Implementation of the gradient for the un-regularized neural network

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets

from sklearn import metrics
from sklearn import decomposition
from sklearn import manifold
import matplotlib.pyplot as plt
import numpy as np

import copy
import random
import time

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
# torch.backends.cudnn.deterministic = True

ROOT = '.data'

train_data = datasets.MNIST(root=ROOT, train=True, download=True)

# normalize data: make data to have mean of zero and a standard
deviation of one.
mean = train_data.data.float().mean() / 255
std = train_data.data.float().std() / 255

print(f'Calculated mean: {mean}')
print(f'Calculated std: {std}')

# Data augmentation: manipulating the available training data in a way
that artifically creates more training
# examples (randomely rotating, adding padding around the image)
augmented data will be transformed to a tensor and
# normalize

train_transforms = transforms.Compose([
    transforms.RandomRotation(5, fill=(0,)),
```

```python
    transforms.RandomCrop(28, padding=2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[mean], std=[std])])

test_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[mean], std=[std])
])

# Load the train and test data with the relevant defined transforms
train_data = datasets.MNIST(root=ROOT,
                            train=True,
                            download=True,
                            transform=train_transforms)

test_data = datasets.MNIST(root=ROOT,
                           train=False,
                           download=True,
                           transform=test_transforms)
# Length of datasets
print(f'Number of training examples: {len(train_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Calculated mean: 0.13066047430038452
Calculated std: 0.30810779333114624
Number of training examples: 60000
Number of testing examples: 10000
```

```python
# Sample image visualization
def plot_images(N_IMAGES, input_data):  # input_data = train_data or
test_data or va
    images = [image for image, label in [input_data[i] for i in
range(N_IMAGES)]]
    n_images = len(images)

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure()
    for i in range(rows * cols):
        ax = fig.add_subplot(rows, cols, i + 1)
        ax.imshow(images[i].view(28, 28).cpu().numpy(), cmap='bone')
        ax.axis('off')


# Creating a validation data for a proxy test to check how model
performs
VALID_RATIO = 0.9

num_train_examples = int(len(train_data) * VALID_RATIO)
```

```
num_valid_examples = len(train_data) - num_train_examples

# Take a random 10% of the training set to use as a validation set
train_data, valid_data = data.random_split(train_data,
                                           [num_train_examples,
num_valid_examples])

# Check number of examples for each portions
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 54000
Number of validation examples: 6000
Number of testing examples: 10000
```

```
# Replace the validation set's transform by overwriting it with
previously built test transforms
valid_data = copy.deepcopy(valid_data)  # To prevent changing of
default transforms of other training data
valid_data.dataset.transform = test_transforms

BATCH_SIZE = 100

train_iterator = data.DataLoader(train_data,
                                 shuffle=True,
                                 batch_size=BATCH_SIZE)

valid_iterator = data.DataLoader(valid_data,
                                 batch_size=BATCH_SIZE)

test_iterator = data.DataLoader(test_data,
                                batch_size=BATCH_SIZE)


class Multilayer_Perceptron(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.input_fc = nn.Linear(input_dim, 250)
        self.hidden_fc = nn.Linear(250, 100)
        self.output_fc = nn.Linear(100, output_dim)

    def forwad_propagation(self, x):  # x:  input tensor to the
network

        # x = [batch_size, height, width]
        batch_size = x.shape[0]
        # transform to
```

```
        x = x.view(batch_size, -1)  # reshape the tensor to x = [batch
size, height * width] -
        # 1 is when not sure about number of rows

        # First neural layer
        nn_layer_1 = self.input_fc(x)
        nn_layer_act_func_1 = F.relu(nn_layer_1)

        # Second or Hidden neural layer
        nn_layer_2 = self.hidden_fc(nn_layer_act_func_1)
        nn_layer_act_func_2 = F.relu(nn_layer_2)

        # Output layer or prediction layer
        # y_predict_layer = [batch_size_output_dim]
        y_predict_layer = self.output_fc(nn_layer_act_func_2)

        return y_predict_layer, nn_layer_act_func_2


# Define Multilayer perceptron model by creating an instance of it and
setting the correct input and output dimensions.

INPUT_DIM = 28 * 28
OUTPUT_DIM = 10

model = Multilayer_Perceptron(INPUT_DIM, OUTPUT_DIM)


