CS 6910 Fundamentals of Deep Learning

Project Code

March 3, 2020

Submitted by **Team 19**

ME17B084 Nishant Prabhu, **ME17B068** Soumyadeep Mondal, **ME16B177** Saye Sharan

---

Contents

The files which execute the model and generate results are under the **Execution Scripts** section. Functions and classes imported in these scripts have been defined in various files under **Supporting Scripts** section.

1 Supporting Scripts

1.1 models.py

```
# Dependencies
import numpy as np

from losses import loss_dict
from metrics import metric_dict
losses_dict = loss_dict()
metrics_dict = metric_dict()


# This script defines the model object and all associated functions

# Available functions in <class> Network
#   set_layers : accepts a list of layers with suitable configurations, sets
 ↪  them as model's layers
#   compile : loss function, performance metric and optimizer are defined here
#   forward : forward pass for the model with current weights and biases
#   backpropagate : based on model outputs, computes the values by which each
 ↪  weight and bias parameter has to be changed
#   predict : performs forward pass on all data points passed to this function
 ↪  and returns array of predictions
#   train : calls forward, backpropagate and predict in correct order for
 ↪  specified number of epochs and validation split to
#           tune model parameters


class Network():

    def __init__(self):
        self.layers = None
        self.train_loss_history = list()
        self.train_metric_history = list()
        self.val_loss_history = list()
        self.val_metric_history = list()

    # SETUP FUNCTIONS =========================================================
     ↪  ===========================================

    def set_layers(self, layer_list):
        """
        Every element in the layer list is a layer object from the
        layers script. When the list of layers is provided, weights
        gradients associated with each layer are initialized here.
        """
        for i in range(1, len(layer_list)):
            layer_list[i].weights = np.random.normal(loc=0, scale=1.0,
             ↪  size=layer_list[i-1].units*layer_list[i].units).reshape((layer_⌋
             ↪  list[i-1].units,
             ↪  layer_list[i].units))
            layer_list[i].gradients = np.zeros((layer_list[i].units, 1))
            layer_list[i].activations = np.zeros((layer_list[i].units, 1))
            layer_list[i].bias = np.ones((layer_list[i].units, 1))
            layer_list[i].w_last_update = np.zeros(layer_list[i].weights.shape)
            layer_list[i].gss = np.zeros(layer_list[i].weights.shape)
            layer_list[i].w_history = [np.zeros(layer_list[i].weights.shape)]
            layer_list[i].g_history = [np.zeros(layer_list[i].weights.shape)]
            layer_list[i].q_vals = np.zeros(layer_list[i].weights.shape)
            layer_list[i].r_vals = np.zeros(layer_list[i].weights.shape)

        self.layers = layer_list

    def compile(self, optimizer, loss='RMSE', metric='RMSE'):
        """
        Sets the loss functions and metrics
        """
        self.loss = losses_dict[loss]
        self.metric_name = metric
        self.metrics = metrics_dict[self.metric_name]
        self.optim = optimizer

    # TRAINING FUNCTIONS ======================================================⌋
     ↪  ============================================

    def forward(self, x):
        """
        Forward pass function to obtain activations at output layer.
        """
        self.layers[0].activations = x.reshape((-1, 1))
        for i in range(1, len(self.layers)):
            self.layers[i].activations = self.layers[i].a_func.get_value(
                np.dot(self.layers[i].weights.T, self.layers[i-1].activations)
                 ↪  + self.layers[i].bias
            )

    def backpropagate(self, y, x_count):

        # For output layer
        y = y.reshape(self.layers[-1].activations.shape)

        # Operations for the output layer
        self.layers[-1].x_count = x_count + 1
        self.layers[-1].gradients =
         ↪  np.dot(self.layers[-1].a_func.grad(self.layers[-1].activations),
         ↪  self.loss(y, self.layers[-1].activations).grad())
        w_grad = np.dot(self.layers[-2].activations,
         ↪  self.layers[-1].gradients.T)
        self.layers[-1].weights += self.optim.get_update(self.layers[-1],
         ↪  w_grad)
        self.layers[-1].bias += -self.optim.lr * self.layers[-1].gradients

        # Updates are made to suitable layer attributes based on chosen
         ↪  optimizer
        if self.optim.name == 'AdaDelta':
            self.layers[-1].g_history.append(w_grad)
            self.layers[-1].w_history.append(self.optim.get_update(self.layers[
             ↪  -1],
             ↪  w_grad))
        elif self.optim.name == 'SGD':
            self.layers[-1].w_last_update =
             ↪  self.optim.get_update(self.layers[-1], w_grad)
        elif self.optim.name == 'AdaGrad':
            self.layers[-1].gss += w_grad**2

        # For other layers
        for i in np.arange(len(self.layers)-2, 0, -1):
            self.layers[i].x_count = x_count + 1
            self.layers[i].gradients =
             ↪  np.dot(self.layers[i].a_func.grad(self.layers[i].activations),
             ↪  np.dot(self.layers[i+1].weights, self.layers[i+1].gradients))
            w_grad = np.dot(self.layers[i-1].activations,
             ↪  self.layers[i].gradients.T)
            self.layers[i].weights += self.optim.get_update(self.layers[i],
             ↪  w_grad)
            self.layers[i].bias += -self.optim.lr * self.layers[i].gradients

            if self.optim.name == 'AdaDelta':
                self.layers[i].g_history.append(w_grad)
                self.layers[i].w_history.append(self.optim.get_update(self.laye
                 ↪  rs[i],
                 ↪  w_grad))
            elif self.optim.name == 'SGD':
                self.layers[i].w_last_update =
                 ↪  self.optim.get_update(self.layers[i], w_grad)
            elif self.optim.name == 'AdaGrad':
                self.layers[i].gss += w_grad**2

    def predict(self, X_test):
```

```
    """
    Performs forward pass on trained model with provided testing data.
    """
    preds = []
    for i in range(len(X_test)):
        self.forward(X_test[i])
        preds.append(self.layers[-1].activations)
    outputs = np.array(preds)
