```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from itertools import product
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import tensorflow as tf
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.callbacks import EarlyStopping
from keras import models, layers, regularizers
from keras.models import Model
from keras.layers import Input, Dense, Flatten, Dropout, BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Flatten, Dense, Dropout
from tensorflow.keras.initializers import GlorotUniform, RandomNormal
from tensorflow.keras.optimizers import Adam, SGD, RMSprop, Adagrad
from tensorflow.keras.regularizers import l1_l2
from keras.regularizers import l1_l2
from keras.initializers import GlorotUniform, RandomNormal
```

HW-2.1: Data preparation

```
# Load MNIST dataset and split it into training and testing sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()


x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=123)


# Print the shape
print("Shape of x_train:", x_train.shape)
print("Shape of x_test:", x_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
print("Shape of x_val:", x_val.shape)
```

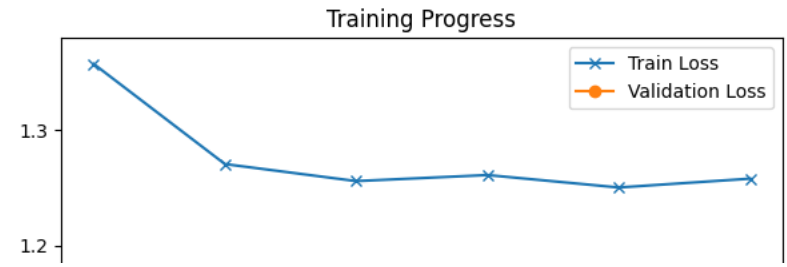
```
Shape of x_train: (48000, 28, 28, 1)
Shape of x_test: (10000, 28, 28, 1)
Shape of y_train: (48000, 10)
Shape of y_test: (10000, 10)
Shape of x_val: (12000, 28, 28, 1)
```

