Activity 1.3 : Regularization

Objective(s):

This activity aims to demonstrate how to apply regularization in neural networks

Intended Learning Outcomes (ILOs):

- Demonstrate how to build and train neural networks with regularization
- Demonstrate how to visualize the model with regularization
- Evaluate the result of model with regularization

Resources:

- Jupyter Notebook
- MNIST

Procedures

Load the necessary libraries

In [1]:
```
from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

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

Load the data, shuffled and split between train and test sets

In [2]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Get the size of the sample train data

In [3]:
```
x_train[0].shape
```

Out[3]:
```
(28, 28)
```

Check the sample train data

In [4]:
```
x_train[333]
```

```
Out[4]: array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  87, 138,
        170, 253, 201, 244, 212, 222, 138,  86,  22,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  95, 253, 252,
        252, 252, 252, 253, 252, 252, 252, 252, 245,  80,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  68, 246, 205,  69,
         69,  69,  69,  69,  69,  69,  69, 205, 253, 240,  50,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 187, 252, 218,  34,
          0,   0,   0,   0,   0,   0,   0, 116, 253, 252,  69,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 116, 248, 252, 253,  92,
          0,   0,   0,   0,   0,   0,  95, 230, 253, 157,   6,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 116, 249, 253, 189,  42,
          0,   0,   0,   0,  36, 170, 253, 243, 158,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 133, 252, 245, 140,
         34,   0,   0,  57, 219, 252, 235,  60,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  25, 205, 253, 252,
        234, 184, 184, 253, 240, 100,  44,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  21, 161, 219,
        252, 252, 252, 234,  37,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 203,
        252, 252, 252, 251, 135,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   9,  76, 255, 253,
        205, 168, 220, 255, 253, 137,   5,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 114, 252, 249, 132,
         25,   0,   0, 180, 252, 252,  45,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  51, 220, 252, 199,   0,
          0,   0,   0,  38, 186, 252, 154,   7,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 184, 252, 252,  21,   0,
          0,   0,   0,   0,  67, 252, 252,  22,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 184, 252, 200,   0,   0,
          0,   0,   0,   0,  47, 252, 252,  22,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 185, 253, 201,   0,   0,
          0,   0,   0,   3, 118, 253, 245,  21,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 163, 252, 252,   0,   0,
          0,   0,   0,  97, 252, 252,  87,   0,   0,   0,   0,   0,   0,
```

```
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  51, 240, 252, 123,  70,
         70, 112, 184, 222, 252, 170,  13,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 165, 252, 253, 252,
        252, 252, 252, 245, 139,  13,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   9,  75, 253, 252,
        221, 137, 137,  21,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0]], dtype=uint8)
```

Check the corresponding label in the training set

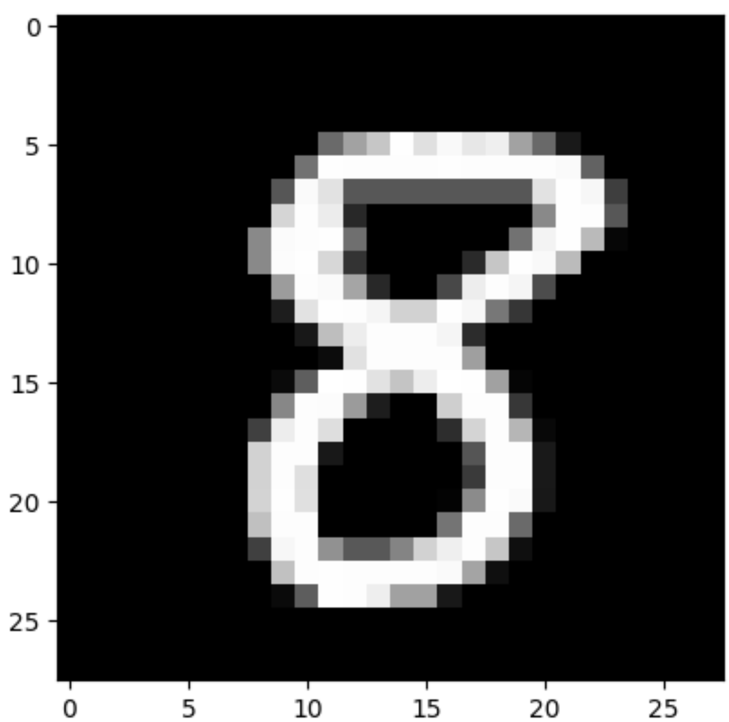
In [5]:
```
y_train[333]
```

Out[5]:
```
np.uint8(8)
```

Check the actual image

In [6]:
```
plt.imshow(x_train[333], cmap='Greys_r')
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x147dc82ef90>
```



Check the shape of the x_train and x_test

In [7]:
```
print(x_train.shape, 'train samples')
print(x_test.shape, 'test samples')
```