    return outputs.reshape(outputs.shape[:2])

def train(self, X, y, epochs=20, val_split=0.0, val_sets=None,
 ↪  log_frequency=1, track_epochs=[]):
    """
    This function performs forward passes and backpropagation sequentially
     ↪  for each data point
    for specified number of epochs with specified learning rate.
    """
    # This dictionary will keep track of layer outputs
    self.layer_outs = {}

    # Performs validation split (or not) based on value passed to argument
    if val_split > 0:
        print('\nTraining on {} samples, validating on {}
         ↪  samples.\n'.format(
                len(X) - int(val_split*len(X)),
                int(val_split*len(X))
            ))
    elif val_sets is not None:
        print('\nTraining on {} samples, validating on {}
         ↪  samples.\n'.format(
                len(X),
                len(val_sets[0])
            ))
    else:
        print('\nTraining on {} samples.\n'.format(
            len(X)
        ))

    # Loop over each epoch
    for epoch in range(epochs):

        if epoch+1 in track_epochs:
            layer_output_dict = {}
            for i in range(1, len(self.layers)):
                layer_output_dict.update({self.layers[i].label: list()})

        if val_split > 0:
```

```
val_index = np.random.choice([i for i in range(len(X))],
 ↪  size=int(val_split * len(X)), replace=False)
X_val, y_val = X[val_index, :], y[val_index, :]
X_train = X[[i for i in range(len(X)) if i not in val_index], :]
y_train = y[[i for i in range(len(y)) if i not in val_index], :]

for i in range(len(X_train)):
    self.forward(X_train[i])
    self.backpropagate(y_train[i], i)

# This section checks whether the current epoch's layer output
 ↪  is needed
# If yes, uses a meshgrid to generate sample data and obtains
 ↪  outputs
# Updates appropriate locations
if epoch+1 in track_epochs:
    x1, x2 = np.arange(0, 1, 0.01), np.arange(0, 1, 0.01)
    x1, x2 = np.meshgrid(x1, x2)
    data = np.c_[x1.ravel(), x2.ravel()]
    for i in range(len(data)):
        self.forward(data[i])
        for j in range(1, len(self.layers)):
            layer_output_dict[self.layers[j].label].append(self ⌋
             ↪  .layers[j].activations.reshape((1,
             ↪  -1)).tolist())
    self.layer_outs.update({epoch+1: layer_output_dict})

y_train_pred = self.predict(X_train)
y_val_pred = self.predict(X_val)

# For every logging frequency step, print the state of the model
if epoch % log_frequency == 0:
    print("Epoch {}/{} \t Train Loss : {:.6f} - Train {} :
     ↪  {:.6f} \t Val Loss : {:.6f} - Val {} : {:.6f}".format(
        epoch,
        epochs,
        self.loss(y_train, y_train_pred).get_value(),
        self.metric_name,
        self.metrics(y_train, y_train_pred),
        self.loss(y_val, y_val_pred).get_value(),
        self.metric_name,
        self.metrics(y_val, y_val_pred)
    ))

# Update histories with suitable values
```

```
        self.train_loss_history.append(self.loss(y_train,
         ↪  y_train_pred).get_value())
        self.train_metric_history.append(self.metrics(y_train,
         ↪  y_train_pred))
        self.val_loss_history.append(self.loss(y_val,
         ↪  y_val_pred).get_value())
        self.val_metric_history.append(self.metrics(y_val, y_val_pred))

    # If external validation data is provided, then use that instead of
     ↪  splitting training data
    elif val_sets is not None:
        X_val, y_val = val_sets[0], val_sets[1]
        X_train, y_train = X, y

        for i in range(len(X_train)):
            self.forward(X_train[i])
            self.backpropagate(y_train[i], i)

        if epoch+1 in track_epochs:
            x1, x2 = np.arange(0, 1, 0.01), np.arange(0, 1, 0.01)
            x1, x2 = np.meshgrid(x1, x2)
            data = np.c_[x1.ravel(), x2.ravel()]
            for i in range(len(data)):
                self.forward(data[i])
                for j in range(1, len(self.layers)):
                    layer_output_dict[self.layers[j].label].append(self
                     ↪  .layers[j].activations.reshape((1,
                     ↪  -1)).tolist())
            self.layer_outs.update({epoch+1: layer_output_dict})

        y_train_pred = self.predict(X_train)
        y_val_pred = self.predict(X_val)

        if epoch % log_frequency == 0:
            print("Epoch {}/{} \t Train Loss : {:.6f} - Train {} :
             ↪  {:.6f} \t Val Loss : {:.6f} - Val {} : {:.6f}".format(
                epoch,
                epochs,
                self.loss(y_train, y_train_pred).get_value(),
                self.metric_name,
                self.metrics(y_train, y_train_pred),
                self.loss(y_val, y_val_pred).get_value(),
                self.metric_name,
                self.metrics(y_val, y_val_pred)
            ))
        self.train_loss_history.append(self.loss(y_train,
         ↪  y_train_pred).get_value())
        self.train_metric_history.append(self.metrics(y_train,
         ↪  y_train_pred))
        self.val_loss_history.append(self.loss(y_val,
         ↪  y_val_pred).get_value())
        self.val_metric_history.append(self.metrics(y_val, y_val_pred))

    # If val_split = 0.0, then perform only training without validation
    else:
        for i in range(len(X)):
            self.forward(X[i])
            self.backpropagate(y[i], i)

        if epoch+1 in track_epochs:
            x1, x2 = np.arange(0, 1, 0.01), np.arange(0, 1, 0.01)
            x1, x2 = np.meshgrid(x1, x2)
            data = np.c_[x1.ravel(), x2.ravel()]
            for i in range(len(data)):
                self.forward(data[i])
                for j in range(1, len(self.layers)):
                    layer_output_dict[self.layers[j].label].append(self⌋
                        ↪  .layers[j].activations.reshape((1,
                        ↪  -1)).tolist())
            self.layer_outs.update({epoch+1: layer_output_dict})

        y_train_pred = self.predict(X)

        if epoch % log_frequency == 0:
            print("Epoch {}/{} \t Train Loss : {:.6f} - Train {} :
             ↪  {:.6f}".format(
                epoch,
                epochs,
                self.loss(y, y_train_pred).get_value(),
                self.metric_name,
                self.metrics(y, y_train_pred),
            ))

        self.train_loss_history.append(self.loss(y,
         ↪  y_train_pred).get_value())
        self.train_metric_history.append(self.metrics(y, y_train_pred))
```

