CS478: Introduction to Deep Learning
Assignment 2 (10 points)

This assignment focuses on understanding and addressing the vanishing gradient problem in deep neural networks. You will explore techniques to improve training, prevent overfitting, and analyze the effectiveness of these strategies.

Dataset

- Use the Spiral Dataset provided in the starter code. This dataset is non-linear and highlights the challenges associated with training deep neural networks.

Tasks

1. Vanishing/Exploding Gradients:
    - Build a simple deep feedforward neural network with 5 hidden layers (each with 64 neurons) using sigmoid activation.
    - Train the model and observe its performance.
    - Analyze the impact of the vanishing gradient problem on training and test accuracy.
2. Improving Training:
    - Implement strategies to address the vanishing gradient problem and improve training:
        - Replace sigmoid with ReLU or other activation functions.
        - Add Batch Normalization to the network.
        - Incorporate Dropout to prevent overfitting.
    - Train and test the model after applying each strategy.
    - For each strategy, document the results and your observations about model performance

3. Summarize Findings: Write a comprehensive summary comparing:
    - The effects of activation functions on the vanishing gradient problem.
    - The benefits of Batch Normalization.
    - The impact of Dropout on generalization.
    - Any additional observations from your experiments.

Submission Requirements

1. Code:
    - Submit well-commented Python code implementing all tasks.
    - Use Keras or TensorFlow for building and training the models.
2. Report:
    - Include a report summarizing your findings for each experiment.
    - Present clear comparisons (e.g., through tables or graphs) to highlight the impact of different strategies.

Colab Link:

https://colab.research.google.com/drive/11ulaKwpheoK2x1xhI3H7YhpLWF6103ZR?usp=sharing

GitHub Link:

https://github.com/jacobalmon/CS-478/blob/main/Homework/Homework%202/hw2.ipynb

Note that the results used are from the GitHub Repo.

Understanding the Starter Code & Purpose:

Within the starter code, we have a method to generate a spiral dataset by specifying the number of points per class and the number of classes. The purpose is to understand the exploding and vanishing gradients we will get by initially creating a model. We will also explore how to solve this problem and how to improve the training even more after solving this problem and explore why each of these methods works to improve the training of our network/model.

Task 1: Vanishing/Exploding Gradients

Initially, we will build a network using the sigmoid activation function to understand and visualize what exploding/vanishing gradients are. Exploding/Vanishing gradients happen during backpropagation, gradients often get smaller or bigger, and training never converges to a good solution. The main cause of this is the use of the sigmoid activation function, which we are purposely using to explore this issue. We create a dense neural network with 5 hidden layers, each having 64 neurons, using a sigmoid activation function, and within the output layer, we output for each class using a softmax activation function. Here is the following code for how the network is built:

```
# Task 1: Vanishing/Exploding Gradients.
# Build Model with Sigmoid Activation.
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(n_classes, activation='softmax')
])

# Compile Model.
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
loss='categorical_crossentropy', metrics=['accuracy'])

# Train Model
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_test,
y_test), verbose=0)
```

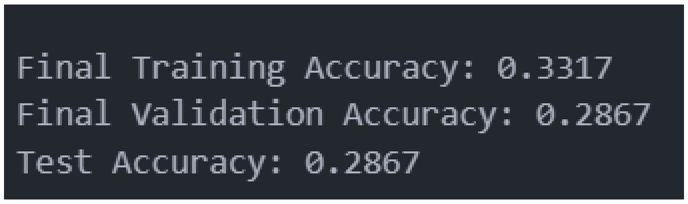
After training the network, we use graphs to visualize the metrics for training & validation loss and training & validation accuracy. We use two different metrics to emphasize the impact of the networks have on differents in this case we are using loss and accuracy. Here is the following code to visualize the vanishing/exploding gradients:

```
# Plot Loss & Accuracy for Visualization
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss',
linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training & Validation Loss')
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy',
linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training & Validation Accuracy')

plt.legend()
plt.show()
```

After visualization, we print out accuracies for training, validation, and testing to evaluate our network. Here is the following code for getting the accuracies:

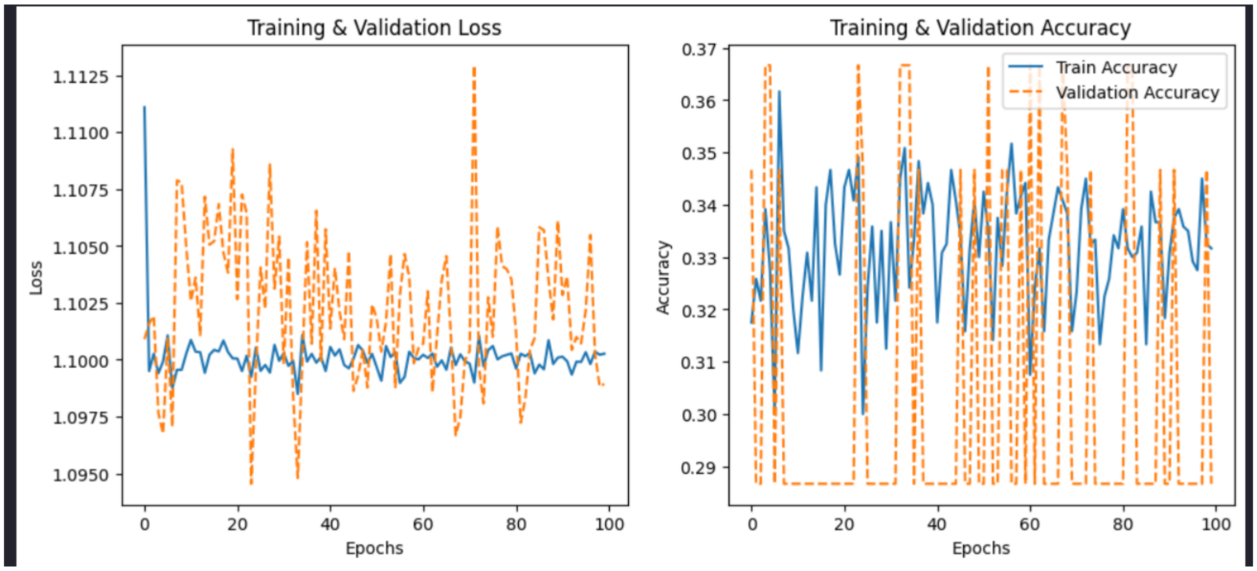
```
# Get Training Accuracy
train_acc = history.history['accuracy'][-1]
val_acc = history.history['val_accuracy'][-1]

# Get Testing Accuracy
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)

# Print Results
print(f"Final Training Accuracy: {train_acc:.4f}")
print(f"Final Validation Accuracy: {val_acc:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
```

Here are the results for the initial network within this task:



```
Final Training Accuracy: 0.3317
Final Validation Accuracy: 0.2867
Test Accuracy: 0.2867
```

Obviously, by just looking at the results, we can see that our accuracies are bad, and to see why this is so, we can look at the graphs we made to visualize how our accuracies are so off. Here are the graphs:



In both of the following graphs, we can see that we are experiencing vanishing gradients because the training loss is oscillating in very small intervals in the 1.1000 range. Another thing is that the validation loss is very unstable as it oscillates pretty erratically, which tells us that our network is not learning. On the other hand, when we look at our accuracy graph, we can see our validation accuracy is very chaotic which tells us that our network is again not learning. Training accuracy oscillates again just like training loss, but not extreme, it's somewhat tame compared to validation. Another note is that the use of the sigmoid activation function condenses outputs into a range between zero and one, and its derivative will be extremely close to zero, where backpropagation has no gradient to propagate back through the network.

Task 2:

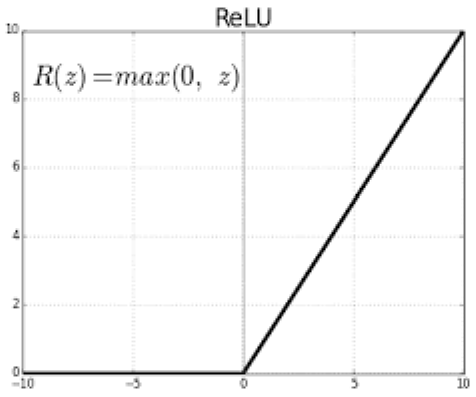
Within this task, we will run three experiments where we see how each experiment improves during training, and what benefits it has towards the network. The first experiment will use a different activation known as ReLU, the second experiment will add a Batch Normalization layer to the previous experiment (first), and the third experiment will add a drop-out layer to the previous experiment (second).