```
(60000, 28, 28) train samples
(10000, 28, 28) test samples
```

- Convert the x_train and x_test
- Cast the numbers to floats
- Normalize the inputs

In [8]:
```python
x_train = x_train.reshape(len(x_train), 28*28)
x_test = x_test.reshape(len(x_test), 28*28)


x_train = x_train.astype('float32')
x_test = x_test.astype('float32')


x_train /= 255
x_test /= 255
```

Convert class vectors to binary class matrices

In [9]:
```python
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

y_train[333]  # now the digit k is represented by a 1 in the kth entry (0-indexed) of the leng
```

Out[9]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 1., 0.])
```

- Build the model with two hidden layers of size 512.
- Use dropout of 0.2
- Check the model summary

In [10]:
```python
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

```
c:\Python312\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `
Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [11]:
```python
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 512)	401,920
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 512)	262,656
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 10)	5,130

Total params: 669,706 (2.55 MB)

Trainable params: 669,706 (2.55 MB)

Non-trainable params: 0 (0.00 B)

Compile the model using learning rate of 0.001 and optimizer of RMSprop

In [12]:
```
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(learning_rate=0.001),
              metrics=['accuracy'])
batch_size = 128  # mini-batch with 128 examples
epochs = 30
history = model.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
Epoch 1/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8572 - loss: 0.4477 - val_accuracy: 0.9
574 - val_loss: 0.1278
Epoch 2/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9663 - loss: 0.1121 - val_accuracy: 0.9
725 - val_loss: 0.0830
Epoch 3/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9764 - loss: 0.0757 - val_accuracy: 0.9
770 - val_loss: 0.0737
Epoch 4/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9824 - loss: 0.0580 - val_accuracy: 0.9
786 - val_loss: 0.0687
Epoch 5/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9865 - loss: 0.0438 - val_accuracy: 0.9
826 - val_loss: 0.0648
Epoch 6/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9887 - loss: 0.0369 - val_accuracy: 0.9
810 - val_loss: 0.0700
Epoch 7/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9891 - loss: 0.0342 - val_accuracy: 0.9
812 - val_loss: 0.0658
Epoch 8/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9912 - loss: 0.0273 - val_accuracy: 0.9
836 - val_loss: 0.0610
Epoch 9/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9917 - loss: 0.0243 - val_accuracy: 0.9
853 - val_loss: 0.0647
Epoch 10/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9928 - loss: 0.0219 - val_accuracy: 0.9
823 - val_loss: 0.0725
Epoch 11/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9930 - loss: 0.0220 - val_accuracy: 0.9
840 - val_loss: 0.0690
Epoch 12/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9940 - loss: 0.0183 - val_accuracy: 0.9
827 - val_loss: 0.0778
Epoch 13/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9953 - loss: 0.0152 - val_accuracy: 0.9
853 - val_loss: 0.0764
Epoch 14/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9958 - loss: 0.0127 - val_accuracy: 0.9
849 - val_loss: 0.0702
Epoch 15/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9953 - loss: 0.0128 - val_accuracy: 0.9
818 - val_loss: 0.0900
Epoch 16/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9962 - loss: 0.0113 - val_accuracy: 0.9
844 - val_loss: 0.0768
Epoch 17/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9959 - loss: 0.0114 - val_accuracy: 0.9
839 - val_loss: 0.0846
Epoch 18/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9968 - loss: 0.0094 - val_accuracy: 0.9
853 - val_loss: 0.0802
Epoch 19/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9961 - loss: 0.0111 - val_accuracy: 0.9
839 - val_loss: 0.0947
Epoch 20/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9964 - loss: 0.0118 - val_accuracy: 0.9
852 - val_loss: 0.0835
Epoch 21/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9967 - loss: 0.0105 - val_accuracy: 0.9
852 - val_loss: 0.0745
Epoch 22/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9974 - loss: 0.0083 - val_accuracy: 0.9
```