1.2 activations.py

```
# Dependencies
import numpy as np


# This script defines various activation functions

# Available activations
#   Linear (NOT WORKING)
#   Sigmoid
#   ReLU (NOT WORKING)
#   Tanh
#   Softmax


class Linear():
    """
    Linear activation, returns whatever activation it gets
    """
    def __init__(self):
        pass

    def get_value(self, x):
        return x

    def grad(self, s):
        return np.identity(s.shape[0])


class Sigmoid():
    """
    Standard sigmoid activation
    beta is a weight hyperparameter
    """
    def __init__(self, c=1.0, beta=1.0):
        self.beta = beta
        self.c = c

    def get_value(self, z):
        y = np.exp(-self.beta * z)
        return (self.c / (1.0 + y))

    def grad(self, s):
        return np.identity(s.shape[0])*(1 - s)*s
```

```
class Tanh():
    """
    Tanh activation function
    """
    def __init__(self, c=1.0):
        self.c = c

    def get_value(self, z):
        return self.c * np.tanh(z)

    def grad(self, s):
        return np.identity(s.shape[0])*(1-s**2)


class ReLU():
    """
    Rectified linear unit activation
    alpha parameter can be set to non-zero value for LeakyReLU
    """
    def __init__(self):
        return

    def get_value(self, z):
        return (z * (z > 0))

    def grad(self, s):
        return np.identity(s.shape[0])*(s > 0).astype('int')


class Softmax():
    """
    Softmax activation with temperature = 1
    Use in output layers for classification problems
    """
    def __init__(self):
        pass

    def get_value(self, z):
        return np.exp(z)/np.sum(np.exp(z))

    def grad(self, s):
        return (np.identity(s.shape[0]) - s).T * s
```