# Calculate the number of trainable parameters (weights and biases)
def count_parameters(mlp_model):
    return sum(p.numel() for p in mlp_model.parameters() if
p.requires_grad)


# Calculate and print number of trainable parameters
print(f'The model has {count_parameters(model):,} trainable
parameters')

# Optimizer
optimizer = optim.Adam(model.parameters())
# Cost function
criterion = nn.CrossEntropyLoss()

# define device to put model and data, by defult it is GPU or else CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Put the model and cost function in the defined device
model = model.to(device)
criterion = criterion.to(device)
```

```python
# Calculate the accuracy of the model
def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim=True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc


'''
Define the training loop function for the model
The training loop will perform following tasks:
    * put the model into train mode
    * iterate over the data loader, returning batches of (image,
label)
    * place the batch on to GPU, if not available on CPU
    * clear the gradients calculated from the last batch
    * pass a batch of images, x, through to model to get predictions,
y_pred
    * calculate the loss between the predictions and the actual labels
    * calculate the accuracy between our predictions and the actual
labels
    * calculate the gradients of each parameter
    * update the parameters by taking an optimizer step
    * update metrics
'''


def train(model, iterator, optimizer, criterion, device):
    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in iterator:
        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model.forwad_propagation(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()
```

```python
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)


'''
The evaluation loop is similar to the training loop bu serves
different purposes performing following:
    * put to model into evaluation mode with model.eval()
    * wrap the iterations inside a with torch.no_grad() to make sure
gradients are not calculated in evaluation step
    * do not calculate gradients as we are not updating parameters
    * do not take an optimizer step as we are not calculating
gradients
'''


def evaluate(model, iterator, criterion, device):
    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():
        for (x, y) in iterator:
            x = x.to(device)
            y = y.to(device)

            y_pred, A = model.forwad_propagation(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)


# Define a function to tell how long an epoch took
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
```

```
    return elapsed_mins, elapsed_secs
```

```
The model has 222,360 trainable parameters
```

```
# Model training
best_valid_loss = float('inf')

EPOCHS = 10

for epoch in range(EPOCHS):

    start_time = time.monotonic()

    train_loss, train_acc = train(model, train_iterator, optimizer,
criterion, device)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion,
device)

    # Make sure always saves the set of parameters that has the best
validation loss (validation accuracy)
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut1-model.pt')

    end_time = time.monotonic()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    print(f'Epoch: {epoch + 1:02} | Epoch Time: {epoch_mins}m
{epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc *
100:.2f}%')
    print(f'\t Valid Loss: {valid_loss:.3f} |  Valid Acc: {valid_acc *
100:.2f}%')

# Test the model
# Afterwards, load the parameters of the model that achieved the best
validation loss
# Then use this to evaluate our model on the test set.