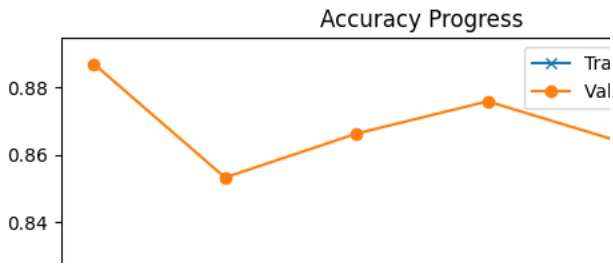
HW-2.2: Generalized model

```
# Define a function for creating a general neural network model with customizable parameters
def general_model_function(input_shape, num_classes, optimizer='adam', learning_rate=0.001,
                           num_layers=3, neurons_per_layer=64, dropout_rate=0.2, activation='relu',
                           metrics=['accuracy'], weight_init='xavier', l1_const=0.0, l2_const=0.0):

    # Define the input layer
    inputs = Input(shape=input_shape)
    x = Flatten()(inputs)  # Flatten the input data if needed

    # Choose weight initialization method based on the 'weight_init' parameter
    if weight_init == 'xavier':
        initial = GlorotUniform(seed=42)  # Initialize weights using Glorot (Xavier) method
    else:
        initial = RandomNormal(seed=42)  # Initialize weights using a random normal distribution
```

```
    # Create the neural network architecture with customizable layers
    for _ in range(num_layers):
        x = Dense(neurons_per_layer, activation=activation, kernel_initializer=initial,
                  kernel_regularizer=l1_l2(l1_const, l2_const))(x)
        x = Dropout(dropout_rate)(x)  # Apply dropout for regularization

    # Define the output layer
    outputs = Dense(num_classes, activation='softmax')(x)

    # Choose the optimizer based on the 'optimizer' parameter
    if optimizer == 'adam':
        opt = Adam(learning_rate=learning_rate)  # Use Adam optimizer
    elif optimizer == 'sgd':
        opt = SGD(learning_rate=learning_rate)  # Use Stochastic Gradient Descent optimizer
    elif optimizer == 'rmsprop':
        opt = RMSprop(learning_rate=learning_rate)  # Use RMSprop optimizer
    elif optimizer == 'adagrad':
        opt = Adagrad(learning_rate=learning_rate)  # Use Adagrad optimizer
    else:
        raise ValueError("Invalid optimizer specified. Please choose from the following optimizers: ['adam', 'sgd', 'rmsprop', 'a

    # Compile the model with the chosen optimizer, loss, and metrics
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=metrics)

    return model


def train_model_function(data, hyperparams, visualization=False):

    x_train, y_train, x_val, y_val = data
    input_shape = x_train[0].shape
    num_classes = y_train.shape[1]

    # Extract hyperparameters
    l1_reg = hyperparams.get('l1', 0.0)
    l2_reg = hyperparams.get('l2', 0.0)
    drop_rate = hyperparams.get('dropout_rate', 0.2)
    learn_rate = hyperparams.get('learning_rate', 0.001)
    weight_initialization = hyperparams.get('weight_init', 'xavier')
    activation_func = hyperparams.get('activation', 'relu')
    num_hidden_layers = hyperparams.get('num_layers', 2)
    neurons_per_hidden_layer = hyperparams.get('neurons_per_layer', 32)
    optimizer_choice = hyperparams.get('optimizer', 'adam')
    batch_size = hyperparams.get('batch_size', 128)

    # Create a customized model using specified hyperparameters
    model = general_model_function(input_shape, num_classes, optimizer=optimizer_choice,
                                   learning_rate=learn_rate, num_layers=num_hidden_layers,
                                   neurons_per_layer=neurons_per_hidden_layer, dropout_rate=drop_rate,
                                   activation=activation_func, weight_init=weight_initialization,
                                   l1_const=l1_reg, l2_const=l2_reg)

    # Set up early stopping to prevent overfitting
    early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

    # Train the customized model
    history = model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=20, batch_size=batch_size, callbacks=[early_stop

    if visualization:
        # Visualize training progress
        plt.figure(figsize=(12, 5))
        plt.subplot(1, 2, 1)
        plt.plot(history.history['loss'], label='Train Loss', marker ='x')
        plt.plot(history.history['val_loss'], label='Validation Loss', marker ='o')
        plt.legend()
        plt.title('Training Progress')

        plt.subplot(1, 2, 2)
        plt.plot(history.history['accuracy'], label='Train Accuracy', marker ='x')
        plt.plot(history.history['val_accuracy'], label='Validation Accuracy', marker ='o')
        plt.legend()
        plt.title('Accuracy Progress')
        plt.tight_layout()
        plt.show()

        # Visualize the confusion matrix
        prediction = model.predict(x_val)
```

```
        predicted_labels = tf.argmax(prediction, axis=1).numpy()
        actual_labels = tf.argmax(y_val, axis=1).numpy()
        confusionmatrix = confusion_matrix(actual_labels, predicted_labels)
        plt.figure(figsize=(10, 7))
        sns.heatmap(confusionmatrix, annot=True, fmt='g',cmap="viridis")
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.show()

    return history.history['loss'][-1], history.history['val_loss'][-1], history.history['val_accuracy'][-1]


# Define a dictionary of hyperparameters for configuring the model
hyperparameters = {
    'l1': 0.001,                # L1 regularization coefficient
    'l2': 0.001,                # L2 regularization coefficient
    'dropout_rate': 0.3,        # Dropout rate for regularization
    'learning_rate': 0.01,     # Learning rate for optimization
    'weight_init': 'xavier',    # Weight initialization method
    'activation': 'relu',       # Activation function for hidden layers
    'num_layers': 2,            # Number of hidden layers
    'neurons_per_layer': 32,    # Number of neurons per hidden layer
    'optimizer': 'adam',     # Choice of optimizer
    'batch_size': 64           # Batch size for training
}

# Call the training function to train the model with the defined hyperparameters
train_loss, val_loss, val_accuracy = train_model_function((x_train, y_train, x_val, y_val), hyperparameters, visualization=True)
```





HW-2.4: Hyper-parameter tuning

choice-0

```
# Define a range of learning rates to explore
learning_rates = [0.001, 0.05, 0.01, 0.1, 0.3]

# Initialize lists to store training and validation errors for each learning rate
train_errors_random_init = []
val_errors_random_init = []
train_errors_xavier_init = []
val_errors_xavier_init = []
val_accuracies_random_init = []
val_accuracies_xavier_init = []


# Hyperparameters for hyper-parameter choice-0
hyperparams_choice_0 = {
    'num_layers': 2,
    'dropout_rate': 0.2,
    'neurons_per_layer': 32,
    'activation': 'relu',
    'optimizer': 'adam',
    'batch_size': 64
}

# Iterate over different learning rates
for learning_rate in learning_rates:
    # Set the learning rate in hyperparameters
    hyperparams_choice_0['learning_rate'] = learning_rate

    # Set weight initialization to random
    hyperparams_choice_0['weight_init'] = 'random'

    # Train the model with random weight initialization
    train_error, val_error, val_accuracy_random_init = train_model_function((x_train, y_train, x_val, y_val), hyperparams_choice_
    val_accuracies_random_init.append(val_accuracy_random_init)
    train_errors_random_init.append(train_error)
    val_errors_random_init.append(val_error)

    # Set weight initialization to Xavier
    hyperparams_choice_0['weight_init'] = 'xavier'

    # Train the model with Xavier weight initialization
    train_error, val_error, val_accuracy_xavier_init = train_model_function((x_train, y_train, x_val, y_val), hyperparams_choice_
    train_errors_xavier_init.append(train_error)
    val_errors_xavier_init.append(val_error)
    val_accuracies_xavier_init.append(val_accuracy_xavier_init)
```