1.3 layers.py

```
# Dependencies
import numpy as np


# This script defines layer objects to be added to the network object

# Available layers:
#   Input
#   Dense


class Input():
    """
    This layers accepts the data as input
    Need to figure the dimension of data that has to be passed to this
    """
    def __init__(self, units, label=None):
        self.units = units
        self.label = label
        self.activations = None
        self.bias = None

    def name(self):
        return self.label

    def size(self):
        return self.units


class Dense():
    """
    Standard fully connected layer
    """
    def __init__(self, units, activation, label=None):
        self.units = units
        self.label = label
        self.a_func = activation
        self.activations = None
        self.gradients = None
        self.bias = None
        self.weights = None
        self.sgd_update = None          # Used in SGD optimizer
        self.gss = None                 # Used in AdaGrad
        self.q_vals = None              # For Adam, expectation of gradients (q)
        self.r_vals = None              # For Adam, expectation of gradient
        ↪  squared (r)
        self.w_history = None           # To be used for AdaDelta
        self.g_history = None           # To be used for AdaDelta
        self.x_count = None             # Counter to keep track of number of
        ↪  input vectors passed

    def name(self):
        return self.label

    def size(self):
        return self.units

    def set_weights(self, weights):
        """
        Sets model weights as given set of weights, if the shape is correct.
        """
        try:
            assert weights.shape == self.weights.shape
        except:
            raise ValueError('Shape {} of weights does not match expected shape
            ↪  {}'.format(
                                weights.shape,
                                self.weights.shape
                            ))
        self.weights = weights
```

1.4 losses.py

```
# Dependencies
import numpy as np


# This script defines some loss functions

# Available loss functions
#   loss_dict : Dictionary with all losses
#   RMSE
#   CrossEntropy


def loss_dict():
    """
    For use by models.py
    """
    losses = {
        'RMSE': RMSE,
        'CrossEntropy': CrossEntropy
    }
    return losses


class RMSE():
    """
    Root mean squared error. Good for regression problems.
    """
    def __init__(self, y_true, y_pred):
        self.y_true = y_true
        self.y_pred = y_pred
        self.N = len(y_true)

    def get_value(self):
        return np.sum((self.y_true - self.y_pred)**2)/self.N

    def grad(self):
        return -(self.y_true - self.y_pred)


class CrossEntropy():
    """
    Useful for classification problems (binary or multiclass)
    """
    def __init__(self, y_true, y_pred):
        self.y_true = y_true
        self.y_pred = y_pred.reshape(y_true.shape)
        self.N = len(y_true)

    def get_value(self):
        return -np.sum(self.y_true * np.log(self.y_pred))/self.N

    def grad(self):
        return -(self.y_true/(self.y_pred + 1e-9))
```

1.5 metrics.py

```
# Dependencies
import numpy as np


# This script defines metrics which will be displayed during model training
 ↪  process

# Available metrics
#   RMSE
#   CrossEntropy
#   Accuracy


def metric_dict():
    metrics = {
        'RMSE': RMSE,
        'Crossentropy': CrossEntropy,
        'Accuracy': Accuracy
    }
    return metrics


def RMSE(y_true, y_pred):
    """
    Root mean squared error
    """
    return np.sum((y_true - y_pred)**2)/len(y_true)


def CrossEntropy(y_true, y_pred):
    """
    For classification tasks
    Expects each row of y_true and y_pred to be array of probabilities
    """
    return np.sum([-np.sum(np.multiply(y_true[i], np.log(y_pred[i]))) for i in
     ↪  range(len(y_true))])/len(y_true)


def Accuracy(y_true, y_pred):
    """
    Ratio of correct answers in y_pred with respect to y_true
    """
    return np.mean([np.argmax(y_true[i]) == np.argmax(y_pred[i]) for i in
     ↪  range(len(y_true))])
```

