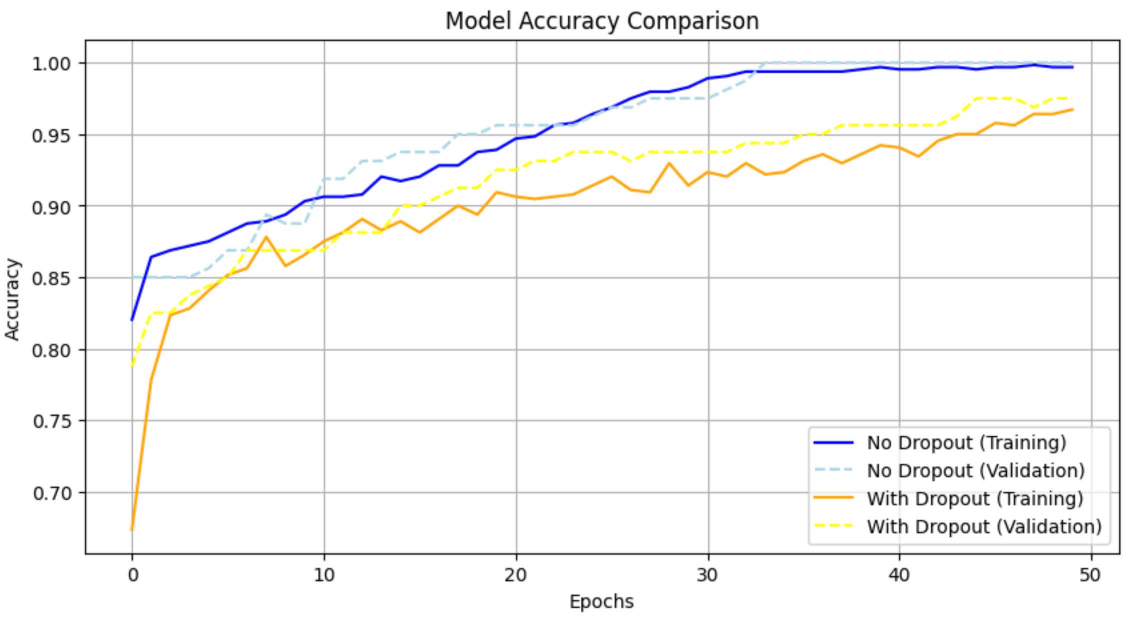
```
#BatchNormalization

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, BatchNormalization
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Reshape and normalize the data
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).astype('float32') / 255.0
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype('float32') / 255.0

# One-hot encoding of labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Train model without Batch Normalization
model_no_bn = Sequential()
model_no_bn.add(Flatten(input_shape=(28, 28, 1)))
model_no_bn.add(Dense(128, activation='relu'))
model_no_bn.add(Dense(64, activation='relu'))
model_no_bn.add(Dense(10, activation='softmax'))
model_no_bn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history_no_bn = model_no_bn.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=0)

# Train model with Batch Normalization
model_with_bn = Sequential()
model_with_bn.add(Flatten(input_shape=(28, 28, 1)))
model_with_bn.add(Dense(128, activation='relu'))
model_with_bn.add(BatchNormalization())  # Add Batch Normalization
model_with_bn.add(Dense(64, activation='relu'))
model_with_bn.add(BatchNormalization())  # Add Batch Normalization
model_with_bn.add(Dense(10, activation='softmax'))
model_with_bn.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history_with_bn = model_with_bn.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=0)

# Plotting the accuracy
plt.figure(figsize=(10, 5))
plt.plot(history_no_bn.history['accuracy'], label='No Batch Normalization (Training)', color='blue')
plt.plot(history_no_bn.history['val_accuracy'], label='No Batch Normalization (Validation)', color='lightblue', linestyle='dash
plt.plot(history_with_bn.history['accuracy'], label='With Batch Normalization (Training)', color='orange')
plt.plot(history_with_bn.history['val_accuracy'], label='With Batch Normalization (Validation)', color='yellow', linestyle='das
plt.title('Model Accuracy Comparison: With vs Without Batch Normalization')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid()
plt.show()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.12/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`ir
  super().__init__(**kwargs)
```

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
# Create a synthetic dataset
X, y = make_moons(n_samples=1000, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train model without dropout
model_no_dropout = Sequential()
model_no_dropout.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model_no_dropout.add(Dense(32, activation='relu'))
model_no_dropout.add(Dense(1, activation='sigmoid'))
model_no_dropout.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history_no_dropout = model_no_dropout.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, verbose=0)

# Train model with dropout
model_with_dropout = Sequential()
model_with_dropout.add(Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model_with_dropout.add(Dropout(0.5))  # Apply dropout
model_with_dropout.add(Dense(32, activation='relu'))
model_with_dropout.add(Dropout(0.5))  # Apply dropout again
model_with_dropout.add(Dense(1, activation='sigmoid'))
model_with_dropout.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history_with_dropout = model_with_dropout.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2, verbose=0)