model.load_state_dict(torch.load('tut1-model.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion,
device)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc * 100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 0m 25s
	Train Loss: 0.462 | Train Acc: 85.66%
	 Valid Loss: 0.166 |  Valid Acc: 95.02%
Epoch: 02 | Epoch Time: 0m 25s
```

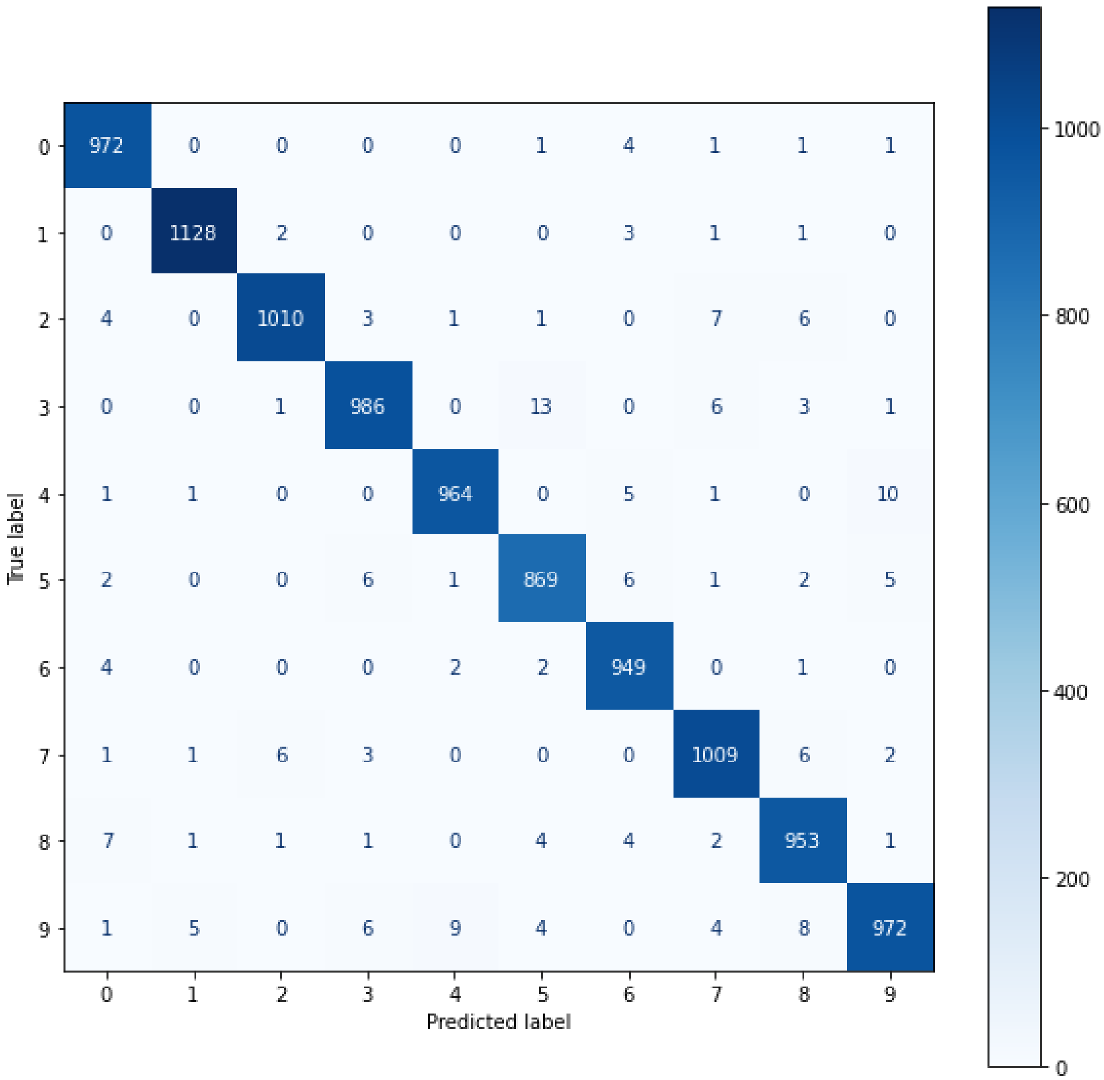
```
      Train Loss: 0.184 | Train Acc: 94.34%
       Valid Loss: 0.117 |  Valid Acc: 96.45%
Epoch: 03 | Epoch Time: 0m 25s
      Train Loss: 0.145 | Train Acc: 95.51%
       Valid Loss: 0.104 |  Valid Acc: 96.92%
Epoch: 04 | Epoch Time: 0m 25s
      Train Loss: 0.125 | Train Acc: 96.14%
       Valid Loss: 0.092 |  Valid Acc: 97.10%
Epoch: 05 | Epoch Time: 0m 25s
      Train Loss: 0.112 | Train Acc: 96.43%
       Valid Loss: 0.078 |  Valid Acc: 97.57%
Epoch: 06 | Epoch Time: 0m 25s
      Train Loss: 0.101 | Train Acc: 96.85%
       Valid Loss: 0.079 |  Valid Acc: 97.48%
Epoch: 07 | Epoch Time: 0m 25s
      Train Loss: 0.093 | Train Acc: 97.11%
       Valid Loss: 0.083 |  Valid Acc: 97.45%
Epoch: 08 | Epoch Time: 0m 25s
      Train Loss: 0.091 | Train Acc: 97.20%
       Valid Loss: 0.084 |  Valid Acc: 97.63%
Epoch: 09 | Epoch Time: 0m 25s
      Train Loss: 0.083 | Train Acc: 97.39%
       Valid Loss: 0.066 |  Valid Acc: 98.03%
Epoch: 10 | Epoch Time: 0m 25s
      Train Loss: 0.081 | Train Acc: 97.41%
       Valid Loss: 0.063 |  Valid Acc: 98.00%
Test Loss: 0.057 | Test Acc: 98.12%
```