Experiment 1 - Using ReLU Activation Function

The network is built the same way as the previous network, but this time we are using the ReLU activation function. We are using this activation function because when $z > 0$, the gradient becomes one, it allows effective backpropagation, and when $z \leq 0$, the gradient becomes zero, it avoids the vanishing gradient problem, where z represents the input of the activation function at a given neuron. Here is the mathematic expression for the ReLU activation function:

$$ReLU(z) = max(0,\, z)$$

As well as the graph for the activation function:



In this activation function our output is not limited to only zero and one compared to sigmoid, it allows bigger gradients for positive z values leading to a faster learning rate. Altogether we avoid the vanishing gradient problem for positive z. However, when $z \leq 0$, it leads to the Dead Neuron Problem which can be experimented with using Leaky ReLU or ELU, but for this experiment, we are going to ignore this problem, as the problem is not always present. Here is the following code for building this network:

```
# Task 2: Improving Training. Experiment 1 - Change Activation Function from
Sigmoid to ReLU.
# Build Model with ReLU Activation.
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(n_classes, activation='softmax')
])

# Compile Model.
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
loss='categorical_crossentropy', metrics=['accuracy'])

# Train Model
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_test,
y_test), verbose=0)
```

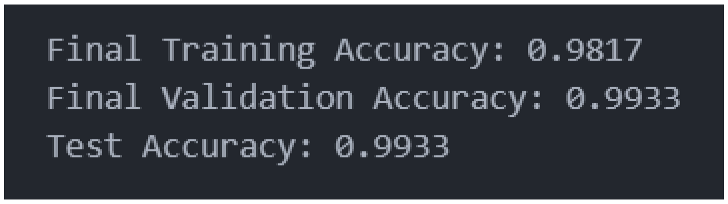
Using the same metrics as before from the previous task we build the same visualizations to compare to our previous model:

```
# Plot Loss & Accuracy for Visualization
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss',
linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training & Validation Loss')

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy',
linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training & Validation Accuracy')

plt.legend()
plt.show()
```

Same for the accuracies, they are retrieved the same way as before, here is the code for that:

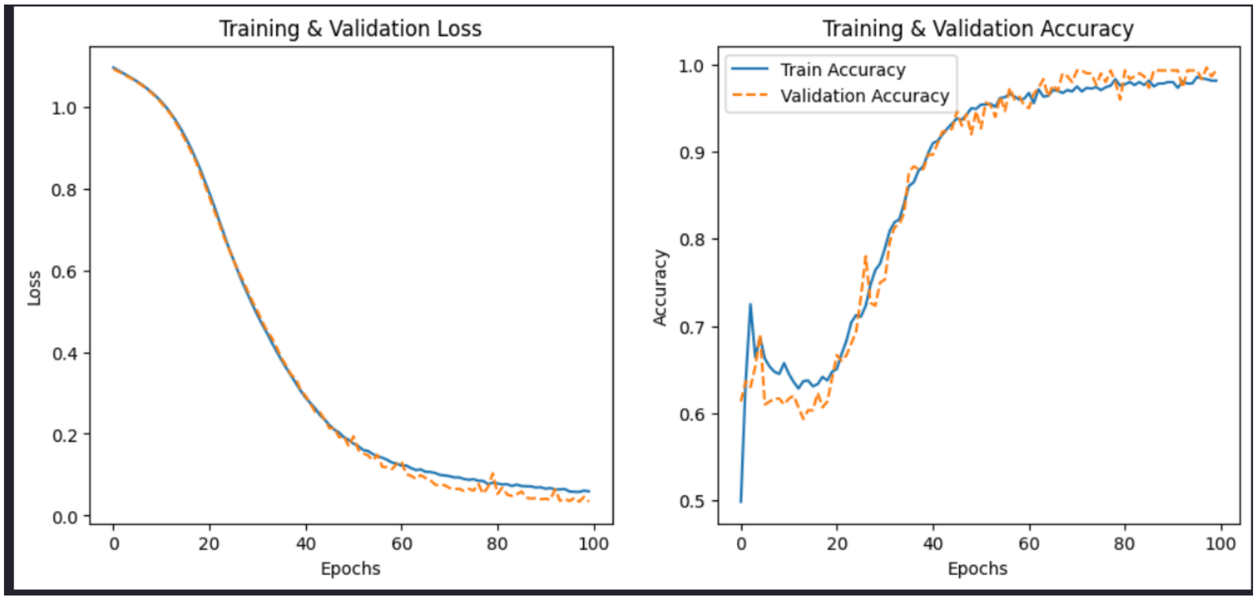
```
# Get Training Accuracy
train_acc = history.history['accuracy'][-1]
val_acc = history.history['val_accuracy'][-1]

# Get Testing Accuracy
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)

# Print Results
print(f"Final Training Accuracy: {train_acc:.4f}")
print(f"Final Validation Accuracy: {val_acc:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
```