# Plotting the accuracy
plt.figure(figsize=(10, 5))
plt.plot(history_no_dropout.history['accuracy'], label='No Dropout (Training)', color='blue')
plt.plot(history_no_dropout.history['val_accuracy'], label='No Dropout (Validation)', color='lightblue', linestyle='dashed')
plt.plot(history_with_dropout.history['accuracy'], label='With Dropout (Training)', color='orange')
plt.plot(history_with_dropout.history['val_accuracy'], label='With Dropout (Validation)', color='yellow', linestyle='dashed')
plt.title('Model Accuracy Comparison')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid()
plt.show()
```

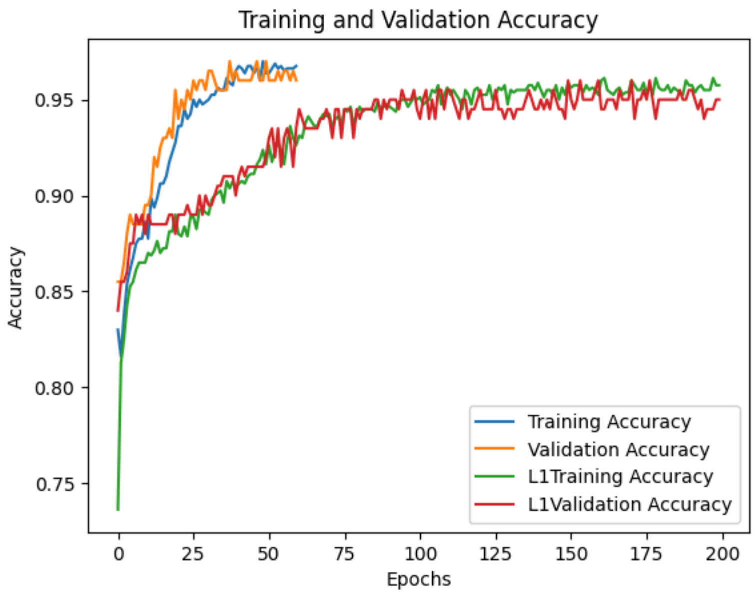


✦ Gemini

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras import regularizers
from tensorflow.keras import regularizers # Corrected import
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.datasets import make_moons
```

```
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping # Import EarlyStopping

x,y = make_moons(n_samples=1000, noise=0.2, random_state=1)

xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.2, random_state=42)

# Define Early Stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)


rmodel = Sequential()
rmodel.add(Dense(64, activation='relu', input_shape=(xtrain.shape[1],)))
rmodel.add(Dense(32, activation='relu'))
rmodel.add(Dense(1, activation='sigmoid'))

rmodel.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

rfit = rmodel.fit(xtrain, ytrain, epochs=200, # Increased epochs, but ES will stop it
                  batch_size=32,
                  validation_data=[xtest, ytest],
                  verbose=0,
                  callbacks=[early_stopping]) # Add early stopping

regmodel = Sequential()
regmodel.add(Dense(64, activation='relu', input_shape=(xtrain.shape[1],), kernel_regularizer=regular
regmodel.add(Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
regmodel.add(Dense(1, activation='sigmoid'))

regmodel.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

regfit = regmodel.fit(xtrain, ytrain, epochs=200, # Increased epochs, but ES will stop it
                      batch_size=32,
                      validation_data=[xtest, ytest],
                      verbose=0,
                      callbacks=[early_stopping]) # Add early stopping


plt.plot(rfit.history['accuracy'], label='Training Accuracy')
plt.plot(rfit.history['val_accuracy'], label='Validation Accuracy')
plt.plot(regfit.history['accuracy'], label='L1Training Accuracy')
plt.plot(regfit.history['val_accuracy'], label='L1Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_di
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```



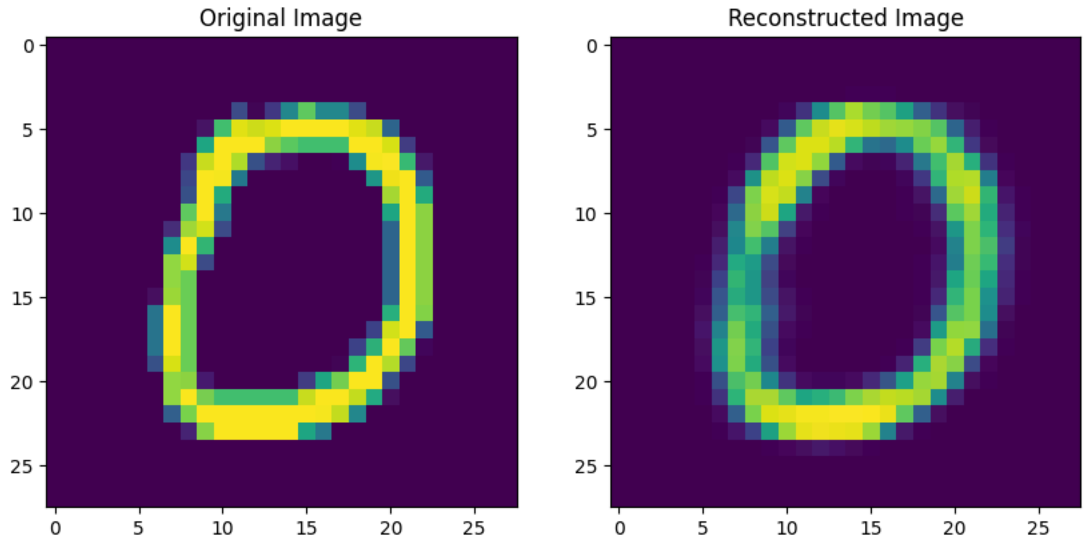

```
#EARLY STOPPING
from tensorflow.keras.callbacks import EarlyStopping

es = EarlyStopping(monitor='val_loss',
                            mode='min',
```