1.6 optimizers.py

```
# Dependencies
import numpy as np


# This script defines optimizers that will be used in training

# Available optimizers
#   SGD : Or Generalized Delta Rule. For usual Delta Rule set alpha = 0.0
#   AdaGrad : Adaptive Gradients Method
#   AdaDelta : Adaptive Gradients improved
#   Adam : Adaptive Moments


class SGD():
    def __init__(self, lr, momentum):
        self.lr = lr
        self.alpha = momentum
        self.name = 'SGD'

    def get_update(self, layer, w_grad):
        return (-self.lr * w_grad) + (self.alpha * layer.w_last_update)


class AdaGrad():
    def __init__(self, lr, epsilon=1e-2):
        self.lr = lr
        self.epsilon = epsilon
        self.name = 'AdaGrad'

    def get_update(self, layer, w_grad):
        return (-self.lr * w_grad) / (self.epsilon + np.sqrt(layer.gss))


class AdaDelta():
    def __init__(self, rho=0.9, L=50, epsilon=0.1):
        self.rho = rho
        self.L = L
        self.epsilon = epsilon

    def get_update(self, layer, w_grad):
        nmr = np.power((self.rho/self.L)*np.sum(np.power(layer.w_history[-(self ⌋
         ↪  .L+1):-1], 2))+(1-self.rho)*(layer.w_history[-1])**2,
         ↪  0.5)
        dmr = np.power((self.rho/self.L)*np.sum(np.power(layer.g_history[-(self ⌋
         ↪  .L+1):-1], 2))+(1-self.rho)*(layer.g_history[-1])**2,
         ↪  0.5)
        return (nmr/(dmr+self.epsilon)) * w_grad


class Adam():
    def __init__(self, lr, rho_1=0.9, rho_2=0.999, epsilon=1e-2):
        self.lr = lr
        self.rho_1 = rho_1
        self.rho_2 = rho_2
        self.epsilon = epsilon
        self.name = 'Adam'

    def get_update(self, layer, w_grad):
        layer.q_vals = (self.rho_1)*(layer.q_vals) + (1-self.rho_1) * w_grad
        layer.r_vals = (self.rho_2)*(layer.r_vals) + (1-self.rho_2) *
         ↪  (w_grad**2)
        q_hat = layer.q_vals/(1 - self.rho_1**layer.x_count)
        r_hat = layer.r_vals/(1 - self.rho_2**layer.x_count)

        return (-self.lr * q_hat)/(self.epsilon + r_hat**0.5)
```