```python
# Examining the model with simple exploratory
# This function will return input image and model prediction output
with ground truth
def get_predictions(model, iterator, device):
    model.eval()

    images = []
    labels = []
    probs = []

    with torch.no_grad():
        for (x, y) in iterator:
            x = x.to(device)

            y_pred, _ = model.forwad_propagation(x)

            y_prob = F.softmax(y_pred, dim=-1)
            top_pred = y_prob.argmax(1, keepdim=True)

            images.append(x.cpu())
            labels.append(y.cpu())
```

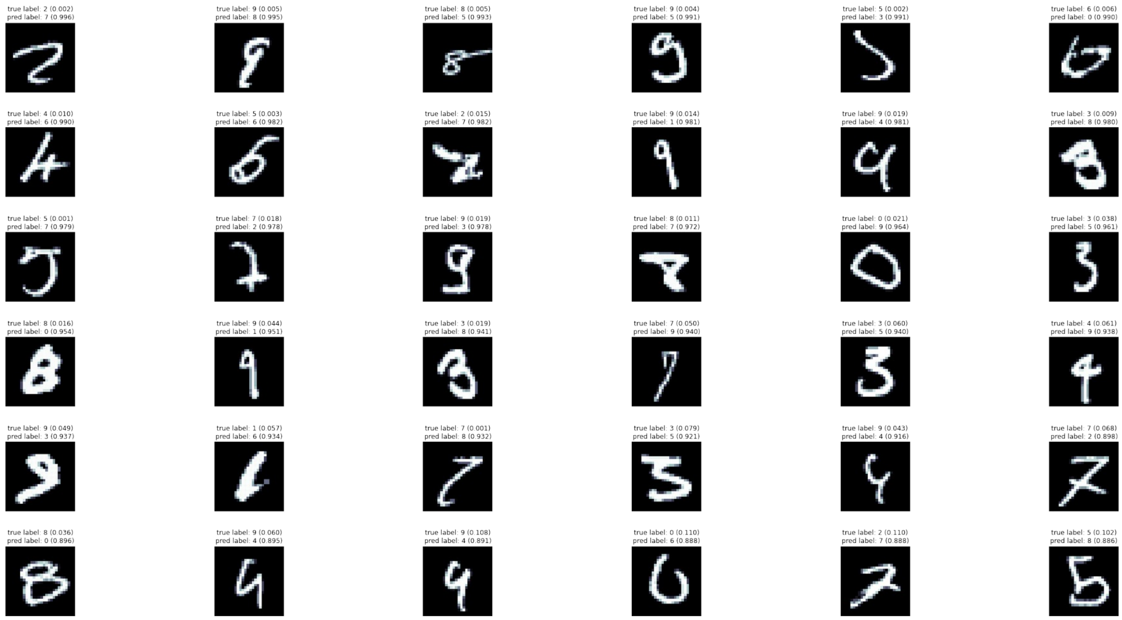
```python
            probs.append(y_prob.cpu())

    images = torch.cat(images, dim=0)
    labels = torch.cat(labels, dim=0)
    probs = torch.cat(probs, dim=0)

    return images, labels, probs


# Getting the predictions
images, labels, probs = get_predictions(model, test_iterator, device)
# It can get these predictions or prediction labels  and, by taking
the index of the highest predicted probability.
pred_labels = torch.argmax(probs, 1)


# Develop confusion matrix from the actual labels and the predicted
labels
def plot_confusion_matrix(labels, pred_labels):
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(1, 1, 1)
    cm = metrics.confusion_matrix(labels, pred_labels)
    cm = metrics.ConfusionMatrixDisplay(cm, display_labels=range(10))
    cm.plot(values_format='d', cmap='Blues', ax=ax)
    plt.savefig('confusion_matrix.png')
    plt.show()


plot_confusion_matrix(labels, pred_labels)