It is not so explicit that whether xavier is better than random_init. However, it is noted that learning rate plays an important role, and smaller learning rate as 0.001 lead to a larger validation accuracy rate

```
# Define colors for the lines
random_init_train_color = 'b'
random_init_val_color = 'lightblue'
xavier_init_train_color = 'r'
xavier_init_val_color = 'pink'
```

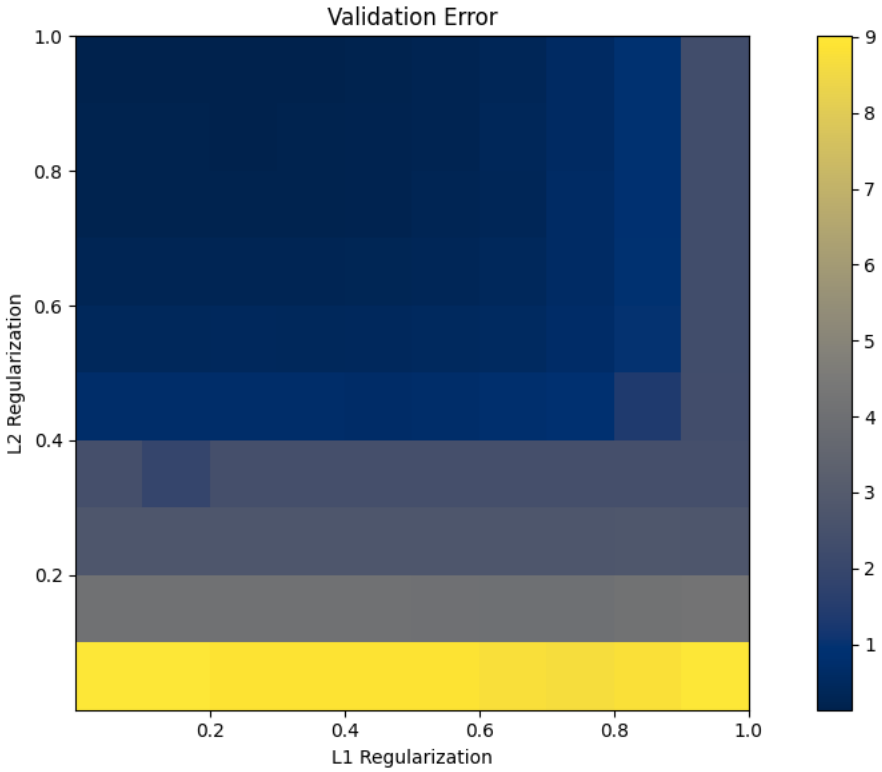
```
# Plot the results
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(learning_rates, train_errors_random_init, label='Random Init Train Loss', color=random_init_train_color, marker='x')
plt.plot(learning_rates, val_errors_random_init, label='Random Init Validation Loss', color=random_init_val_color, marker='o')
plt.plot(learning_rates, train_errors_xavier_init, label='Xavier Init Train Loss', color=xavier_init_train_color, linestyle='dott
plt.plot(learning_rates, val_errors_xavier_init, label='Xavier Init Validation Loss', color=xavier_init_val_color, linestyle='dot
plt.legend()
plt.title('Training and Validation Error vs. Learning Rate')
plt.xlabel('Learning Rate')
plt.ylabel('Error')

plt.show()

# Print the validation accuracies
for i, learning_rate in enumerate(learning_rates):
    print(f"Learning Rate: {learning_rate}")
    print(f"Validation Accuracy (Random Init): {val_accuracies_random_init[i]}")
    print(f"Validation Accuracy (Xavier Init): {val_accuracies_xavier_init[i]}")
    print()
```

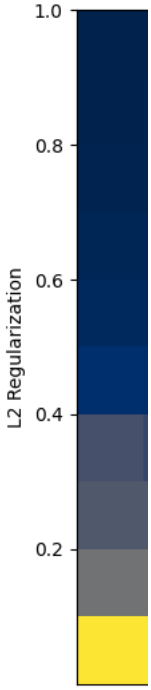

```
Learning Rate: 0.001
Validation Accuracy (Random Init): 0.9584166407585144
Validation Accuracy (Xavier Init): 0.9565833210945129

Learning Rate: 0.05
Validation Accuracy (Random Init): 0.7276666760444641
Validation Accuracy (Xavier Init): 0.5347499847412109

Learning Rate: 0.01
Validation Accuracy (Random Init): 0.9399999976158142
Validation Accuracy (Xavier Init): 0.9423333406448364

Learning Rate: 0.1
Validation Accuracy (Random Init): 0.10366666316986084
Validation Accuracy (Xavier Init): 0.20358332991600037

Learning Rate: 0.3
Validation Accuracy (Random Init): 0.11441666632890701
Validation Accuracy (Xavier Init): 0.09650000184774399
```