```
                                       patience=10,
                                       restore_best_weights=True)
```

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Flatten images to vectors
x_train = x_train.reshape((len(x_train), 784)).astype('float32') / 255
x_test = x_test.reshape((len(x_test), 784)).astype('float32') / 255

input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)  # Bottleneck layer

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)


# Create the autoencoder model
autoencoder = Model(input_img, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the autoencoder
autoencoder.fit(x_train, x_train,
                epochs=5,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

# Use the encoder part to get encoded representations
encoder = Model(input_img, encoded)
encoded_imgs = encoder.predict(x_test)

# Use the decoder part to reconstruct images from encoded representations
decoder_input = Input(shape=(32,))
decoder_layer = autoencoder.layers[-3](decoder_input)
decoder_layer = autoencoder.layers[-2](decoder_layer)
decoder_layer = autoencoder.layers[-1](decoder_layer)
decoder = Model(decoder_input, decoder_layer)
decoded_imgs = decoder.predict(encoded_imgs)
```

```
Epoch 1/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - loss: 0.3365 - val_loss: 0.1674
Epoch 2/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - loss: 0.1588 - val_loss: 0.1343
Epoch 3/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.1323 - val_loss: 0.1222
Epoch 4/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - loss: 0.1222 - val_loss: 0.1161
Epoch 5/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - loss: 0.1167 - val_loss: 0.1126
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```

```
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(x_test[10].reshape(28,28))
plt.title('Original Image')
plt.subplot(1,2,2)
plt.imshow(decoded_imgs[10].reshape(28,28))
plt.title('Reconstructed Image')
plt.show()
#
```

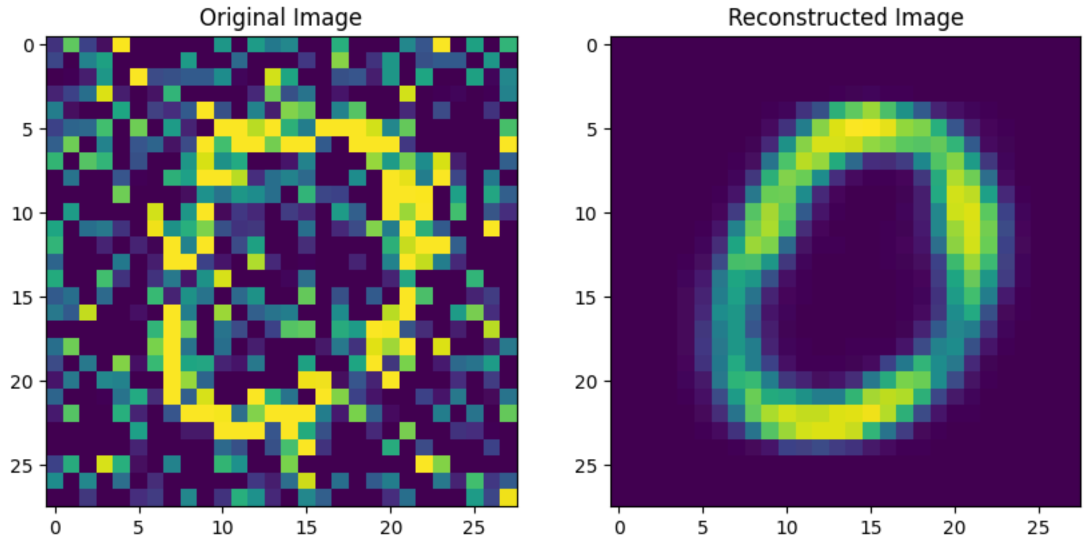



```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Flatten images to vectors
x_train = x_train.reshape((len(x_train), 784)).astype('float32') / 255
x_test = x_test.reshape((len(x_test), 784)).astype('float32') / 255

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)  # Bottleneck layer

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)


# Create the autoencoder model
autoencoder = Model(input_img, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the autoencoder
autoencoder.fit(x_train_noisy, x_train,
                epochs=5,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test_noisy, x_test))

# Use the encoder part to get encoded representations
encoder = Model(input_img, encoded)
encoded_imgs = encoder.predict(x_test_noisy)

# Use the decoder part to reconstruct images from encoded representations
decoder_input = Input(shape=(32,))
decoder_layer = autoencoder.layers[-3](decoder_input)
decoder_layer = autoencoder.layers[-2](decoder_layer)
decoder_layer = autoencoder.layers[-1](decoder_layer)
decoder = Model(decoder_input, decoder_layer)
decoded_imgs = decoder.predict(encoded_imgs)
```

```
Epoch 1/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 11s 37ms/step - loss: 0.3382 - val_loss: 0.2011
Epoch 2/5
```

```
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.1904 - val_loss: 0.1682
Epoch 3/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - loss: 0.1656 - val_loss: 0.1555
Epoch 4/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - loss: 0.1540 - val_loss: 0.1473
Epoch 5/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.1474 - val_loss: 0.1432
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```

```
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(x_test_noisy[10].reshape(28,28))
plt.title('Original Image')
plt.subplot(1,2,2)
plt.imshow(decoded_imgs[10].reshape(28,28))
plt.title('Reconstructed Image')
plt.show()
#
```