# Check whether predicted labels and actual labesl matches or no
corrects = torch.eq(labels, pred_labels)
# FInd out incorrectly classified examples into an array
incorrect_examples = []

for image, label, prob, correct in zip(images, labels, probs,
corrects):
    if not correct:
        incorrect_examples.append((image, label, prob))

incorrect_examples.sort(reverse=True, key=lambda x: torch.max(x[2],
dim=0).values)


# Plot the incorrectly predicted images along with how confident they
were on the actual label
# Then see how confident they were at the incorrect label.

def plot_most_incorrect(incorrect, n_images):
```

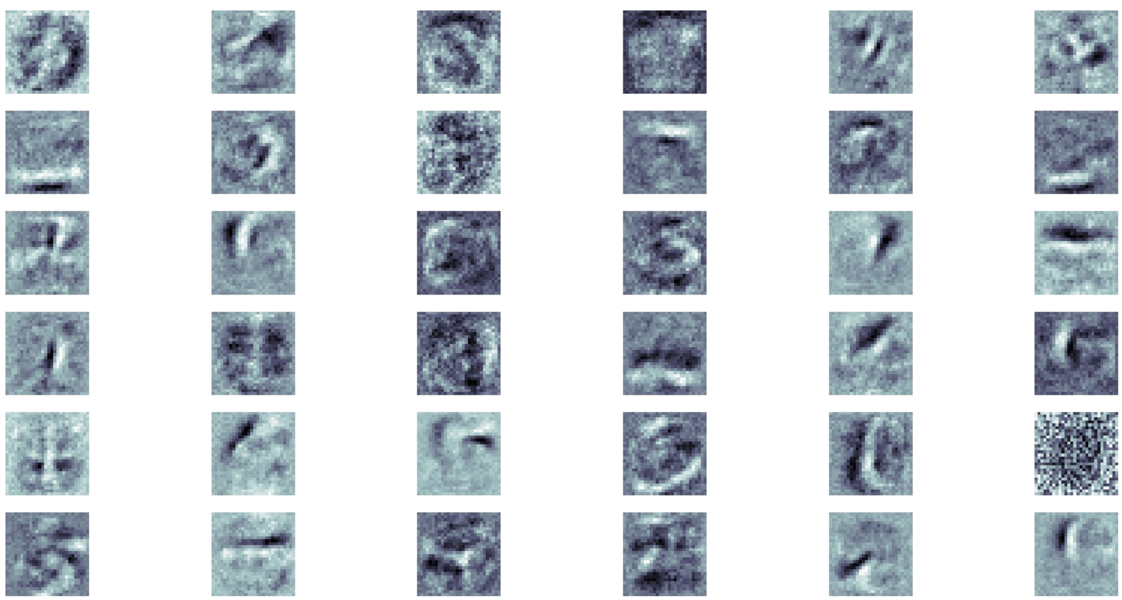
```python
    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize=(40, 20))
    for i in range(rows * cols):
        ax = fig.add_subplot(rows, cols, i + 1)
        image, true_label, probs = incorrect[i]
        true_prob = probs[true_label]
        incorrect_prob, incorrect_label = torch.max(probs, dim=0)
        ax.imshow(image.view(28, 28).cpu().numpy(), cmap='bone')
        ax.set_title(f'true label: {true_label} ({true_prob:.3f})\n' \
                     f'pred label: {incorrect_label}
({incorrect_prob:.3f})')
        ax.axis('off')
    fig.subplots_adjust(hspace=0.5)
    plt.savefig('most_incorrect.png')
    plt.show()


# Try with 30 incorrectly classified image compared with ground truth
and see how confident are the predections
N_IMAGES = 40
plot_most_incorrect(incorrect_examples, N_IMAGES)