Here are the accuracies found for training, validation, and testing:



```
Final Training Accuracy: 0.9817
Final Validation Accuracy: 0.9933
Test Accuracy: 0.9933
```

Our accuracies have improved a lot more than the previous experiments, we will notice that in all the experiments within this task, they will have about the same accuracies, however how fast they learn is different between each improvement as we add onto this experiment, this will be the focus of the report. Here are the graphs for the network using the same metrics as before:



Looking at the training and validation loss we can see that both the lines are very close to each other which is good because it tells it's not overfitting. On the other graph, we notice there is a little bit of oscillation at the beginning, but it eventually converges as we go through more epochs, and this applies to both training and validation, so it starts converging at around 50 epochs.

Experiment 2 - Adding Batch Normalization

The network is built the same way as the previous network, but we are adding a Batch Normalization layer to our network. Batch Normalization is a technique to stabilize and accelerate training by normalizing the inputs in each layer. This is done by finding the vector of input means, evaluated over the whole mini-batch using:

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)}$$

With our vector of input means, we can find the vector of input standard deviations, evaluated over the whole mini-batch using:

$$\sigma_{B^2} = \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2$$

With our vector of input standard deviations, we can find the vector of zero-centered and normalized inputs for instance i using:

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B{}^2 + \varepsilon}}$$

Finally, we can get out our output for our batch normalization, using:

$$z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta$$

This is essentially how batch normalization is found, for simplicity, we're not going delve into it to much besides the process, but it makes training more stable. Batch Normalization is typically done before the hidden layers, but we received better convergence when we used it after the first hidden layer. Here is the code for building the model:

```
# Task 2: Improving Training. Experiment 3 - Added Dropout Layer.
# Build Model with ReLU Activation & Batch Normalization & Dropout.
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(n_classes, activation='softmax')
])

# Compile Model.
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
loss='categorical_crossentropy', metrics=['accuracy'])

# Train Model
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_test, y_test),
verbose=0)
```

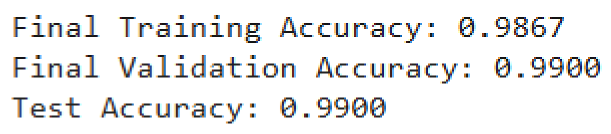
Similarly, we use the same metrics as the previous networks, Here is the code for that:

```
# Plot Loss & Accuracy for Visualization
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss', linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training & Validation Loss')

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy',
linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training & Validation Accuracy')

plt.legend()
plt.show()
```

Again, similarly we find the accuracies for our training, validation, and testing, this is done in this following code:

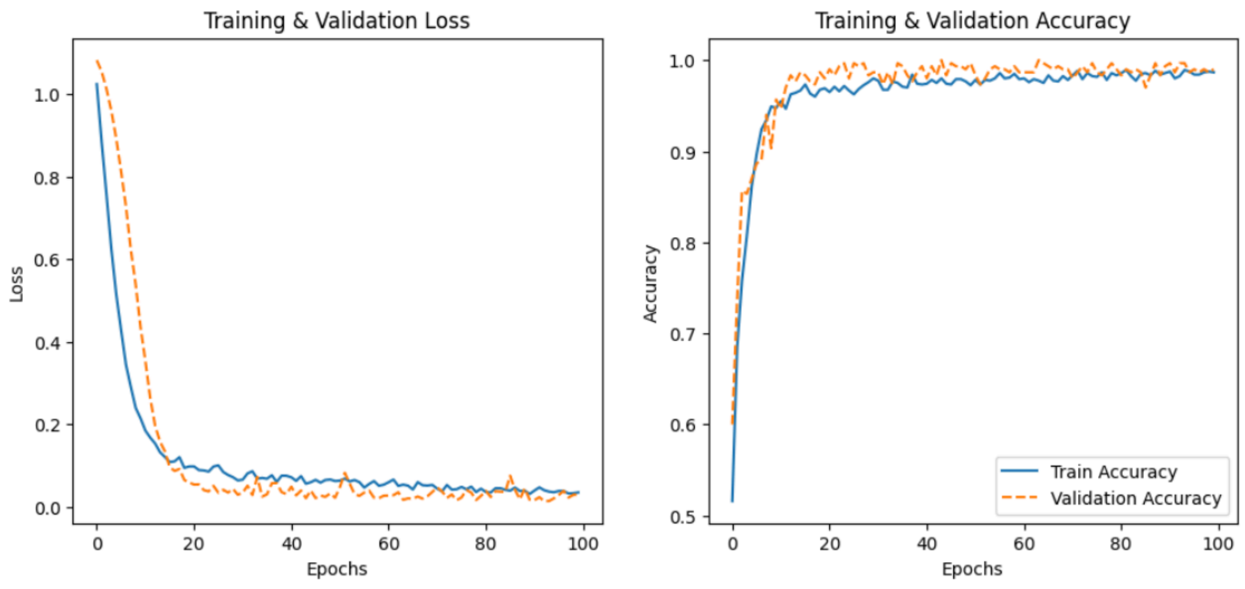
```
# Get Training Accuracy
train_acc = history.history['accuracy'][-1]
val_acc = history.history['val_accuracy'][-1]

# Get Testing Accuracy
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)

# Print Results
print(f"Final Training Accuracy: {train_acc:.4f}")
print(f"Final Validation Accuracy: {val_acc:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
```