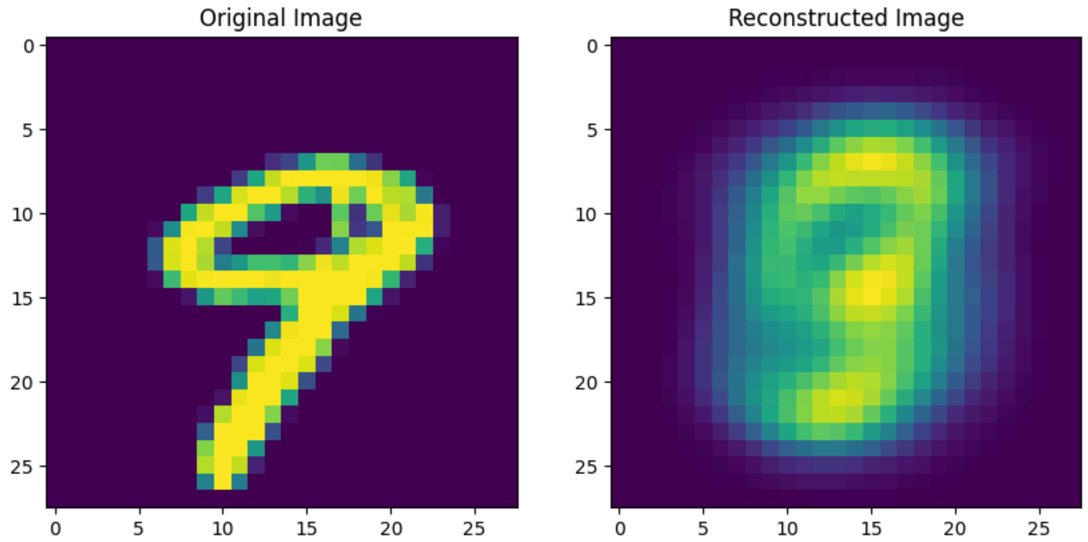



```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
import numpy as np
import matplotlib.pyplot as plt


# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()
# Normalize pixel values to range [0, 1]
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
# Flatten images to vectors
x_train = x_train.reshape((len(x_train), 784))
x_test = x_test.reshape((len(x_test), 784))
# Define autoencoder architecture with sparsity constraint
input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu', activity_regularizer=regularizers.l1(10e-5))(input_img)
encoded = Dense(64, activation='relu', activity_regularizer=regularizers.l1(10e-5))(encoded)
encoded = Dense(32, activation='relu', activity_regularizer=regularizers.l1(10e-5))(encoded)  # Bottleneck layer
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)
# Create the autoencoder model
autoencoder = Model(input_img, decoded)
# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
# Train the autoencoder
autoencoder.fit(x_train, x_train,
                epochs=50, # Increased epochs for sparsity to take effect
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
# Use the encoder part to get encoded representations
encoder = Model(input_img, encoded)
encoded_imgs = encoder.predict(x_test)
# Use the decoder part to reconstruct images from encoded representations
```

```
decoder_input = Input(shape=(32,))
decoder_layer = autoencoder.layers[-3](decoder_input)
decoder_layer = autoencoder.layers[-2](decoder_layer)
decoder_layer = autoencoder.layers[-1](decoder_layer)
decoder = Model(decoder_input, decoder_layer)
decoded_imgs = decoder.predict(encoded_imgs)
```

```
Epoch 23/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2630 - val_loss: 0.2627
Epoch 24/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.2630 - val_loss: 0.2627
Epoch 25/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - loss: 0.2629 - val_loss: 0.2626
Epoch 26/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2632 - val_loss: 0.2626
Epoch 27/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 21ms/step - loss: 0.2631 - val_loss: 0.2625
Epoch 28/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2630 - val_loss: 0.2626
Epoch 29/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2631 - val_loss: 0.2626
Epoch 30/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - loss: 0.2628 - val_loss: 0.2626
Epoch 31/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2631 - val_loss: 0.2626
Epoch 32/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2627 - val_loss: 0.2626
Epoch 33/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 22ms/step - loss: 0.2632 - val_loss: 0.2627
Epoch 34/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2631 - val_loss: 0.2626
Epoch 35/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2630 - val_loss: 0.2626
Epoch 36/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 22ms/step - loss: 0.2632 - val_loss: 0.2626
Epoch 37/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2628 - val_loss: 0.2626
Epoch 38/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2628 - val_loss: 0.2626
Epoch 39/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 22ms/step - loss: 0.2626 - val_loss: 0.2627
Epoch 40/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2630 - val_loss: 0.2626
Epoch 41/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 20ms/step - loss: 0.2631 - val_loss: 0.2626
Epoch 42/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 19ms/step - loss: 0.2629 - val_loss: 0.2626
Epoch 43/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2627 - val_loss: 0.2626
Epoch 44/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 22ms/step - loss: 0.2628 - val_loss: 0.2626
Epoch 45/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 17ms/step - loss: 0.2628 - val_loss: 0.2625
Epoch 46/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 22ms/step - loss: 0.2628 - val_loss: 0.2625
Epoch 47/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.2629 - val_loss: 0.2626
Epoch 48/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 19ms/step - loss: 0.2627 - val_loss: 0.2627
Epoch 49/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - loss: 0.2627 - val_loss: 0.2627
Epoch 50/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - loss: 0.2628 - val_loss: 0.2627
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```

```
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(x_test[9].reshape(28,28))
plt.title('Original Image')
plt.subplot(1,2,2)
plt.imshow(decoded_imgs[9].reshape(28,28))
plt.title('Reconstructed Image')
plt.show()
#
```




```
#RNN
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding
from keras.layers import LSTM
from keras.datasets import imdb


# Load data
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words = 5000)

#length of words
x_train = sequence.pad_sequences(x_train, maxlen=80)
x_test = sequence.pad_sequences(x_test, maxlen=80)

# Creating Model
model = Sequential()
model.add(Embedding(5000, 128))
model.add(LSTM(128,activation="tanh",recurrent_activation="sigmoid"))
model.add(Dense(1, activation = 'sigmoid'))

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

# Training
lstm=model.fit(x_train, y_train, batch_size =70, epochs = 1, validation_data = (x_test, y_test),shuffle=True,verbose=1)