# For more understanding it can get the output and intermediate
representations from the model and try to visualize them
def get_representations(model, iterator, device):
    model.eval()

    outputs = []
    intermediates = []
    labels = []

    with torch.no_grad():
        for (x, y) in iterator:
            x = x.to(device)

            y_pred, h = model.forwad_propagation(x)

            outputs.append(y_pred.cpu())
            intermediates.append(h.cpu())
            labels.append(y)

    outputs = torch.cat(outputs, dim=0)
    intermediates = torch.cat(intermediates, dim=0)
    labels = torch.cat(labels, dim=0)

    return outputs, intermediates, labels
```

```python
# Get representations
outputs, intermediates, labels = get_representations(model,
train_iterator, device)

# Output representations from the ten dimensional output layer,
reduced down to two dimensions (10-dim)
output_pca_data = get_pca(outputs)

def plot_weights(weights, n_weights):
    rows = int(np.sqrt(n_weights))
    cols = int(np.sqrt(n_weights))

    fig = plt.figure(figsize=(40, 20))
    for i in range(rows * cols):
        ax = fig.add_subplot(rows, cols, i + 1)
        ax.imshow(weights[i].view(28, 28).cpu().numpy(), cmap='bone')
        ax.axis('off')

    plt.savefig('weights_visualization.png')
    plt.show()

# Plotting 40 weights
N_WEIGHTS = 40
weights = model.input_fc.weight.data
plot_weights(weights, N_WEIGHTS)
```



True label
Predicted label
0
1
2
3
4
5
6
7
8
9
972 0 0 0 0 1 4 1 1 1
0 1128 2 0 0 0 3 1 1 0
4 0 1010 3 1 1 0 7 6 0
0 0 1 986 0 13 0 6 3 1
1 1 0 0 964 0 5 1 0 10
2 0 0 6 1 869 6 1 2 5
4 0 0 0 2 2 949 0 1 0
1 1 6 3 0 0 0 1009 6 2
7 1 1 1 0 4 4 2 953 1
1 5 0 6 9 4 0 4 8 972
1000
800
600
400
200
0





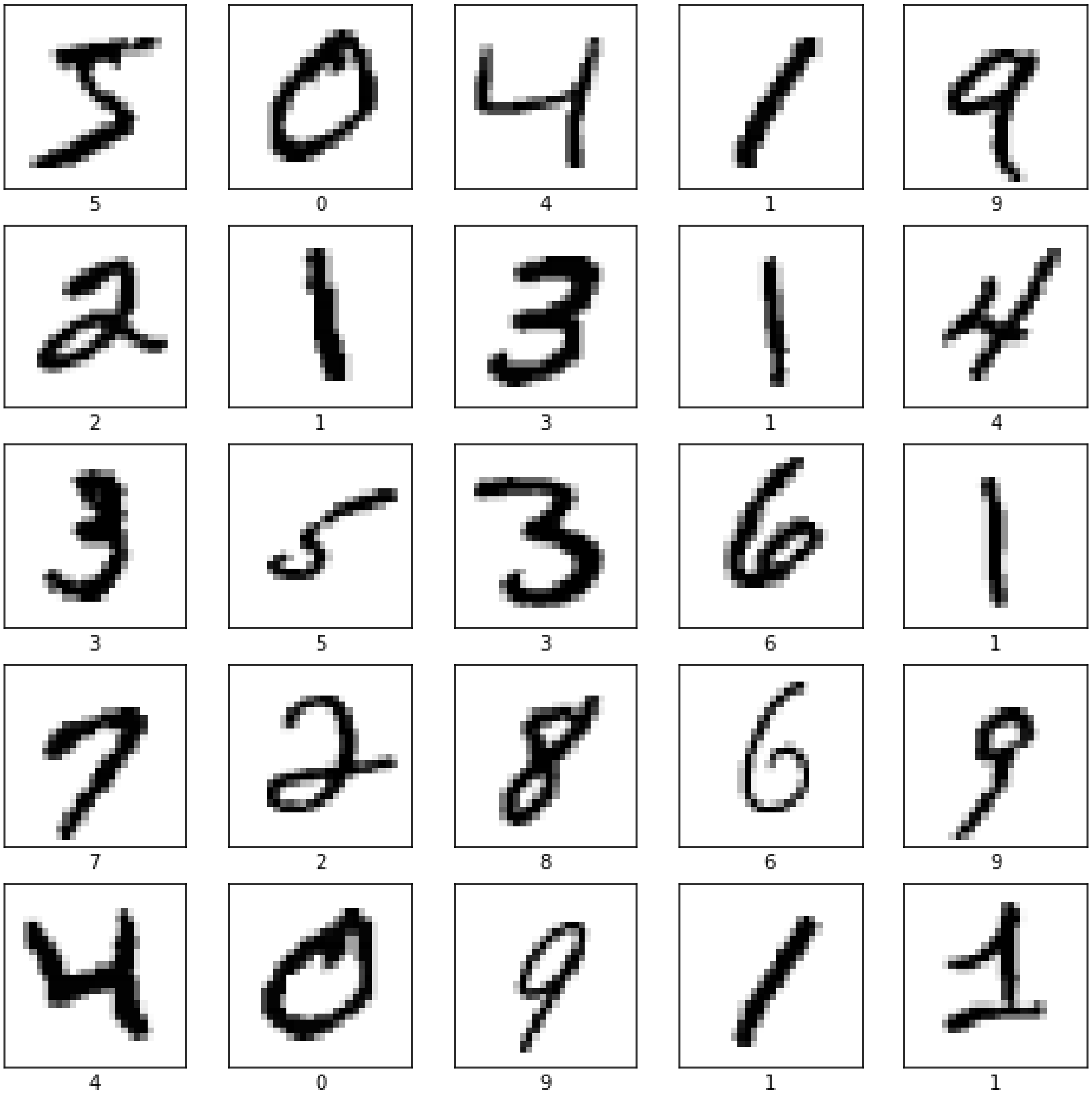
Implementation of the gradient for the regularized neural network