Here are the accuracies for the following network:

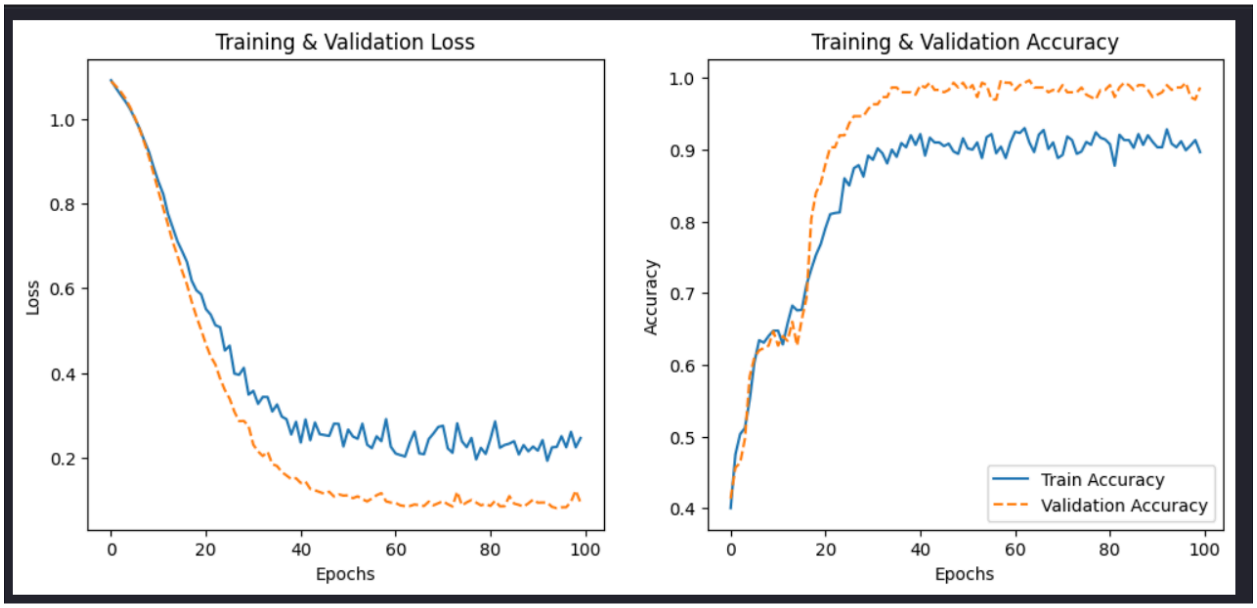


```
Final Training Accuracy: 0.9867
Final Validation Accuracy: 0.9900
Test Accuracy: 0.9900
```

As we can see, our accuracies from this and the previous network are very similar which is expected, but what we care about is how fast it takes to converge/learn in the graph. Previously it started converging around 50 epochs, here is the graph for this network:



In our graph this time, it took around 20 epochs to converge and oscillate to a point as it converges, so in this network, we get a much faster learning rate than the last network. As mentioned before this is applied after one hidden layer which was found to have better convergence and closeness between training and validation. Here is what it looked like when it was done before all the hidden layers:



The training and validation are sparse from each other and it takes longer to learn than the previous one above.