# Summary
model.summary()

# Test
op=model.predict(x_test)
op


# Using test data to check the predicted values
from random import randint
arr_ind=randint(0,24999)
index=imdb.get_word_index()
reverse_index = dict([(value, key) for (key, value) in index.items()])
decoded = "".join([reverse_index.get(i - 3, "#") for i in x_test[arr_ind]])
arr=[]
for i in op:
  if(i<0.5):
    arr.append("Negative")
  else:
    arr.append("Positive")
print("Sentence:",decoded)
print("Review:",arr[arr_ind])
print("Predicted Value:",op[arr_ind][0])
print("Expected Value:",y_test[arr_ind])
```

```
358/358 ━━━━━━━━━━━━━━━━━━━━ 91s 248ms/step - accuracy: 0.5007 - loss: 0.0000e+00 - val_accuracy: 0.5000 - val_loss: 0.0000e+0(
Model: "sequential_13"
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 80, 128)	640,000
lstm_1 (LSTM)	(None, 128)	131,584
dense_79 (Dense)	(None, 1)	129

```
 Total params: 2,315,141 (8.83 MB)
 Trainable params: 771,713 (2.94 MB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 1,543,428 (5.89 MB)
782/782 ━━━━━━━━━━━━━━━━━━━━ 33s 41ms/step
array([[1.1735944e-25],
       [1.1830156e-25],
       [1.1749382e-25],
       ...,
       [1.1516416e-25],
       [1.1729768e-25],
```

```
from random import randint
arr_ind=randint(0,24999)
index=imdb.get_word_index()
reverse_index = dict([(value, key) for (key, value) in index.items()])
decoded = "".join([reverse_index.get(i - 3, "#") for i in x_test[arr_ind]])
arr=[]
for i in op:
  if(i<0.5):
    arr.append("Negative")
  else:
    arr.append("Positive")
print("Sentence:",decoded)
print("Review:",arr[arr_ind])
print("Predicted Value:",op[arr_ind][0])
print("Expected Value:",y_test[arr_ind])
```

```
Sentence: isthethememusicforthismusicalthisfilmwasnominatedformanyawardsandwasabighitattheboxofficeduring#iiwhichkeptpeoplesmir
Review: Negative
Predicted Value: 1.1760502e-25
Expected Value: 1
```

```
#################################Pooling################################
import tensorflow as tf
from keras.layers import AveragePooling2D

# Define a 3x3 matrix using TensorFlow constant
x = tf.constant([[1., 2., 3.],
                 [4., 5., 6.],
                 [7., 8., 9.]])

x= tf.reshape(x,[1,3,3,1])


modelp= Sequential()
modelp.add(AveragePooling2D(pool_size=(2, 2), strides=1, padding='valid'))

modelp.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

modelp.predict(x)
```

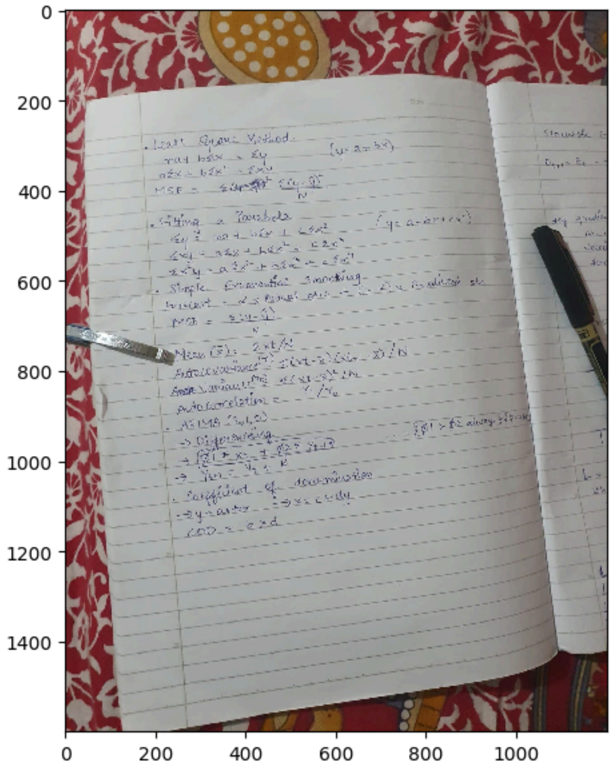
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step
array([[[[3.],
         [4.]],

        [[6.],
         [7.]]]], dtype=float32)
```

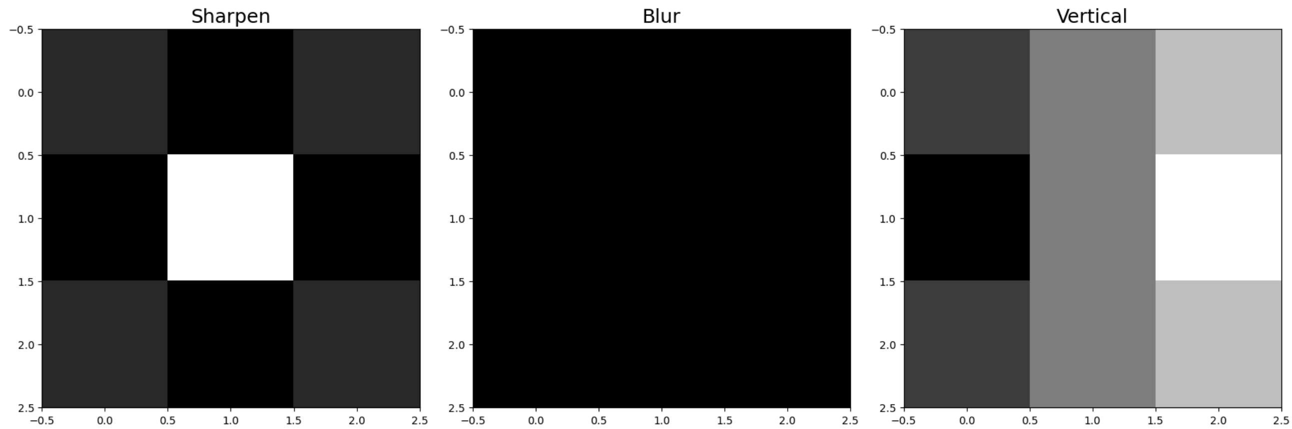
```
################################Filters################################

