test, y_train, y_test)
    # exercise_1_d(X_train, X_test, y_train, y_test)
    # exercise_1_e(X_train, X_test, y_train, y_test)
    # exercise_2(X_train, X_test, y_train, y_test)
```

Neural Network Workflow



```python
# ############# NeuralNetworkExperiment Class #############
class NeuralNetworkExperiment:
    def __init__(self, X_train, X_test, y_train, y_test, learning_rate):
        # Initialize with prepared data and learning rate
        self.X_train = X_train ; self.X_test = X_test ; self.y_train = y_train ; self.y_test = y_test ; self.learning_rate =
learning_rate ; self.model = None

    # ############# Section 2: Build the Model #############
    def build_model(self, momentum=None):
        # Initialize the Sequential model
        self.model = models.Sequential()

        # Add the Input layer explicitly
        self.model.add(layers.Input(shape=(self.X_train.shape[1],)))

        # Add a 16-neuron hidden dense layer with the tanh activation function
        self.model.add(layers.Dense(16, activation='tanh'))

        # Add a 3-neuron output dense layer with the softmax activation function
        self.model.add(layers.Dense(3, activation='softmax'))

        # Compile the model with SGD optimizer
        optimizer = optimizers.SGD(learning_rate=self.learning_rate,
                                   momentum=momentum) if momentum is not None else optimizers.SGD(
                                    learning_rate=self.learning_rate)

        self.model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

    # ############# Section 3: Train the Model #############
    def train_model(self):
        # Train the model with the specified parameters
        history = self.model.fit(self.X_train, self.y_train, epochs=40, batch_size=32, validation_split=0.1)
        """
        The model is trained using the fit method with:
        - epochs=40 -> Number of times the training data will be passed through the model.
        - batch_size=32 -> Number of samples per gradient update.
        - validation_split=0.1 -> Reserve 10% of training data for validation to check how well the model generalizes.
        """
        return history.history['accuracy']  # Return training accuracy for each epoch
```

1.a

```python
"""########################################################################"""
    for lr in learning_rates:
        # Loop through each learning rate and run the experiment 20 times
        print(f"Running experiments for learning rate: {lr}")
        accuracies = []

        for run in range(runs_per_learning_rate):
            print(f"Run {run + 1} for learning rate {lr}")
            """#############"""
            experiment = NeuralNetworkExperiment(X_train, X_test, y_train, y_test, learning_rate=lr)
            experiment.build_model()
            accuracy = experiment.train_model()
            accuracies.append(accuracy)
            """#############"""

        # Compute the average accuracy over the 20 runs
        avg_accuracy = np.mean(accuracies, axis=0)
        avg_accuracy_per_lr[lr] = avg_accuracy
    """########################################################################"""
```

- a. **Best performing learning rate**: - 0.5, as it achieves the best accuracy quickly with fewer epochs. - b. **Learning rate explanation**: - Determines the step size for adjusting weights. - Higher rates lead to faster accuracy but risk overshooting and instability. - Lower rates ensure stability but slower convergence. - A rate of 0.2 avoids fluctuations while maintaining efficiency.

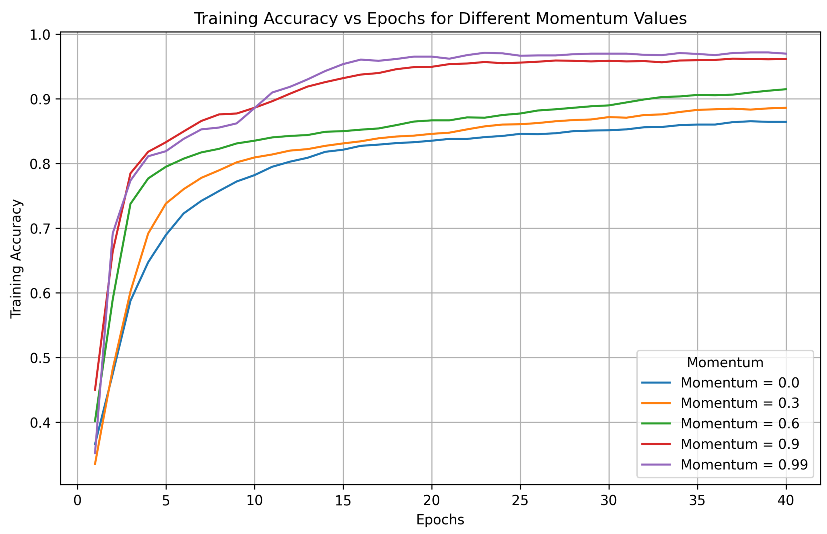
1.b