1.7 plotting.py

```
# Dependencies
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns

from warnings import filterwarnings
filterwarnings(action='ignore')


# This script defines plotting functions used to
# visualize various results and intermediate conditions
# Available functions:
#   SpatialPlot : Plots the target with data points in space
#   ParityPlot : Plots target versus predictions


def SpatialPlot(X, y_true, y_pred):
    """
    [WARNING] Function assumes data is two dimensional. For any other
    dimensional data, plot will not be representative of complete situation.
    Blue dots represent actual data, red dots represent predictions.
    """
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], y_true, zdir='z', s=20, c='blue', alpha=0.5,
     ↪  depthshade=True)    # True values
    ax.scatter(X[:, 0], X[:, 1], y_pred, zdir='z', s=20, c='red', alpha=0.5,
     ↪  depthshade=True)     # Predicted values
    plt.legend(['True', 'Predicted'])
    plt.show()


def ParityPlot(y_true, y_pred):
    """
    Plots actual values versus predicted values
    Helps determine goodness of fit for regression problems
    """
    fig = plt.figure(figsize=(6, 6))
    ax = fig.add_subplot(111)
    ax.scatter(y_pred, y_true, s=30, c='red', alpha=0.6)
    x = np.linspace(*ax.get_xlim())
    ax.plot(x, x, color='black')
    plt.xlabel('Predictions')
    plt.ylabel('Target')
    plt.title('Parity plot')
    plt.grid()
    plt.show()


def CallbackPlot(model, callback='Loss'):
    """
    Plots loss history and metric history, based on callback argument.
    """
    plt.figure(figsize=(8, 8))

    if callback == 'Loss':
        plt.plot(model.train_loss_history, color='blue', alpha=1.0)
        plt.plot(model.val_loss_history, color='orange', alpha=1.0)
    elif callback == 'Metric':
        plt.plot(model.train_metric_history, color='blue', alpha=1.0)
        plt.plot(model.val_metric_history, color='orange', alpha=1.0)

    plt.grid()
    plt.legend(['Train', 'Validation'])
    plt.title('{} history'.format(callback))
    plt.ylabel('{}'.format(callback))
    plt.xlabel('Epoch')
    plt.show()


def DecisionRegionPlot(model, X, y_true, y_pred, colors=['red', 'yellow',
 ↪  'blue'], cmap=plt.cm.RdYlBu):
    """
    [WARNING]: Function assumes data is 2 dimensional
    Plots decision regions of predictions vs. scatter of actual data on a
     ↪  colormap.
    """
    n_classes = len(np.unique(y_true))
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.05, X[:, 1].max() + 0.05
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max,
     ↪  0.02))
    Z_preds = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = np.array([np.argmax(i) for i in Z_preds])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(8, 8))
    plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.6)

    for i, color in zip(range(n_classes), colors):
        idx = np.where(y_true == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, label=i, edgecolor='black',
         ↪  s=20)

    plt.title('Decision region plot')
    plt.xlabel('x_1')
    plt.ylabel('x_2')
    plt.show()


def ConfusionMatrix(y_true, y_pred):
    """
    Calculates values consisting a confusion matrix and renders on seaborn
     ↪  heatmap.
    """
    y_true = y_true.reshape((1, -1))
    y_pred = y_pred.reshape((1, -1))
    data = np.vstack((y_true, y_pred)).T
    df = pd.DataFrame(data, columns=['a', 'p'])
    confusion_matrix = pd.crosstab(df['a'], df['p'], rownames=['Actual'],
     ↪  colnames=['Predicted'])
    plt.figure(figsize=(8, 8))
    sns.heatmap(confusion_matrix, annot=True, square=True, cmap='BuGn',
     ↪  fmt='g', linewidth=1, linecolor='black')
    plt.title('Confusion Matrix')
    plt.show()


def LayerOutputPlot(model, track_epochs, save_path):
    """
    Plots outputs of hidden layers of the model as surfaces (over contour maps).
    """
    x1, x2 = np.meshgrid(np.arange(0, 1, 0.01), np.arange(0, 1, 0.01))
    for epoch in track_epochs:
        for label in model.layer_outs[epoch].keys():
            for node in
             ↪  range(np.array(model.layer_outs[epoch][label]).shape[2]):
                y_grid = np.array(model.layer_outs[epoch][label])[:, :,
                 ↪  node].reshape(x1.shape)

                fig = plt.figure(figsize=(8, 8))
                ax = fig.add_subplot(111, projection='3d')
                ax.plot_surface(x1, x2, y_grid, rstride=1, cstride=1,
                 ↪  cmap='jet')
                plt.title('Epoch {} - Layer {} - Node {}'.format(epoch, label,
                 ↪  node))
```

```
plt.savefig(save_path + 'epoch' + str(epoch) + '_' + label +
 ↪  '_node' + str(node) + '.png')
```