import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imshow, imread
from skimage.color import rgb2yuv, rgb2hsv, rgb2gray, yuv2rgb, hsv2rgb
from scipy.signal import convolve2d

#Reading Image
dog = imread("/content/1.jpeg")
plt.figure(figsize=(8,6))
imshow(dog)
```

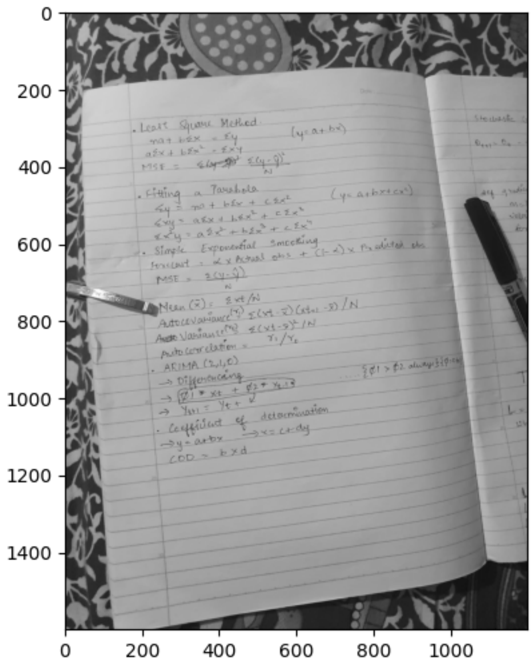
```
import numpy as np
import matplotlib.pyplot as plt

# Filter Matrices
sharpen = np.array([[0, -1, 0],
                    [-1, 5, -1],
                    [0, -1, 0]])

blur = np.array([[0.11, 0.11, 0.11],
                 [0.11, 0.11, 0.11],
                 [0.11, 0.11, 0.11]])

vertical = np.array([[-1, 0, 1],
                     [-2, 0, 2],
                     [-1, 0, 1]])

gaussian = (1/16.0) * np.array([[1, 2, 1],
                                [2, 4, 2],
                                [1, 2, 1]])

# Plotting the filters
fig, ax = plt.subplots(1, 3, figsize=(17, 10))

ax[0].imshow(sharpen, cmap='gray')
ax[0].set_title('Sharpen', fontsize=18)

ax[1].imshow(blur, cmap='gray')
ax[1].set_title('Blur', fontsize=18)

ax[2].imshow(vertical, cmap='gray')
ax[2].set_title('Vertical', fontsize=18)

plt.tight_layout()
plt.show()


# Grayscaling Image
dog_gray = rgb2gray(dog)
plt.figure(figsize=(8, 6))
plt.imshow(dog_gray, cmap='gray')

# Function for applying filters
def multi_convolver(image, kernel, iterations):
    for i in range(iterations):
        image = convolve2d(image, kernel, 'same', boundary='fill', fillvalue=0)
    return image

convolved_image = multi_convolver(dog_gray, sharpen, 1)
plt.figure(figsize=(8, 6))
plt.imshow(convolved_image, cmap='gray')


# For colored Image
def convolver_rgb(image, kernel, iterations=1):
    convolved_image_r = multi_convolver(image[:, :, 0], kernel, iterations)
    convolved_image_g = multi_convolver(image[:, :, 1], kernel, iterations)
    convolved_image_b = multi_convolver(image[:, :, 2], kernel, iterations)

    reformed_image = np.dstack((np.rint(abs(convolved_image_r)),
                                np.rint(abs(convolved_image_g)),
                                np.rint(abs(convolved_image_b)))) / 255

    fig, ax = plt.subplots(1, 3, figsize=(17, 10))

    ax[0].imshow(abs(convolved_image_r), cmap='Reds')
    ax[0].set_title('Red', fontsize=15)

    ax[1].imshow(abs(convolved_image_g), cmap='Greens')
    ax[1].set_title('Green', fontsize=18)

    ax[2].imshow(abs(convolved_image_b), cmap='Blues')
    ax[2].set_title('Blue', fontsize=18)

    return np.array(reformed_image * 255).astype(np.uint8)

# Can add different filters (defined above) here
convolved_rgb_gauss = convolver_rgb(dog, vertical.T, 1)


plt.figure(num=None, figsize=(8, 6), dpi=86)
plt.imshow(convolved_rgb_gauss, vmin=0, vmax=255)
```

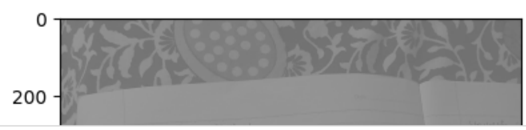
```
plt.axis('off')  # Optional: to hide the axes
plt.show()
```