```python
for momentum in momentum_values:
    print(f"Running experiments for momentum: {momentum}")
    accuracies = []

    for run in range(runs_per_momentum):
        print(f"Run {run + 1} for momentum {momentum}")
        """#############"""
        experiment = NeuralNetworkExperiment(X_train, X_test, y_train, y_test, learning_rate)
        experiment.build_model(momentum=momentum)  # Modify build_model to accept momentum
        accuracy = experiment.train_model()
        accuracies.append(accuracy)
        """#############"""

    # Compute the average accuracy over the 20 runs
    avg_accuracy = np.mean(accuracies, axis=0)
    avg_accuracy_per_momentum[momentum] = avg_accuracy
```

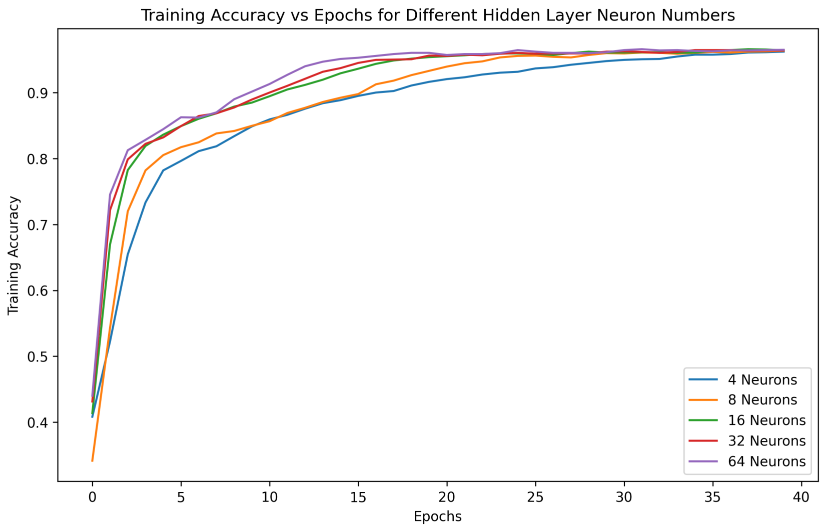
- a. **Best performing momentum:** - 0.9, as it reduces oscillations and accelerates convergence effectively(better than 0.99), leading to smoother updates. - b. **What is momentum and how does it affect the training process?**: - Momentum is a technique used in Stochastic Gradient Descent (SGD) to accelerate the optimization process by incorporating a fraction of the previous gradients into the current update. - This helps smooth out fluctuations in the gradient updates and reduces oscillations, allowing the model to converge faster and more steadily toward the optimal solution. - Without momentum (momentum = 0), the optimization behaves like standard SGD, where the model updates weights based only on the current gradient. - This can introduce noise and cause oscillations, especially when using mini-batches, as the gradients are calculated from random subsets of data. - This randomness can make the optimization slower and unstable. - With momentum, a fraction of the previous weight updates is carried forward into the current update. - This allows the optimizer to “gain speed” in the right direction (where the gradients are more significant) while decelerating movement in less important directions. - Higher momentum values like 0.9 or 0.99 help smooth the trajectory of updates by averaging the gradient updates over time, which reduces oscillations and helps escape local minima. - As a result, the model converges more quickly and smoothly.



1.c