```
847 - val_loss: 0.0866
Epoch 23/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9973 - loss: 0.0076 - val_accuracy: 0.9
857 - val_loss: 0.0821
Epoch 24/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9980 - loss: 0.0065 - val_accuracy: 0.9
837 - val_loss: 0.0950
Epoch 25/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9976 - loss: 0.0074 - val_accuracy: 0.9
850 - val_loss: 0.0821
Epoch 26/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9980 - loss: 0.0065 - val_accuracy: 0.9
862 - val_loss: 0.0894
Epoch 27/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9984 - loss: 0.0051 - val_accuracy: 0.9
839 - val_loss: 0.0987
Epoch 28/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9980 - loss: 0.0066 - val_accuracy: 0.9
843 - val_loss: 0.0986
Epoch 29/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9978 - loss: 0.0064 - val_accuracy: 0.9
862 - val_loss: 0.0890
Epoch 30/30
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9986 - loss: 0.0041 - val_accuracy: 0.9
845 - val_loss: 0.0979
```

Use Keras evaluate function to evaluate performance on the test set

In [13]:
```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.09788373112678528
Test accuracy: 0.984499990940094
```

Interpret the result

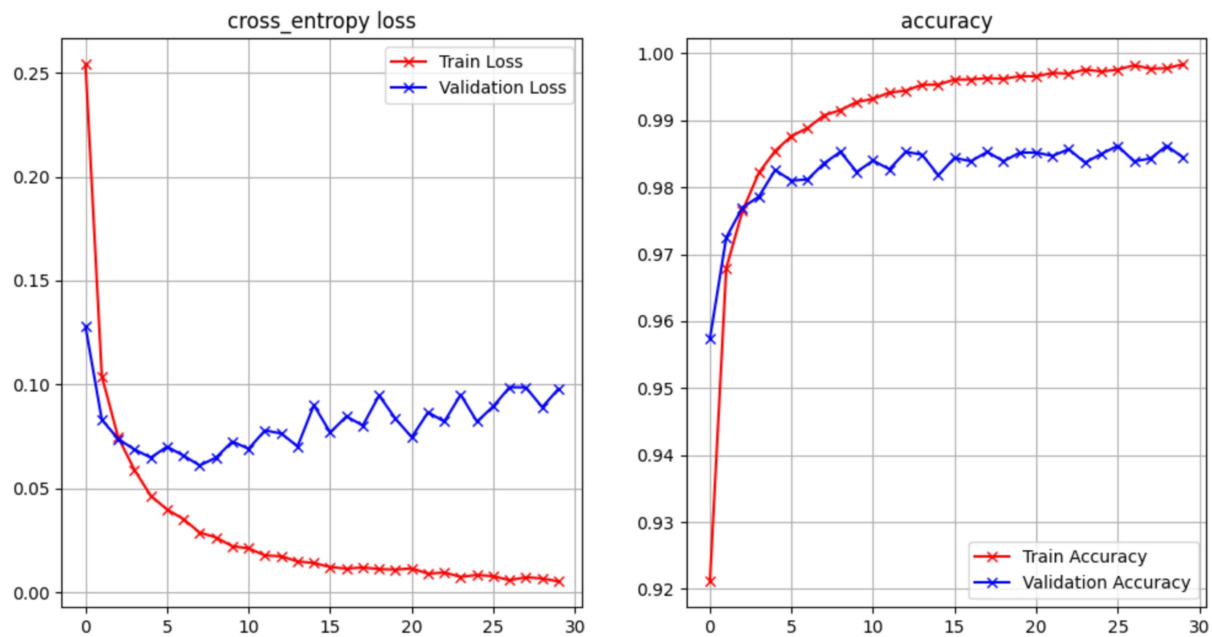
In this activity, we train the MNIST dataset with neural network model using a Sequential mode where it has three dense layerReLU and SoftMax as activation function with two hidden layers of the size of 512. We also use a dropout of 0.2. Dropout is a regularization technique used in neural networks to prevent overfitting of the data were it randomly sets a fraction input units to zero at each update step which reduce the reliance of specific neuron on forcing the network to learn robust features. We specifically use the categorical_crossentropy as our loss function which is use for multi-class classification with RMSprop as optimizer and learning rate of 0.001, batch size of 128 and epoch of 30. In this model we got an accuracy of 98.55 % and a loss of 9.31 % which indicate that our model perform well during the training.

In [14]:

```
def plot_loss_accuracy(history):
    fig = plt.figure(figsize=(12, 6))
    ax = fig.add_subplot(1, 2, 1)
    ax.plot(history.history["loss"],'r-x', label="Train Loss")
    ax.plot(history.history["val_loss"],'b-x', label="Validation Loss")
    ax.legend()
    ax.set_title('cross_entropy loss')
    ax.grid(True)


    ax = fig.add_subplot(1, 2, 2)
    ax.plot(history.history["accuracy"],'r-x', label="Train Accuracy")
    ax.plot(history.history["val_accuracy"],'b-x', label="Validation Accuracy")
    ax.legend()
    ax.set_title('accuracy')
    ax.grid(True)


plot_loss_accuracy(history)
```



Interpret the result

In the graph shown above, our training and validation loss both decrease and stabilize at low values. we also notice that theres a slight fluctuation on the validation loss this shows that the regularization work well in the data to minimize the overfitting. The training and validation accuracy also both increase and like on the cross entropy loss our validation accuracy have a slight fluctuation which demonstrate the use of regularization to prevent the model on overfitting.

Supplementary Activity

- Use the Keras "Sequential" functionality to build a new model (model_1) with the following specifications:

1. Two hidden layers.
2. First hidden layer of size 400 and second of size 300
3. Dropout of .4 at each layer
4. How many parameters does your model have? How does it compare with the previous model?
5. Train this model for 20 epochs with RMSProp at a learning rate of .001 and a batch size of 128
6. Use at least two regularization techniques and apply it to the new model (model_2)
7. Train this model for your preferred epochs , learning rate, batch size and optimizer
8. Compare the accuracy and loss (training and validation) of model_1 and model_2

In [15]:
```
model_1 = Sequential(
    [
        Dense(400, activation='relu', input_shape=(784,)),
        Dropout(0.4),
        Dense(300, activation='relu'),
        Dropout(0.4),
        Dense(10, activation='softmax')
    ]
)
```

How many parameters does your model have? How does it compare with the previous model?

In [16]:
```
model_1.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 400)	314,000
dropout_2 (Dropout)	(None, 400)	0
dense_4 (Dense)	(None, 300)	120,300
dropout_3 (Dropout)	(None, 300)	0
dense_5 (Dense)	(None, 10)	3,010

Total params: 437,310 (1.67 MB)
Trainable params: 437,310 (1.67 MB)
Non-trainable params: 0 (0.00 B)

The total parameter for *model_1* is 437,310 while the *model* have 669,706 parameters.