```
/tmp/ipython-input-3963258008.py:12: FutureWarning: `imshow` is deprecated since version 0.25 and will be removed in version 0
  imshow(dog)
```



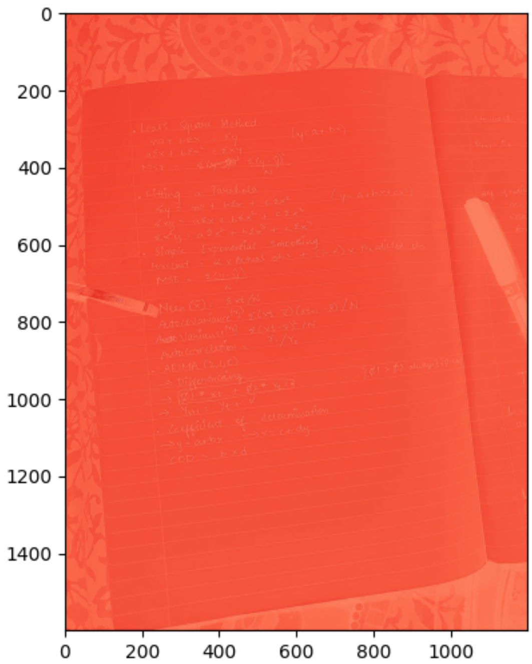


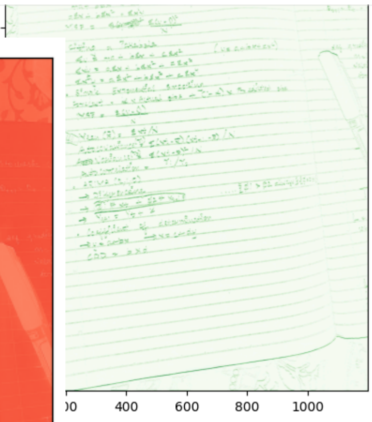


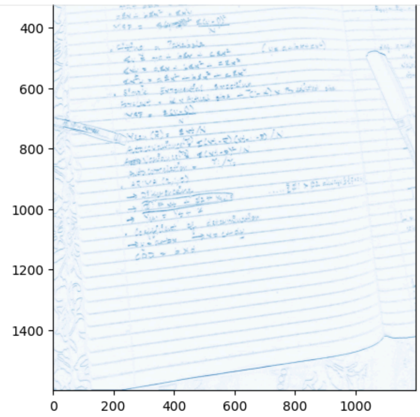


```
sharpen = np.array([[0, -1, 0],
                    [-1, 5, -1],
                    [0, -1, 0]])

blur = np.array([[0.11, 0.11, 0.11],
                 [0.11, 0.11, 0.11],
                 [0.11, 0.11, 0.11]])

vertical = np.array([[-1, 0, 1],
                     [-2, 0, 2],
                     [-1, 0, 1]])

gaussian = (1/16.0) * np.array([[1, 2, 1],
                                 [2, 4, 2],
                                 [1, 2, 1]])

image = convolve2d(dog_gray, sharpen, 'same', boundary='fill', fillvalue=0)

# convolved_image = multi_convolver(dog_gray, sharpen, 1)
plt.figure(figsize=(8, 6))
plt.imshow(image, cmap='gray')

redimg = convolve2d(dog[:, :, 0], sharpen, 'same', boundary='fill', fillvalue=0)
greenimg = convolve2d(dog[:, :, 1], sharpen, 'same', boundary='fill', fillvalue=0)
blueimg = convolve2d(dog[:, :, 2], sharpen, 'same', boundary='fill', fillvalue=0)
plt.imshow(redimg, cmap='Reds')
```

```
<matplotlib.image.AxesImage at 0x7dee08229010>
```







```
######################################################################################################################
#CNN
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, AveragePooling2D

#################################CNN Parametrs################################
model1=Sequential()
model1.add(Conv2D(32,(3,3),strides=1,activation='relu', input_shape=(28,28,3),padding='same' , use_bias=False))
model1.add(Conv2D(17,(3,3), activation='relu',padding='same' , use_bias=False))
model1.add(Conv2D(13,(3,3), activation='relu',padding='same' ))
model1.add(Conv2D(7,(3,3), activation='relu',padding='same' ))
model1.add(Conv2D(3,(3,3), activation='relu',padding='same' ))

model1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model1.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_sha
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

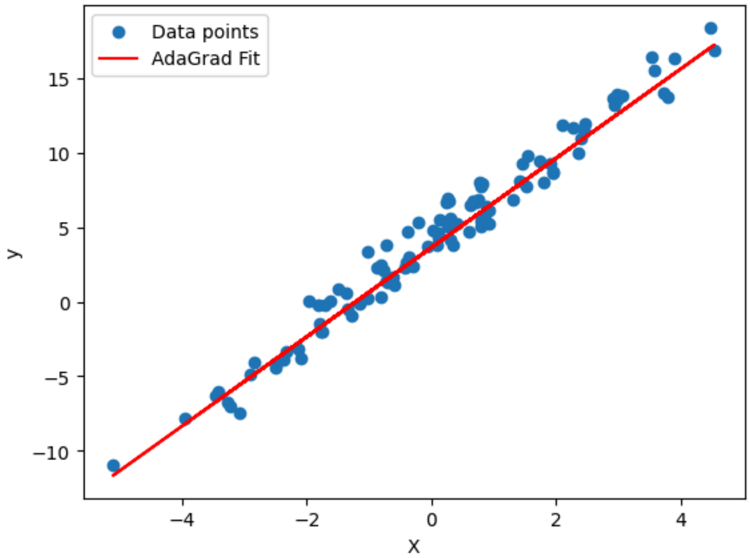
Model: "sequential_14"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	864
conv2d_1 (Conv2D)	(None, 28, 28, 17)	4,896
conv2d_2 (Conv2D)	(None, 28, 28, 13)	2,002
conv2d_3 (Conv2D)	(None, 28, 28, 7)	826
conv2d_4 (Conv2D)	(None, 28, 28, 3)	192