```python
for neurons in neuron_list:
        accuracies = []
        for i in range(20):
            """#############"""
            """Build a New NN"""
            model = models.Sequential()
            model.add(layers.Input(shape=(X_train.shape[1],)))
            model.add(layers.Dense(neurons, activation='tanh')) # Neurons
            model.add(layers.Dense(y_train.shape[1], activation='softmax'))
            """#############"""
            optimizer = optimizers.SGD(learning_rate=0.02, momentum=0.9)
            model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
            """#############"""
            history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.1, verbose=0)
            accuracies.append(history.history['accuracy'])
            """#############"""
```

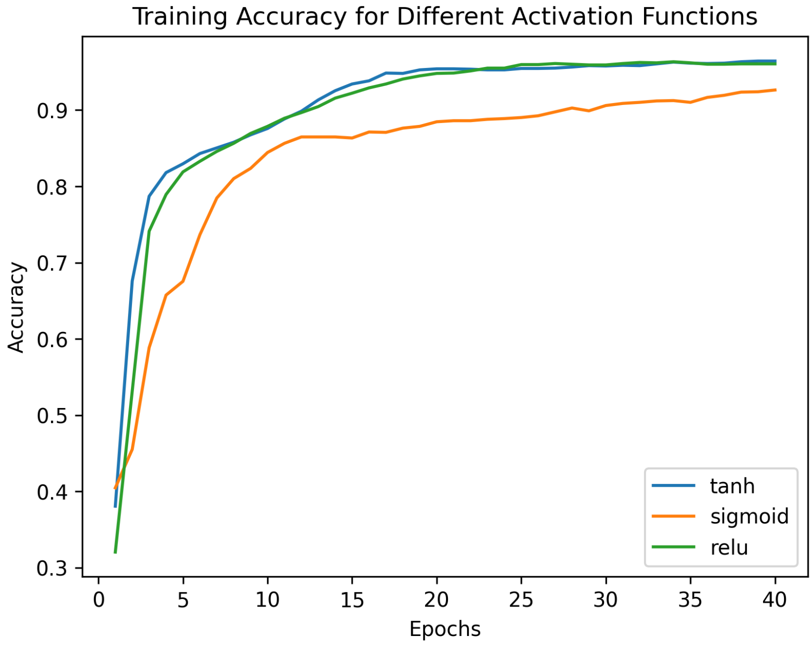
- a. **Best performing hidden layer neuron number**: - Based on the plot, 64 neurons perform the best as they achieve the highest accuracy after more epochs. - However, 16 and 32 neurons perform well at lower epochs (around 10-15), reaching good accuracy quickly. - If the goal is to minimize training time, fewer neurons such as 16 or 32 might be more appropriate, while 64 neurons achieve the best long-term performance. - If the goal is to minimize the risk of overfitting, it would be better to choose fewer neurons, such as 16 or 32 - b. **Neurons explanation**: - More neurons (e.g., 64) allow the model to capture more complex patterns, leading to higher accuracy, but may risk overfitting as the model becomes more complex. - Fewer neurons (e.g., 16 or 32) generalize better at lower epochs, reducing the risk of overfitting but may not capture as complex patterns as efficiently.



1.d

```
"""######################################################################"""
    for activation in activations:
        for _ in range(20):
            """#############"""
            model = Sequential()
            model.add(Dense(n_hidden_neurons, input_dim=X_train.shape[1], activation=activation))
            model.add(Dense(n_classes, activation='softmax'))
            """#############"""
            optimizer = SGD(learning_rate=learning_rate, momentum=momentum)
            model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
            """#############"""
            history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=0, validation_split=0.1)
            results[activation] += np.array(history.history['accuracy'])
        """#############"""
        results[activation] /= 20
    """######################################################################"""
```

- a. **Best performing activation function**: - ReLU, as it is computationally cheaper and avoids the vanishing gradient problem while 'dying ReLU' is less harmful. - b. **Activation function explanation**: - Activation function: It introduces non-linearity to help the neural network learn complex patterns. - Tanh & Sigmoid: Both squash the input into a specific range (Tanh: -1 to 1, Sigmoid: 0 to 1), but can lead to vanishing gradients, slowing down learning. - ReLU: It outputs the input directly if positive, otherwise returns zero, avoiding the vanishing gradient problem but can suffer from "dying ReLU," where neurons stop updating. - All the above are Non-linear activation functions that can help the network learn complex patterns.

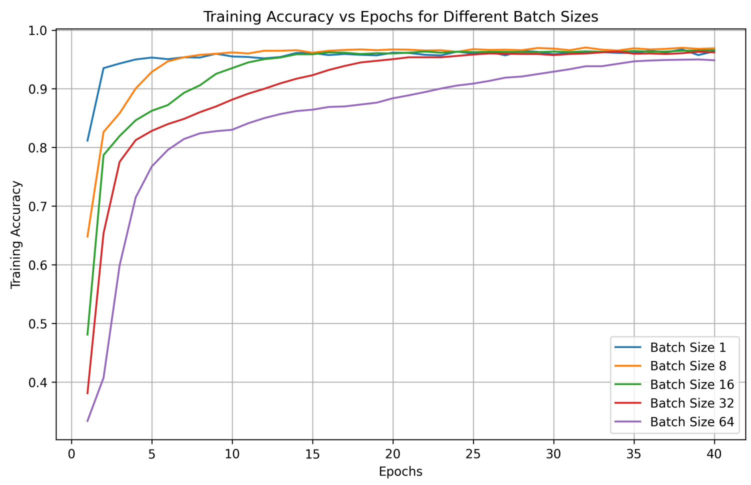


1.e