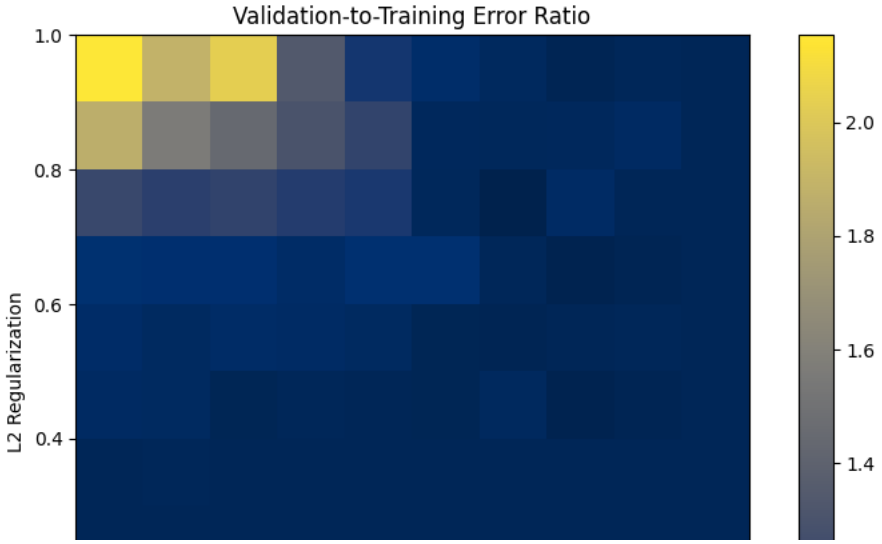
choice-2 - gridsearch

```
# Define a grid of L1 and L2 regularization values
l1_values = np.logspace(-5, 0, 10)  # 10 values between 1e-5 and 1
l2_values = np.logspace(-5, 0, 10)  # 10 values between 1e-5 and 1

# Initialize arrays to store training and validation errors
train_errors = np.zeros((len(l1_values), len(l2_values)))
```

```
val_errors = np.zeros((len(l1_values), len(l2_values)))
error_ratio = np.zeros((len(l1_values), len(l2_values)))

# Hyperparameters for hyper-parameter choice-1 without dropout
hyperparams_choice_1 = {
    'num_layers': 2,
    'neurons_per_layer': 64,
    'activation': 'relu',
    'optimizer': 'adam',
    'batch_size': 64,
    'dropout_rate': 0  # Set dropout rate to 0 for no dropout
}

for i, l1 in enumerate(l1_values):
    for j, l2 in enumerate(l2_values):
        # Set L1 and L2 regularization in hyperparameters
        hyperparams_choice_1['l1'] = l1
        hyperparams_choice_1['l2'] = l2

        # Train the model without dropout
        train_error, val_error, val_accuracy = train_model_function((x_train, y_train, x_val, y_val), hyperparams_choice_1, visua
        train_errors[i, j] = train_error
        val_errors[i, j] = val_error
        error_ratio[i, j] = val_error / train_error



# Plot the results
plt.figure(figsize=(20, 6))

# Plot validation and training errors
plt.subplot(1, 2, 1)
plt.imshow(val_errors, interpolation='nearest', cmap='cividis', extent=[l1_values[0], l1_values[-1], l2_values[0], l2_values[-1]]
plt.colorbar()
plt.title('Validation Error')
plt.xlabel('L1 Regularization')
plt.ylabel('L2 Regularization')

plt.subplot(1, 2, 2)
plt.imshow(train_errors, interpolation='nearest', cmap='cividis', extent=[l1_values[0], l1_values[-1], l2_values[0], l2_values[-1
plt.colorbar()
plt.title('Training Error')
plt.xlabel('L1 Regularization')
plt.ylabel('L2 Regularization')

plt.tight_layout()
plt.show()

# Plot the ratio of validation to training error
plt.figure(figsize=(8, 6))
plt.imshow(error_ratio, interpolation='nearest', cmap='cividis', extent=[l1_values[0], l1_values[-1], l2_values[0], l2_values[-1]
plt.colorbar()
plt.title('Validation-to-Training Error Ratio')
plt.xlabel('L1 Regularization')
plt.ylabel('L2 Regularization')
plt.tight_layout()
plt.show()


# Print the validation accuracy
best_validation_error = np.min(val_errors)
best_validation_error_index = np.unravel_index(np.argmin(val_errors), val_errors.shape)
best_l1 = l1_values[best_validation_error_index[0]]
best_l2 = l2_values[best_validation_error_index[1]]

print(f"The Best Validation Accuracy: {100 - best_validation_error*100:.2f}%")
print(f"The Best L1 Regularization: {best_l1}")
print(f"The Best L2 Regularization: {best_l2}")
```