In [17]:
```
model_1.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(learning_rate=0.001),
              metrics=['accuracy'])
batch_size = 128  # mini-batch with 128 examples
epochs = 20
history_1 = model_1.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
Epoch 1/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8216 - loss: 0.5611 - val_accuracy: 0.9
581 - val_loss: 0.1345
Epoch 2/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9503 - loss: 0.1636 - val_accuracy: 0.9
703 - val_loss: 0.0981
Epoch 3/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9628 - loss: 0.1253 - val_accuracy: 0.9
753 - val_loss: 0.0751
Epoch 4/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9719 - loss: 0.0971 - val_accuracy: 0.9
774 - val_loss: 0.0781
Epoch 5/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9721 - loss: 0.0908 - val_accuracy: 0.9
784 - val_loss: 0.0740
Epoch 6/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9769 - loss: 0.0757 - val_accuracy: 0.9
796 - val_loss: 0.0698
Epoch 7/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9789 - loss: 0.0676 - val_accuracy: 0.9
788 - val_loss: 0.0707
Epoch 8/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9807 - loss: 0.0643 - val_accuracy: 0.9
836 - val_loss: 0.0633
Epoch 9/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9824 - loss: 0.0589 - val_accuracy: 0.9
818 - val_loss: 0.0706
Epoch 10/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9830 - loss: 0.0570 - val_accuracy: 0.9
821 - val_loss: 0.0711
Epoch 11/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9826 - loss: 0.0561 - val_accuracy: 0.9
834 - val_loss: 0.0639
Epoch 12/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9846 - loss: 0.0509 - val_accuracy: 0.9
827 - val_loss: 0.0687
Epoch 13/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9847 - loss: 0.0498 - val_accuracy: 0.9
828 - val_loss: 0.0706
Epoch 14/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9864 - loss: 0.0483 - val_accuracy: 0.9
836 - val_loss: 0.0689
Epoch 15/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9870 - loss: 0.0428 - val_accuracy: 0.9
846 - val_loss: 0.0665
Epoch 16/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9865 - loss: 0.0450 - val_accuracy: 0.9
827 - val_loss: 0.0734
Epoch 17/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9875 - loss: 0.0420 - val_accuracy: 0.9
850 - val_loss: 0.0668
Epoch 18/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9884 - loss: 0.0372 - val_accuracy: 0.9
831 - val_loss: 0.0768
Epoch 19/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9888 - loss: 0.0366 - val_accuracy: 0.9
830 - val_loss: 0.0877
Epoch 20/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9894 - loss: 0.0381 - val_accuracy: 0.9
826 - val_loss: 0.0785
```

Use at least two regularization techniques and apply it to the new model (model_2)

In [36]:

```python
# Define the new model with L2 regularization
from keras.regularizers import l2

model_2 = Sequential([
    Dense(64, activation='relu', kernel_regularizer=l2(0.01), input_shape=(784,)),
    Dropout(0.5),
    Dense(64, activation='relu', kernel_regularizer=l2(0.01)),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])

# Summary of the model
model_2.summary()
```

```
c:\Python312\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_6"

Layer (type)	Output Shape	Param #
dense_19 (Dense)	(None, 64)	50,240
dropout_12 (Dropout)	(None, 64)	0
dense_20 (Dense)	(None, 64)	4,160
dropout_13 (Dropout)	(None, 64)	0
dense_21 (Dense)	(None, 10)	650

Total params: 55,050 (215.04 KB)

Trainable params: 55,050 (215.04 KB)

Non-trainable params: 0 (0.00 B)

Train this model for your preferred epochs , learning rate, batch size and optimizer

In [38]:

```python
from keras.optimizers import Adam
model_2.compile(loss='categorical_crossentropy',
              optimizer=Adam(learning_rate=0.001),
              metrics=['accuracy'])
batch_size = 128  # mini-batch with 128 examples
epochs = 40
history_2 = model_2.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
Epoch 1/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.5320 - loss: 2.2432 - val_accuracy: 0.9
048 - val_loss: 0.6897
Epoch 2/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8178 - loss: 0.9194 - val_accuracy: 0.9
123 - val_loss: 0.5932
Epoch 3/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8404 - loss: 0.8000 - val_accuracy: 0.9
196 - val_loss: 0.5291
Epoch 4/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8514 - loss: 0.7518 - val_accuracy: 0.9
197 - val_loss: 0.5069
Epoch 5/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8558 - loss: 0.7276 - val_accuracy: 0.9
248 - val_loss: 0.4895
Epoch 6/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8578 - loss: 0.7084 - val_accuracy: 0.9
276 - val_loss: 0.4681
Epoch 7/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8656 - loss: 0.6815 - val_accuracy: 0.9
299 - val_loss: 0.4547
Epoch 8/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8662 - loss: 0.6707 - val_accuracy: 0.9
310 - val_loss: 0.4401
Epoch 9/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8659 - loss: 0.6593 - val_accuracy: 0.9
317 - val_loss: 0.4385
Epoch 10/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8702 - loss: 0.6437 - val_accuracy: 0.9
331 - val_loss: 0.4246
Epoch 11/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8733 - loss: 0.6323 - val_accuracy: 0.9
362 - val_loss: 0.4133
Epoch 12/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8735 - loss: 0.6262 - val_accuracy: 0.9
348 - val_loss: 0.4095
Epoch 13/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8745 - loss: 0.6239 - val_accuracy: 0.9
385 - val_loss: 0.4034
Epoch 14/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8794 - loss: 0.6106 - val_accuracy: 0.9
353 - val_loss: 0.4016
Epoch 15/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8761 - loss: 0.6123 - val_accuracy: 0.9
386 - val_loss: 0.3981
Epoch 16/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8774 - loss: 0.6013 - val_accuracy: 0.9
376 - val_loss: 0.3916
Epoch 17/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8808 - loss: 0.5973 - val_accuracy: 0.9
372 - val_loss: 0.3889
Epoch 18/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8826 - loss: 0.5910 - val_accuracy: 0.9
345 - val_loss: 0.3928
Epoch 19/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8807 - loss: 0.5828 - val_accuracy: 0.9
407 - val_loss: 0.3782
Epoch 20/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8828 - loss: 0.5877 - val_accuracy: 0.9
421 - val_loss: 0.3746
Epoch 21/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8833 - loss: 0.5748 - val_accuracy: 0.9
414 - val_loss: 0.3664
Epoch 22/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8837 - loss: 0.5804 - val_accuracy: 0.9
```