```
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sn
import numpy as np
import pandas as pd
import math
import datetime
import platform

print('Python version:', platform.python_version())
```

```
print('Tensorflow version:', tf.__version__)
print('Keras version:', tf.keras.__version__)

mnist_dataset = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist_dataset.load_data()
```

```
Python version: 3.7.12
Tensorflow version: 2.7.0
Keras version: 2.7.0
```

```
numbers_to_display = 25
num_cells = math.ceil(math.sqrt(numbers_to_display))
plt.figure(figsize=(10,10))
for i in range(numbers_to_display):
    plt.subplot(num_cells, num_cells, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    plt.xlabel(y_train[i])
plt.show()
```




```
(_, IMAGE_WIDTH, IMAGE_HEIGHT) = x_train.shape
IMAGE_CHANNELS = 1

print('IMAGE_WIDTH:', IMAGE_WIDTH);
print('IMAGE_HEIGHT:', IMAGE_HEIGHT);
print('IMAGE_CHANNELS:', IMAGE_CHANNELS);

x_train_with_chanels = x_train.reshape(
    x_train.shape[0],
    IMAGE_WIDTH,
    IMAGE_HEIGHT,
    IMAGE_CHANNELS
)

x_test_with_chanels = x_test.reshape(
    x_test.shape[0],
```

```
    IMAGE_WIDTH,
    IMAGE_HEIGHT,
    IMAGE_CHANNELS
)
```

```
IMAGE_WIDTH: 28
IMAGE_HEIGHT: 28
IMAGE_CHANNELS: 1
```

```
print('x_train_with_chanels:', x_train_with_chanels.shape)
print('x_test_with_chanels:', x_test_with_chanels.shape)
```

```
x_train_with_chanels: (60000, 28, 28, 1)
x_test_with_chanels: (10000, 28, 28, 1)
```

```
x_train_normalized = x_train_with_chanels / 255
x_test_normalized = x_test_with_chanels / 255
```

```
model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Convolution2D(
    input_shape=(IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_CHANNELS),
    kernel_size=5,
    filters=8,
    strides=1,
    activation=tf.keras.activations.relu,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))

model.add(tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2),
    strides=(2, 2)
))

model.add(tf.keras.layers.Convolution2D(
    kernel_size=5,
    filters=16,
    strides=1,
    activation=tf.keras.activations.relu,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))

model.add(tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2),
    strides=(2, 2)
))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(
    units=128,
```