1.8 utils.py

```
# Dependencies
import numpy as np
import os
from random import shuffle
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import torchvision.models as models

# Some utility functions
# Available functions
#   OneHotEncoder : One-hot encodes categorical outputs
#   MinMaxScaler : Scales data by min-max rule
#   ImageFeatureExtractor : Scripts given by course TAs


class OneHotEncoder():
    """
    Converts categorical data into one hot encoded vectors.
    Returns a numpy array.
    Can perform an inverse transform to get back labels from one-hot arrays.
    """
    def __init__(self):
        pass

    def transform(self, data):
        num_classes = len(np.unique(data))
        retval = []
        for label in data:
            retval.append([1.0 if i == label else 0.0 for i in
             ↪  range(num_classes)])
        return np.array(retval).astype(np.float32)

    def inverse_transform(self, data):
        return np.array([np.argmax(i) for i in data])


class MinMaxScaler():
    """
    Returns scaled matrix such that all values along columns lie between 0 and 1
    """
    def __init__(self):
        self.maxs = None
        self.mins = None
        self.dupl = None

    def fit(self, data):
        self.dupl = data.astype(np.float64)
        self.maxs = data.max(axis=0)
        self.mins = data.min(axis=0)

    def transform(self, data):
        for col in range(self.dupl.shape[1]):
            self.dupl[:, col] = (data[:, col] - self.mins[col])/(self.maxs[col]
             ↪  - self.mins[col])
        return self.dupl

    def inverse_transform(self, data):
        for col in range(data.shape[1]):
            self.dupl[:, col] = self.mins[col] + data[:, col]*(self.maxs[col] -
             ↪  self.mins[col])
        return self.dupl

    def fit_transform(self, data):
        self.fit(data)
        return self.transform(data)


class ImFeatureExtractor():
    """
    Based on functions provided for extraction of features
    from image data.
    """
    def __init__(self):
        pass

    def generate_features(self, load_path, save_path):
        net = models.vgg16_bn(pretrained=True)
        os.mkdir(save_path)
        get_class_names = os.listdir(load_path)

        for i in get_class_names:
            new_save_path = save_path + i
            class_path = load_path + i
            img = np.load(class_path)
            arr = []

            for j in img:
                j = torch.tensor(j)
                j = j.view([-1, 3, 32, 32])
                j = F.interpolate(j, (224, 224))
                z = net.features(j)
                m = F.avg_pool2d(z, (7, 7), 1, 0)
                m = m.view([-1]).detach()
                m = np.asarray(m)
                arr.append(m)

            arr = np.asarray(arr)
            np.save(new_save_path, arr)

    def load_features(self, feature_path):
        path = feature_path
        self.class_names = os.listdir(path)
        self.label_map = {}
        val = 0
        data_points = []
        data_points_class = []

        for i in self.class_names:
            load_name = os.path.join(path, i)
            extracted_features = np.load(load_name)

            for j in extracted_features:
                data_points.append(j)
                data_points_class.append(val)

            self.label_map.update({val: i})
            val += 1

        temp = list(zip(data_points,data_points_class))
        shuffle(temp)

        data_points, data_points_class = zip(*temp)
        data_points = np.asanyarray(data_points)
        return data_points, data_points_class, self.label_map

    def get_features(self, load_path, save_path):
        """
        Combines the two functions above this.
        """
        print("\n[INFO] Creating features ...\n")
        self.generate_features(load_path, save_path)
        print("\n[INFO] Loading features ...\n")
        features, labels, label_map = self.load_features(save_path)
        return features, labels, label_map
```

2 Execution Scripts

2.1 Function Approximation

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from models import Network
from layers import Dense, Input
from activations import Sigmoid, ReLU, Tanh, Softmax, Linear
from optimizers import SGD, AdaGrad, Adam
from plotting import SpatialPlot, ParityPlot, CallbackPlot, LayerOutputPlot


# Data preparation
train_reg = pd.read_csv('/home/nishant/Desktop/Semester
↪  6/CS6910/Assignments/Assignment 1/data/function_approx/train.csv')
val_reg = pd.read_csv('/home/nishant/Desktop/Semester
↪  6/CS6910/Assignments/Assignment 1/data/function_approx/val.csv')

X_train = train_reg[['x1', 'x2']].values.reshape((-1, 2))
y_train = train_reg['y'].values.reshape((-1, 1))
X_val = val_reg[['x1', 'x2']].values.reshape((-1, 2))
y_val = val_reg['y'].values.reshape((-1, 1))


# Define layers in correct sequence
layers = [
    Input(2, label='Input'),
    Dense(8, activation=Sigmoid(), label='Hidden_1'),
    Dense(6, activation=Sigmoid(), label='Hidden_2'),
    Dense(1, activation=Linear(), label='Output')
]


# Define model object and set layers
model = Network()
model.set_layers(layers)
model.compile(loss='RMSE', optimizer=SGD(lr=2e-06, momentum=0.9), metric='RMSE')

# Model training
model.train(X_train, y_train, epochs=30000, log_frequency=1000, val_split=0.0,
↪  val_sets=[X_val, y_val], track_epochs=[30000])
y_pred = model.predict(X_train)

# Plot regression results
```

```
SpatialPlot(X_train, y_train, y_pred)
ParityPlot(y_train, y_pred)