```
427 - val_loss: 0.3612
Epoch 23/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8867 - loss: 0.5645 - val_accuracy: 0.9
408 - val_loss: 0.3634
Epoch 24/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8834 - loss: 0.5674 - val_accuracy: 0.9
344 - val_loss: 0.3805
Epoch 25/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8819 - loss: 0.5775 - val_accuracy: 0.9
420 - val_loss: 0.3576
Epoch 26/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8873 - loss: 0.5624 - val_accuracy: 0.9
415 - val_loss: 0.3667
Epoch 27/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8837 - loss: 0.5585 - val_accuracy: 0.9
392 - val_loss: 0.3670
Epoch 28/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8860 - loss: 0.5575 - val_accuracy: 0.9
438 - val_loss: 0.3594
Epoch 29/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8863 - loss: 0.5616 - val_accuracy: 0.9
429 - val_loss: 0.3564
Epoch 30/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8830 - loss: 0.5641 - val_accuracy: 0.9
421 - val_loss: 0.3644
Epoch 31/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8881 - loss: 0.5461 - val_accuracy: 0.9
430 - val_loss: 0.3522
Epoch 32/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8875 - loss: 0.5532 - val_accuracy: 0.9
415 - val_loss: 0.3540
Epoch 33/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8896 - loss: 0.5499 - val_accuracy: 0.9
434 - val_loss: 0.3530
Epoch 34/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8858 - loss: 0.5535 - val_accuracy: 0.9
467 - val_loss: 0.3501
Epoch 35/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8864 - loss: 0.5574 - val_accuracy: 0.9
392 - val_loss: 0.3621
Epoch 36/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8895 - loss: 0.5510 - val_accuracy: 0.9
449 - val_loss: 0.3488
Epoch 37/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8881 - loss: 0.5441 - val_accuracy: 0.9
443 - val_loss: 0.3501
Epoch 38/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8894 - loss: 0.5417 - val_accuracy: 0.9
418 - val_loss: 0.3509
Epoch 39/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8907 - loss: 0.5394 - val_accuracy: 0.9
367 - val_loss: 0.3613
Epoch 40/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8875 - loss: 0.5456 - val_accuracy: 0.9