Experiment 3 - Adding Dropout Layer

The network is built the same way as the previous network, but we are going to add a dropout layer. The purpose of the dropout layer is give our network a one to two percent boost, where every input neuron would have a probability p of being temporarily dropped out, meaning it will be ignored in the training process. Note that p is a hyperparameter called dropout rate, and in ours we set it to 20%. We use 20% as our dropout because its a good balance for preventing overfitting by randomly dropping neurons in training, and for this experiment, it was found best to keep it at this. Here is the following code for build of the network:

```python
# Task 2: Improving Training. Experiment 3 - Added Dropout Layer.
# Build Model with ReLU Activation & Batch Normalization & Dropout.
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(n_classes, activation='softmax')
])

# Compile Model.
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
loss='categorical_crossentropy', metrics=['accuracy'])

# Train Model
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_test, y_test),
verbose=0)
```

Similarly, we use the same metrics as the previous networks, Here is the code for that:

```python
# Plot Loss & Accuracy for Visualization
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss', linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training & Validation Loss')

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy',
linestyle='dashed')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training & Validation Accuracy')
plt.legend()
plt.show()
```

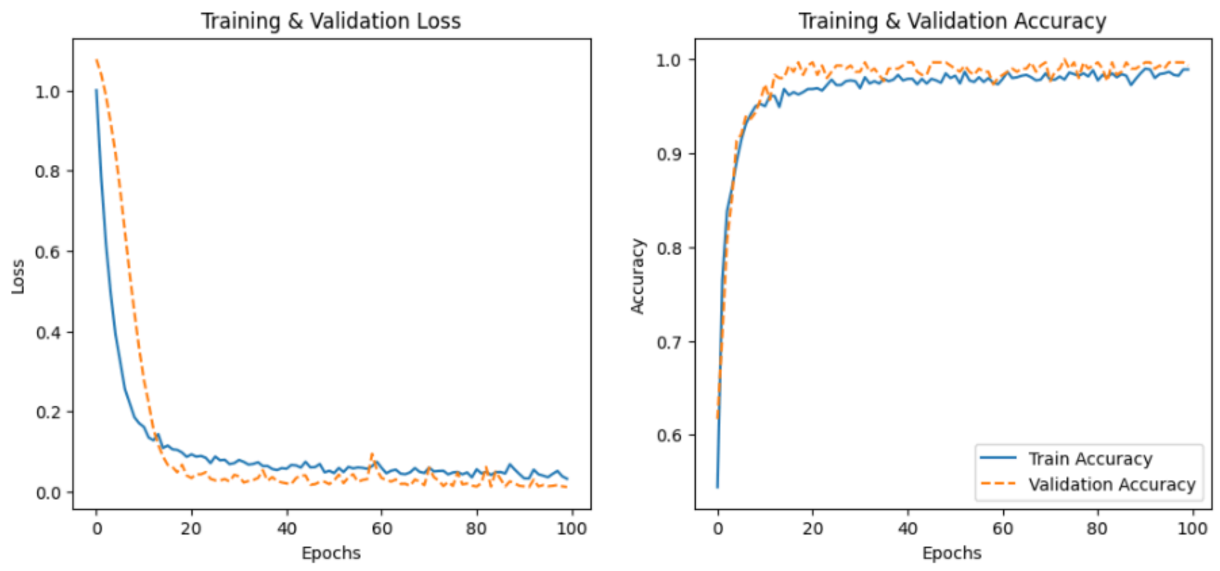
Again, similarly we find the accuracies for our training, validation, and testing, this is done in this following code:

```
# Get Training Accuracy
train_acc = history.history['accuracy'][-1]
val_acc = history.history['val_accuracy'][-1]

# Get Testing Accuracy
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)

# Print Results
print(f"Final Training Accuracy: {train_acc:.4f}")
print(f"Final Validation Accuracy: {val_acc:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
```

Here are the results of the accuracy:

```
Final Training Accuracy: 0.9892
Final Validation Accuracy: 0.9933
Test Accuracy: 0.9933
```

As we can the accuracy barely improved from the last network by just 0.33%, so yes better for this network it may not have been needed and batch normalization was just enough, but if we look at our graphs:



Similarly to last one it start converging at around 20 epochs so there isnt much difference, however if our accuracy was a little lower, it would have helped, but since the accuracy is pretty close to 100%, there won’t be a substantial difference.