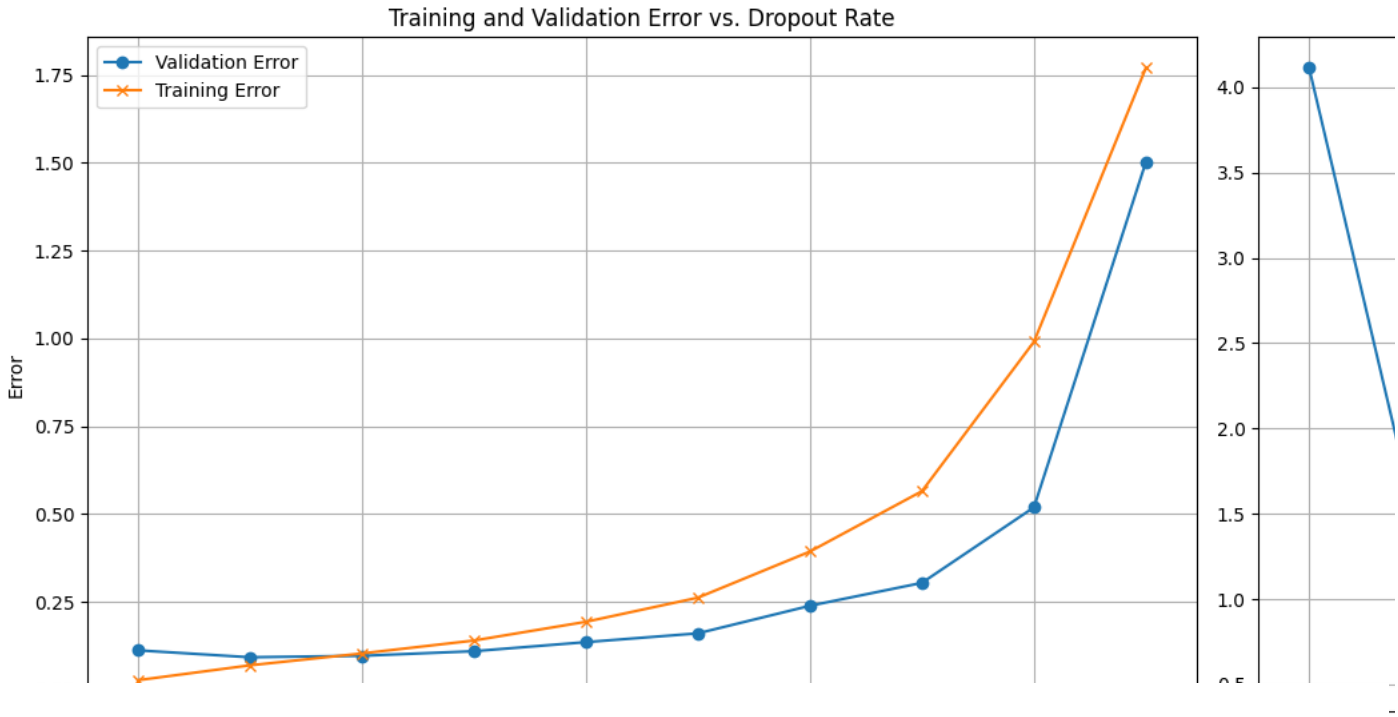




choice-2

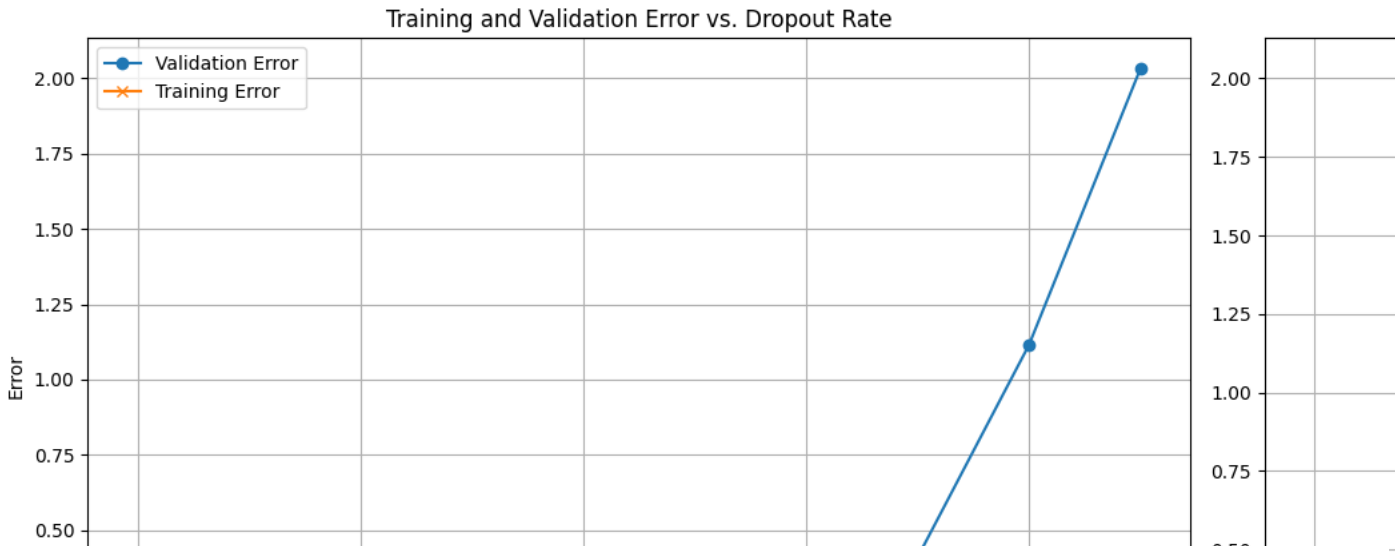
```
# Define a grid of dropout rates to search for optimal model configurations
dropout_rates = np.linspace(0.0, 0.9, 10)  # Vary dropout rates from 0.0 to 0.9

# Initialize arrays to store results
training_errors = np.zeros(len(dropout_rates))
validation_errors = np.zeros(len(dropout_rates))
error_ratios = np.zeros(len(dropout_rates))

best_dropout_rate = None  # Variable to store the best dropout rate
best_val_accuracy = 0.0  # Variable to store the best validation accuracy

hyperparams_choice_2 = {
    'l1': 0.0,  # No L1 regularization
    'l2': 0.0,  # No L2 regularization
    'learning_rate': 0.001,  # Keep learning rate constant
    'activation': 'sigmoid',  # Sigmoid activation
    'num_layers': 3,  # Three hidden layers [96, 96, 96]
    'neurons_per_layer': 96,  # Neurons per hidden layer
    'optimizer': 'rmsprop',  # RMSprop optimizer
```

```
    'batch_size': 64
}

for i, dropout_rate in enumerate(dropout_rates):
    # Set dropout rate in hyperparameters
    hyperparams_choice_2['dropout_rate'] = dropout_rate

    # Train the model without L1 or L2 regularization
    train_error, val_error, val_accuracy = train_model_function((x_train, y_train, x_val, y_val), hyperparams_choice_2, visualiza

    # Store the errors in the arrays
    training_errors[i] = train_error
    validation_errors[i] = val_error
    error_ratios[i] = val_error / train_error

    # Check if the current dropout rate results in a higher validation accuracy
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_dropout_rate = dropout_rate


# Plot the results
plt.figure(figsize=(18, 6))

# Plot training and validation errors
plt.subplot(1, 2, 1)
plt.plot(dropout_rates, validation_errors, label='Validation Error', marker='o')
plt.plot(dropout_rates, training_errors, label='Training Error', marker='x')
plt.legend()
plt.title('Training and Validation Error vs. Dropout Rate')
plt.xlabel('Dropout Rate')
plt.ylabel('Error')
plt.grid(True)

# Plot the ratio of validation to training error
plt.subplot(1, 2, 2)
plt.plot(dropout_rates, error_ratios, label='Validation-to-Training Error Ratio', marker='o')
plt.title('Validation-to-Training Error Ratio vs. Dropout Rate')
plt.xlabel('Dropout Rate')
plt.grid(True)

plt.tight_layout()
plt.show()

# Print the best dropout rate and its corresponding validation accuracy
print("Best Dropout Rate:", best_dropout_rate)
print("Best Validation Accuracy:", best_val_accuracy)
```