457 - val_loss: 0.3421
```

Compare the accuracy and loss (training and validation) of model_1 and model_2

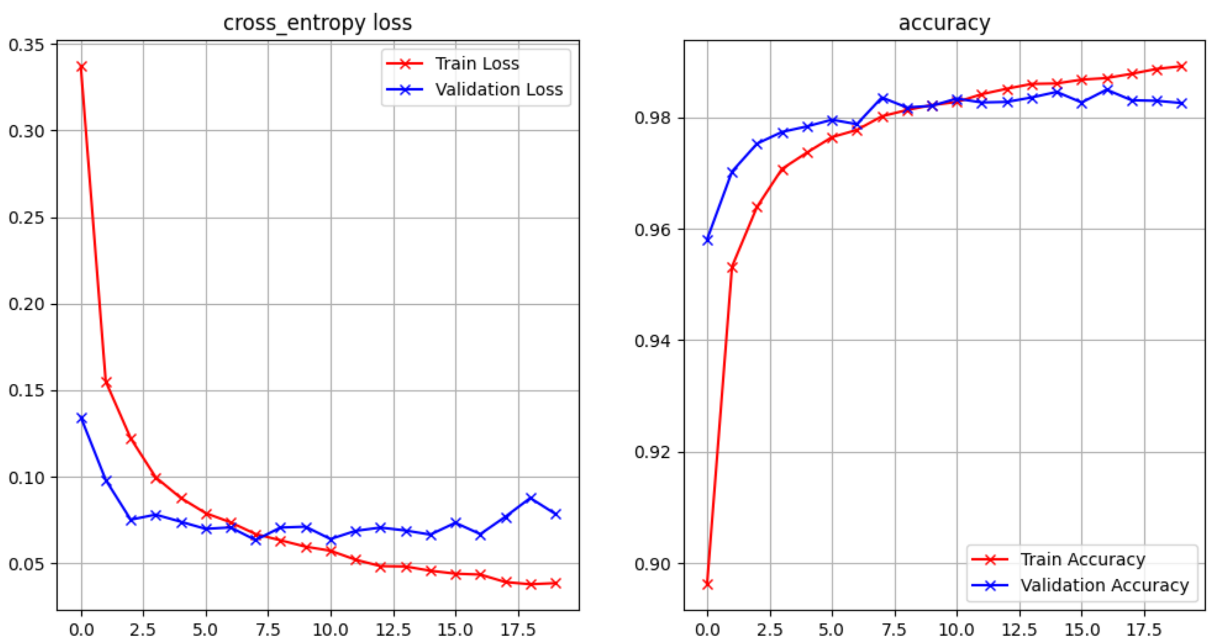
In [39]:
```
score = model_1.evaluate(x_test, y_test, verbose=0)
score_2 = model_2.evaluate(x_test, y_test, verbose=0)
print('Model 1 Performance')
print('Test loss:', score[0])
print('Test accuracy:', score[1])
print('Model 2 Performance')
print('Test loss:', score_2[0])
print('Test accuracy:', score_2[1])
```

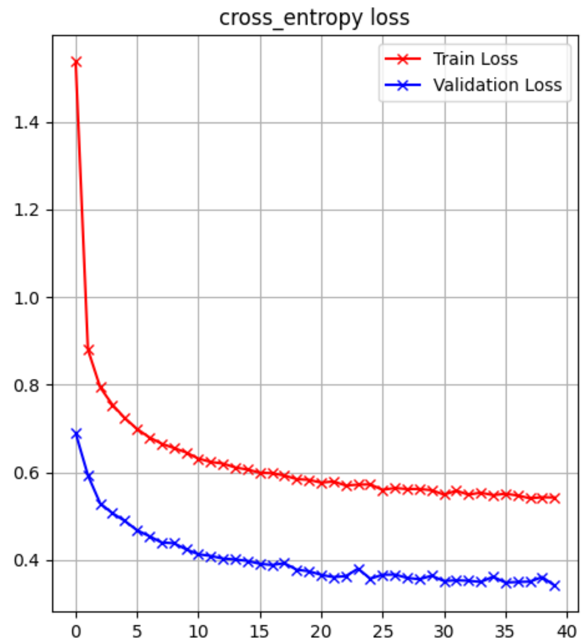
```
Model 1 Performance
Test loss: 0.07852204889059067
Test accuracy: 0.9825999736785889
Model 2 Performance
Test loss: 0.3421157896518707
Test accuracy: 0.9456999897956848
```

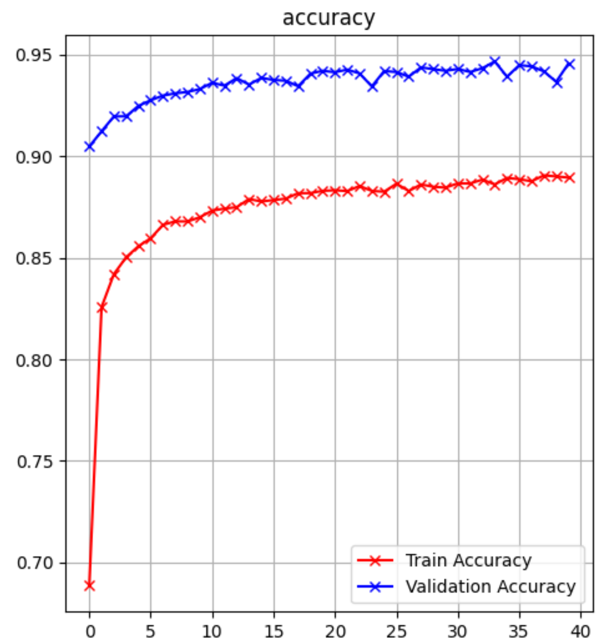
In [40]:
```
def plot_loss_accuracy(history):
    fig = plt.figure(figsize=(12, 6))
    ax = fig.add_subplot(1, 2, 1)
    ax.plot(history.history["loss"],'r-x', label="Train Loss")
    ax.plot(history.history["val_loss"],'b-x', label="Validation Loss")
    ax.legend()
    ax.set_title('cross_entropy loss')
    ax.grid(True)


    ax = fig.add_subplot(1, 2, 2)
    ax.plot(history.history["accuracy"],'r-x', label="Train Accuracy")
    ax.plot(history.history["val_accuracy"],'b-x', label="Validation Accuracy")
    ax.legend()
    ax.set_title('accuracy')
    ax.grid(True)


plot_loss_accuracy(history_1)
plot_loss_accuracy(history_2)
```







Conclusion

In this activity we learn the importance of regularization in minimizing the overfitting of the model. In this activity we make a 2 different model with different learning rate, epoch and optimizer. We learn that optimizer also affect the performance of the model by reducing the model to learn futher. In the two model we our performance in training and validation are very different because of the parameters we set during the training. In this activity Model 1 got an accuract 0.98 and loss of 0.07 which indicate a better performance in comparison to model 2 which has 0.94 accuracy and 0.34 loss. This can be explain further in our graph shown above. We see that at much higher epoch our model train and validation loss and accuracy are distant from each other compare to the much smaller epoch of 20 which give our model a higher performance.