# Plot trend of loss function
CallbackPlot(model, 'Loss')
CallbackPlot(model, 'Metric')

# Output surface plot
save_path = '/home/nishant/Desktop/Semester 6/CS6910/Assignments/Assignment
 ↪  1/data/function_approx/plots/'
LayerOutputPlot(model, track_epochs=[30000], save_path=save_path)
```

2.2 2D Nonlinear Classifier

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from models import Network
from layers import Dense, Input
from losses import RMSE
from activations import Sigmoid, ReLU, Tanh, Softmax, Linear
from optimizers import SGD, AdaGrad, Adam
from utils import OneHotEncoder, MinMaxScaler
from plotting import SpatialPlot, ParityPlot, CallbackPlot, DecisionRegionPlot,
↪  ConfusionMatrix, LayerOutputPlot


# Data preparation
train_clf = pd.read_csv("/home/nishant/Desktop/Semester
↪  6/CS6910/Assignments/Assignment 1/data/2d_nonlinear/2d_nonlinear_data.csv")

X_train = train_clf[['x1', 'x2']].values.reshape((-1, 2))
y_true = train_clf['label'].values.reshape((-1, 1))
y_train = OneHotEncoder().transform(y_true)

# Scale data
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)


# Define layers in correct sequence
layers = [
    Input(2, label='Input'),
    Dense(5, activation=Sigmoid(), label='Hidden_1'),
    Dense(5, activation=Sigmoid(), label='Hidden_2'),
    Dense(3, activation=Softmax(), label='Output')
]


# Define model object and set layers
model = Network()
model.set_layers(layers)
model.compile(loss='CrossEntropy', optimizer=SGD(lr=0.01, momentum=0.9),
↪  metric='Accuracy')


# Model training
track_epochs = [1, 2, 10, 50, 200]
```

```
model.train(X_train, y_train, epochs=200, log_frequency=10, val_split=0.3,
 ↪  track_epochs=track_epochs)
y_probs = model.predict(X_train)
y_pred = np.array([np.argmax(i) for i in y_probs])

# Plot regression results
SpatialPlot(X_train, y_true, y_pred)

# Plot trend of loss function
CallbackPlot(model, 'Loss')
CallbackPlot(model, 'Metric')

# Plot decision regions
DecisionRegionPlot(model, X_train, y_true, y_pred)

# Confusion matrix
ConfusionMatrix(y_true, y_pred)

# Layer output plots
LayerOutputPlot(model, track_epochs=track_epochs,
 ↪  save_path='../../data/2d_nonlinear/plots/')
```

2.3 Image Classification

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
np.random.seed(1)

from models import Network
from layers import Dense, Input
from losses import RMSE
from activations import Sigmoid, ReLU, Tanh, Softmax, Linear
from optimizers import SGD, AdaGrad, Adam
from utils import OneHotEncoder, ImFeatureExtractor, MinMaxScaler
from plotting import SpatialPlot, ParityPlot, CallbackPlot, ConfusionMatrix


# Data preparation
load_path = '/home/nishant/Desktop/Semester 6/CS6910/Assignments/Assignment
↪  1/data/image_data/train/'
save_path = '/home/nishant/Desktop/Semester 6/CS6910/Assignments/Assignment
↪  1/data/image_data/FeatureExtraction_2D/'
X_train, y_true, label_map = ImFeatureExtractor().load_features(save_path)
y_true = np.array(list(y_true)).reshape((-1, 1))
y_train = OneHotEncoder().transform(y_true)

# Scale data
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)

# Define layers in correct sequence
input_dim = X_train.shape[1]
layers = [
    Input(input_dim, label='Input'),
    Dense(64, activation=Sigmoid(), label='Hidden_1'),
    Dense(64, activation=Sigmoid(), label='Hidden_2'),
    Dense(len(label_map), activation=Softmax(), label='Output')
]


# Define model object and set layers
model = Network()
model.set_layers(layers)
model.compile(loss='CrossEntropy', optimizer=SGD(lr=0.001, momentum=0.9),
↪  metric='Accuracy')


# Model training
```

```
model.train(X_train, y_train, epochs=500, log_frequency=10, val_split=0.3)
y_probs = model.predict(X_train)
y_pred = np.array([np.argmax(i) for i in y_probs])

# Plot regression results
SpatialPlot(X_train, y_true, y_pred)

# Plot trend of loss function
CallbackPlot(model, 'Loss')
CallbackPlot(model, 'Metric')

# Confusion Matrix
ConfusionMatrix(y_true, y_pred)
```