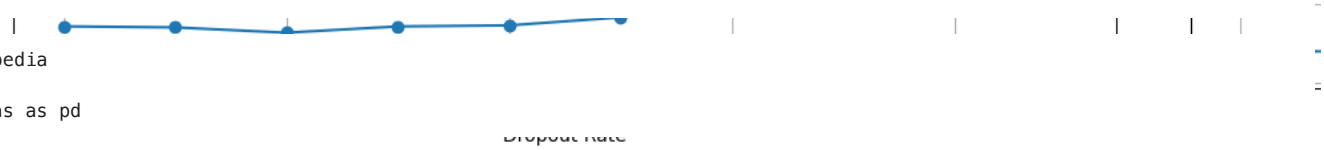


This is so interesting, as the dropout rate the training error tends to increase constantly while the validation error decreases. This actually makes perfect sense to explain why drop out help generalization.

```
# Define a grid of dropout rates to search for optimal model configurations
dropout_rates = np.linspace(0.0, 0.9, 10)  # Vary dropout rates from 0.0 to 0.9

# Initialize arrays to store results
training_errors = np.zeros(len(dropout_rates))
validation_errors = np.zeros(len(dropout_rates))
error_ratios = np.zeros(len(dropout_rates))
train_errors = np.zeros(len(dropout_rates))
validation_errors = np.zeros(len(dropout_rates))

best_val_accuracy = 0.0  # Variable to store the best validation accuracy
best_dropout_rate = None  # Variable to store the dropout rate corresponding to the best accuracy

hyperparams_choice_3 = {
    'l1': 0.0,  # No L1 regularization
    'l2': 0.0,  # No L2 regularization
    'learning_rate': 0.001,  # Keep learning rate constant
    'activation': 'relu',  # ReLU activation
    'num_layers': 3,  # Three hidden layers [96, 96, 96]
    'neurons_per_layer': 96,  # Neurons per hidden layer
    'optimizer': 'adam',  # Adam optimizer
    'batch_size': 64
}

for i, dropout_rate in enumerate(dropout_rates):
    # Set dropout rate in hyperparameters
    hyperparams_choice_3['dropout_rate'] = dropout_rate

    # Train the model without L1 or L2 regularization
    train_error, val_error, val_accuracy = train_model_function((x_train, y_train, x_val, y_val), hyperparams_choice_3, visualiza
    train_errors[i] = train_error
    validation_errors[i] = val_error
    error_ratios[i] = val_error / train_error

    # Check if the current validation accuracy is the best
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_dropout_rate = dropout_rate


# Plot the results
plt.figure(figsize=(18, 6))

# Plot training and validation errors as a function of dropout rate
plt.subplot(1, 2, 1)
plt.plot(dropout_rates, validation_errors, label='Validation Error', marker='o')
plt.plot(dropout_rates, training_errors, label='Training Error', marker='x')
plt.legend()
plt.title('Training and Validation Error vs. Dropout Rate')
plt.xlabel('Dropout Rate')
plt.ylabel('Error')
plt.grid(True)

# Plot the difference between validation and training errors as a function of dropout rate
plt.subplot(1, 2, 2)
error_differences = validation_errors - training_errors
plt.plot(dropout_rates, error_differences, label='Validation - Training Error', marker='o')
plt.title('Validation - Training Error Difference vs. Dropout Rate')
plt.xlabel('Dropout Rate')
plt.grid(True)

plt.tight_layout()
plt.show()

# Print the best validation accuracy and the corresponding dropout rate
print("Best Validation Accuracy:", best_val_accuracy)
print("Best Dropout Rate:", best_dropout_rate)
```