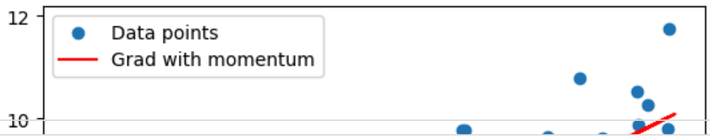
Total params: 8,780 (34.30 KB)
Trainable params: 8,780 (34.30 KB)
Non-trainable params: 0 (0.00 B)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(units=128, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
model.summary()
from tensorflow.keras.optimizers import Adam
model.compile(optimizer=Adam(),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
print("Model compilation complete.")
```

```
#ADAGRAD
import numpy as np
import matplotlib.pyplot as plt

# Generate some example data
np.random.seed(0)  # For reproducibility
X = 2 * np.random.randn(100, 1)  # 100 random samples from 0 to 2
y = 4 + 3 * X + np.random.randn(100, 1)  # Linear relation with noise

# Add bias term (intercept) to the feature matrix
X_b = np.c_[np.ones((100, 1)), X]

# Initialize theta
theta = np.random.randn(2, 1)
eta=0.01
n_iter=1000
# epsilon=1e-8
# S=np.zeros(theta.shape)
# AdaGrad function
def adagrad(X, y, theta, eta, n_iter):
    m = len(y)
    grad_squared = np.zeros(theta.shape)  # Initialize squared gradients
    for i in range(n_iter):
        g = (2 / m) * X_b.T.dot(X_b.dot(theta) - y)  # Compute gradient
        grad_squared += g ** 2  # Accumulate squared gradients
        adjusted_eta = eta / (np.sqrt(grad_squared) + 1e-8)  # Adjust learning rate
        theta = theta - adjusted_eta * g  # Update theta
    return theta

# Perform AdaGrad
theta_adagrad = adagrad(X_b, y, theta, eta=0.1, n_iter=1000)

# Generate predictions based on final theta
y_b = X_b.dot(theta_adagrad)

# Plotting
plt.scatter(X, y, label='Data points')  # Scatter plot for original data points
plt.plot(X, y_b, 'r-', label='AdaGrad Fit')  # Line plot for predictions
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()
```




```
#GRAD WITH MOMENTUM
#import numpy as np
#import matplotlib.pyplot as plt

# Generate some example data
np.random.seed(0)  # For reproducibility
X = 2 * np.random.rand(100, 1)  # 100 random samples from 0 to 2
y = 4 + 3 * X + np.random.randn(100, 1)  # Linear relation with noise

# Add bias term (intercept) to the feature matrix
X_b = np.c_[np.ones((100, 1)), X]

# Initialize theta
theta = np.random.randn(2, 1)

# Gradient Descent with Momentum function
def gd_m(X, y, theta, eta, gamma, n_iter):
    m = len(y)
    vel = np.zeros(theta.shape)  # Initialize velocity
    for i in range(n_iter):
        g = (2 / m) * X.T.dot(X.dot(theta) - y)  # Compute gradient
        vel = gamma * vel + eta * g               # Update velocity
        theta = theta - vel                        # Update theta
    return theta

# Perform Gradient Descent with Momentum
theta_gd_m = gd_m(X_b, y, theta, eta=0.01, gamma=0.9, n_iter=1000)

# Generate predictions based on final theta
y_gd_m = X_b.dot(theta_gd_m)

# Plotting
plt.scatter(X, y, label='Data points')  # Scatter plot for original data points
plt.plot(X, y_gd_m, 'r-', label='Grad with momentum')  # Line plot for predictions
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()
```




```
#ADAGRAD
import numpy as np
import matplotlib.pyplot as plt

# Generate some example data
np.random.seed(0)  # For reproducibility
X = 2 * np.random.randn(100, 1)  # 100 random samples from 0 to 2
y = 4 + 3 * X + np.random.randn(100, 1)  # Linear relation with noise

# Add bias term (intercept) to the feature matrix
X_b = np.c_[np.ones((100, 1)), X]

# Initialize theta
theta = np.random.randn(2, 1)
eta=0.01
n_iter=1000
epsilon=1e-8
# S=np.zeros(theta.shape)
# AdaGrad function
def adagrad(X, y, theta, eta, n_iter):
    m = len(y)
    grad_squared = np.zeros(theta.shape)  # Initialize squared gradients
    for i in range(n_iter):
        g = (2 / m) * X_b.T.dot(X_b.dot(theta) - y)  # Compute gradient
        grad_squared += g ** 2  # Accumulate squared gradients
        adjusted_eta = eta / (np.sqrt(grad_squared) + 1e-8)  # Adjust learning rate
        theta = theta - adjusted_eta * g  # Update theta
    return theta

# Perform AdaGrad
theta_adagrad = adagrad(X_b, y, theta, eta=0.1, n_iter=1000)

# Generate predictions based on final theta
y_b = X_b.dot(theta_adagrad)

# Plotting
plt.scatter(X, y, label='Data points')  # Scatter plot for original data points
plt.plot(X, y_b, 'r-', label='AdaGrad Fit')  # Line plot for predictions
plt.xlabel("X")
```