```
    activation=tf.keras.activations.relu
));

model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.Dense(
    units=10,
    activation=tf.keras.activations.softmax,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 24, 24, 8)         208

 max_pooling2d (MaxPooling2D  (None, 12, 12, 8)        0
 )

 conv2d_1 (Conv2D)           (None, 8, 8, 16)          3216

 max_pooling2d_1 (MaxPooling  (None, 4, 4, 16)         0
 2D)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 128)               32896

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 37,610
Trainable params: 37,610
Non-trainable params: 0
_________________________________________________________________
```

```
!pip3 install ann_visualizer
!sudo apt-get install graphviz && pip3 install graphviz
from ann_visualizer.visualize import ann_viz;
#Build your model here
ann_viz(model, title = "Regularized Net")
```

```
Requirement already satisfied: ann_visualizer in
/usr/local/lib/python3.7/dist-packages (2.5)
Reading package lists... Done
Building dependency tree
```

```
Reading state information... Done
graphviz is already the newest version (2.40.1-2).
0 upgraded, 0 newly installed, 0 to remove and 37 not upgraded.
Requirement already satisfied: graphviz in
/usr/local/lib/python3.7/dist-packages (0.10.1)
```

```
adam_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

model.compile(
    optimizer=adam_optimizer,
    loss=tf.keras.losses.sparse_categorical_crossentropy,
    metrics=['accuracy']
)

log_dir=".logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M
%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

training_history = model.fit(
    x_train_normalized,
    y_train,
    epochs=10,
    validation_data=(x_test_normalized, y_test),
    callbacks=[tensorboard_callback]
)
```

```
Epoch 1/10
1875/1875 [==============================] - 32s 17ms/step - loss:
0.2065 - accuracy: 0.9360 - val_loss: 0.0566 - val_accuracy: 0.9824
Epoch 2/10
1875/1875 [==============================] - 30s 16ms/step - loss:
0.0661 - accuracy: 0.9792 - val_loss: 0.0428 - val_accuracy: 0.9859
Epoch 3/10
1875/1875 [==============================] - 30s 16ms/step - loss:
0.0485 - accuracy: 0.9846 - val_loss: 0.0355 - val_accuracy: 0.9882
Epoch 4/10
1875/1875 [==============================] - 32s 17ms/step - loss:
0.0387 - accuracy: 0.9878 - val_loss: 0.0398 - val_accuracy: 0.9865
Epoch 5/10
1875/1875 [==============================] - 32s 17ms/step - loss:
0.0344 - accuracy: 0.9890 - val_loss: 0.0295 - val_accuracy: 0.9905
Epoch 6/10
1875/1875 [==============================] - 32s 17ms/step - loss:
0.0297 - accuracy: 0.9907 - val_loss: 0.0310 - val_accuracy: 0.9898
Epoch 7/10
1875/1875 [==============================] - 32s 17ms/step - loss:
0.0250 - accuracy: 0.9921 - val_loss: 0.0302 - val_accuracy: 0.9901
Epoch 8/10
1875/1875 [==============================] - 31s 16ms/step - loss:
0.0229 - accuracy: 0.9926 - val_loss: 0.0323 - val_accuracy: 0.9901
```

```
Epoch 9/10
1875/1875 [==============================] - 31s 16ms/step - loss:
0.0215 - accuracy: 0.9929 - val_loss: 0.0312 - val_accuracy: 0.9902
Epoch 10/10
1875/1875 [==============================] - 32s 17ms/step - loss:
0.0184 - accuracy: 0.9938 - val_loss: 0.0317 - val_accuracy: 0.9908
```

```
%%capture
train_loss, train_accuracy = model.evaluate(x_train_normalized,
y_train)

print('Training loss: ', train_loss)
print('Training accuracy: ', train_accuracy)
```

```
Training loss:  0.008854968473315239
Training accuracy:  0.9970499873161316
```

```
%%capture
validation_loss, validation_accuracy =
model.evaluate(x_test_normalized, y_test)

print('Validation loss: ', validation_loss)
print('Validation accuracy: ', validation_accuracy)
```

```
Validation loss:  0.03173688054084778
Validation accuracy:  0.9908000230789185
```