bonus points



```
import wikipedia
import nltk
import pandas as pd
```

```
# define the topic
topics = ["Animals", "Technology", "History", "Science", "Sports", "Food","Music","Artificial Intelligence","Travel"]
wikipedia.set_lang("en")


data = []
for topic in topics:
  results = wikipedia.search(topic,results=20,suggestion=True)
  for i in results[0]:
    try:
      text = wikipedia.summary(i)
      temp = [topic,text]
      data.append(temp)
    except:
      continue
```

```
/usr/local/lib/python3.10/dist-packages/wikipedia/wikipedia.py:389: GuessedAtParserWarning: No parser was explicitly specifi

The code that caused this warning is on line 389 of the file /usr/local/lib/python3.10/dist-packages/wikipedia/wikipedia.py.

  lis = BeautifulSoup(html).find_all('li')
```

```
# split the data
df = pd.DataFrame(data = data,columns=['label','content'])


def create_text_chunks(text, label):
    sentences = nltk.sent_tokenize(text)
    chunk_size = 5
    text_chunks = [sentences[i:i + chunk_size] for i in range(0, len(sentences), chunk_size)]
    labeled_chunks = [(chunk, label) for chunk in text_chunks]
    return labeled_chunks


dataset = []
for label,text in data:
  labeled_chunks = create_text_chunks(label = label,text = text)
  for i in labeled_chunks[0][0]:
    temp = [i,labeled_chunks[0][1]]
    dataset.append(temp)
texts, labels = zip(*dataset)


from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
```

```
from tensorflow import keras
from tensorflow.keras.models import Sequential


# Create a TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer()
X = tfidf_vectorizer.fit_transform(texts)

# Encode labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(labels)

# Convert labels to one-hot encoding
y = to_categorical(y, num_classes=len(label_encoder.classes_))

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Build an artificial neural network (ANN) model
model = Sequential()
model.add(Dense(128, input_dim=X_train.shape[1], activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(len(label_encoder.classes_), activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
history = model.fit(X_train.toarray(), y_train, epochs=30, batch_size=256, validation_split=0.2)

# Evaluate the model
loss, accuracy = model.evaluate(X_test.toarray(), y_test)
print(f'Test Loss: {loss:.4f}')
print(f'Test Accuracy: {accuracy * 100:.2f}%')
```

```
2/2 [==============================] - 0s 93ms/step - loss: 2.1669 - accuracy: 0.5390 - val_loss: 2.1841 - val_accuracy: 0.2
Epoch 4/30
2/2 [==============================] - 0s 87ms/step - loss: 2.1490 - accuracy: 0.7049 - val_loss: 2.1777 - val_accuracy: 0.3
Epoch 5/30
2/2 [==============================] - 0s 49ms/step - loss: 2.1287 - accuracy: 0.7780 - val_loss: 2.1696 - val_accuracy: 0.3
Epoch 6/30
2/2 [==============================] - 0s 63ms/step - loss: 2.1047 - accuracy: 0.8390 - val_loss: 2.1596 - val_accuracy: 0.4
Epoch 7/30
2/2 [==============================] - 0s 35ms/step - loss: 2.0768 - accuracy: 0.8415 - val_loss: 2.1474 - val_accuracy: 0.4
Epoch 8/30
2/2 [==============================] - 0s 47ms/step - loss: 2.0442 - accuracy: 0.8659 - val_loss: 2.1329 - val_accuracy: 0.4
Epoch 9/30
2/2 [==============================] - 0s 33ms/step - loss: 2.0051 - accuracy: 0.8829 - val_loss: 2.1162 - val_accuracy: 0.4
Epoch 10/30
2/2 [==============================] - 0s 34ms/step - loss: 1.9655 - accuracy: 0.8756 - val_loss: 2.0973 - val_accuracy: 0.5
Epoch 11/30
2/2 [==============================] - 0s 38ms/step - loss: 1.9102 - accuracy: 0.9049 - val_loss: 2.0756 - val_accuracy: 0.5
Epoch 12/30
2/2 [==============================] - 0s 33ms/step - loss: 1.8525 - accuracy: 0.9073 - val_loss: 2.0511 - val_accuracy: 0.5
Epoch 13/30
2/2 [==============================] - 0s 34ms/step - loss: 1.7923 - accuracy: 0.9171 - val_loss: 2.0239 - val_accuracy: 0.5
Epoch 14/30
2/2 [==============================] - 0s 38ms/step - loss: 1.7217 - accuracy: 0.9268 - val_loss: 1.9938 - val_accuracy: 0.5
Epoch 15/30
2/2 [==============================] - 0s 37ms/step - loss: 1.6471 - accuracy: 0.9415 - val_loss: 1.9609 - val_accuracy: 0.5
Epoch 16/30
2/2 [==============================] - 0s 34ms/step - loss: 1.5507 - accuracy: 0.9634 - val_loss: 1.9252 - val_accuracy: 0.5
Epoch 17/30
2/2 [==============================] - 0s 33ms/step - loss: 1.4705 - accuracy: 0.9585 - val_loss: 1.8875 - val_accuracy: 0.5
Epoch 18/30
2/2 [==============================] - 0s 34ms/step - loss: 1.3732 - accuracy: 0.9780 - val_loss: 1.8478 - val_accuracy: 0.5
Epoch 19/30
2/2 [==============================] - 0s 33ms/step - loss: 1.2678 - accuracy: 0.9805 - val_loss: 1.8064 - val_accuracy: 0.6
Epoch 20/30
2/2 [==============================] - 0s 41ms/step - loss: 1.1782 - accuracy: 0.9805 - val_loss: 1.7634 - val_accuracy: 0.6
Epoch 21/30
2/2 [==============================] - 0s 39ms/step - loss: 1.0669 - accuracy: 0.9878 - val_loss: 1.7187 - val_accuracy: 0.6
Epoch 22/30
2/2 [==============================] - 0s 37ms/step - loss: 0.9714 - accuracy: 0.9854 - val_loss: 1.6731 - val_accuracy: 0.6
```

```
2/2 [==============================] - 0s 42ms/step - loss: 0.6858 - accuracy: 0.9976 - val_loss: 1.5325 - val_accuracy: 0.6
Epoch 26/30
2/2 [==============================] - 0s 45ms/step - loss: 0.6009 - accuracy: 0.9976 - val_loss: 1.4862 - val_accuracy: 0.6
Epoch 27/30
2/2 [==============================] - 0s 53ms/step - loss: 0.5173 - accuracy: 0.9951 - val_loss: 1.4407 - val_accuracy: 0.7
Epoch 28/30
2/2 [==============================] - 0s 62ms/step - loss: 0.4480 - accuracy: 0.9976 - val_loss: 1.3967 - val_accuracy: 0.7
Epoch 29/30
2/2 [==============================] - 0s 39ms/step - loss: 0.3761 - accuracy: 0.9927 - val_loss: 1.3548 - val_accuracy: 0.7
Epoch 30/30
2/2 [==============================] - 0s 42ms/step - loss: 0.3269 - accuracy: 1.0000 - val_loss: 1.3156 - val_accuracy: 0.7
5/5 [==============================] - 0s 5ms/step - loss: 1.2641 - accuracy: 0.7519
Test Loss: 1.2641
Test Accuracy: 75.19%
```

```
# Visualize training progress
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss', marker ='x')
plt.plot(history.history['val_loss'], label='Validation Loss', marker ='o')
plt.legend()
plt.title('Training Progress')

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy', marker ='x')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', marker ='o')
plt.legend()
plt.title('Accuracy Progress')
plt.tight_layout()
plt.show()
```