```
"""######################################################################"""
    for batch_size in batch_sizes:
        print(f"Training with batch size: {batch_size}")
        accuracies = []
        for _ in range(20):
            """#############"""
            model = Sequential()
            model.add(Dense(hidden_neurons, activation=activation_function, input_shape=(X_train.shape[1],)))
            model.add(Dense(3, activation='softmax'))  # 3 output classes for Iris dataset
            """#############"""
            optimizer = SGD(learning_rate=learning_rate, momentum=momentum)
            model.compile(optimizer=optimizer,
                          loss=CategoricalCrossentropy(),
                          metrics=['accuracy'])
            """#############"""
            history = model.fit(X_train, y_train,
                                epochs=epochs,
                                batch_size=batch_size,
                                validation_split=0.1,
                                verbose=0)
            """#############"""
            accuracies.append(history.history['accuracy'])
        """######################################################################"""
```

- a. **Best performing batch size**: - Based on the performance at lower epochs, smaller batch sizes (like 8 or 16) perform better, as they allow for quicker adaptation. - However, larger batch sizes (like 64) eventually lead to more stable and higher accuracy, especially over longer training epochs. - If the goal is fast adaptation early in training, smaller batch sizes are better. - However, if the goal is overall stability and generalization, batch size 64 would provide the best performance over the entire training process.

- b. **Batch size explanation**:
  - Batch size determines how many samples are processed before weight updates.
  - Smaller sizes provide quicker adaptation but with more instability. Larger sizes ensure stable and gradual updates.



2.

```python
"""#########################################################################"""
    for lr in learning_rates:
        for neurons in neurons_list:
            for batch_size in batch_sizes:
                print(f"Running model with lr={lr}, neurons={neurons}, batch_size={batch_size}")
                """#############"""
                model = Sequential()
                model.add(Dense(neurons, input_shape=(X_train.shape[1],), activation='relu'))
                model.add(Dense(3, activation='softmax'))  # 3 output classes for Iris dataset
                """#############"""
                optimizer = SGD(learning_rate=lr, momentum=0.9)
                model.compile(optimizer=optimizer,
                              loss=CategoricalCrossentropy(),
                              metrics=['accuracy'])
                """#############"""
                history = model.fit(X_train, y_train,
                                    epochs=epochs,
                                    batch_size=batch_size,
                                    validation_split=0.1,
                                    verbose=0)
                """#########################################################################"""
```

- activation function : ReLU - Avoid vanishing gradient problem in better than avoid dying relu - Computationally simpler , promotes sparsity

- Momentum : 0.9
  - A good balance between smoothing the updates and reducing oscillations, while avoiding excessive inertia that might come with higher momentum values like 0.99
- Learning rate : 0.2
  - Less instability risks associated with the higher rate of 0.5.
- hidden layer neurons: 32
  - More neurons allow the model to capture complex patterns in the data.
  - Select 32 over 64 to reduce the risk of Overfitting.
- batch size: 64
  - table and gradual updates
  - Larger batch sizes tend to generalize

Tricky part :

- Learning Rate: 0.2 instead 0.5
  - stability , less overshooting risk
- Neurons: 32 instead 64 -
  - more patters less overfitting risk
- Batch size: 64 instead 32 -
  - stable and gradual updates

To be sure about the [Learning Rate, Neurons, Batch size] I prepare the following experiment: * Keep constant ReLu, Momentum and changing the above (total 8 combinations).