IST 597 Assignment – 2

Vaibhav Singh

PSU ID – 997500644

Access ID – vxs5308

Dataset - MNIST

Design MLP with 2 hidden layers (Input Layer - 2 hidden Layer - Output layer) to classify objects (fashion MNIST) and digits (MNIST).

Answer -

- I have designed a model which includes a default mode with no regularization, l2 mode and dropout.
- I have added knobs for each mode and hyperparameter setting in the class itself.
- For L2 regularization, I have updated loss to loss + (lambda/(2 * num_training_samples)) * sum of squares of each term in each weight matrix
- I have performed data preprocessing prior to using this model
    - Reshaped the data
    - Normalized the image data
    - One hot encoding

```
# Combined model with L2 and dropout enable knobs
import sys
from tensorflow.python.ops.gen_math_ops import square
import numpy as np
# Define class to build mlp model
class MLP_combined(object):
  def __init__(self, size_input, size_hidden_1, size_hidden_2,
size_output, reg_mode = "default", lambda_l2 = 0.8, dropout = 0.3, lr =
0.05, device=None):
    """
    size_input: int, size of input layer
    size_hidden: int, size of hidden layer
    size_output: int, size of output layer
    device: str or None, either 'cpu' or 'gpu' or None. If None, the
device to be used will be decided automatically during Eager Execution
    """
    import sys
    self.size_input, self.size_hidden_1, self.size_hidden_2,
self.size_output, self.reg_mode, self.lambda_l2, self.dropout, self.lr,
self.device =\
    size_input, size_hidden_1, size_hidden_2, size_output, reg_mode,
lambda_l2, dropout, lr, device
```

```
    # Initialize weights between input layer and hidden layer 1 - (128,
128)
    self.W1 = tf.Variable(tf.random.normal([self.size_input,
self.size_hidden_1]))
    # Initialize biases for hidden layer 1 - (1,128)
    self.b1 = tf.Variable(tf.random.normal([1, self.size_hidden_1]))
    # Initialize weights between hidden layer 1 and hidden layer 2 - (128,
128)
    self.W2 = tf.Variable(tf.random.normal([self.size_hidden_1,
self.size_hidden_2]))
    # Initialize biases for hidden layer 2 - (1, 128)
    self.b2 = tf.Variable(tf.random.normal([1, self.size_hidden_2]))
    # Initialize weights between hidden layer 2 and output layer - (128,
10)
    self.W3 = tf.Variable(tf.random.normal([self.size_hidden_2,
self.size_output]))
    # Initialize biases for output layer - (1,10)
    self.b3 = tf.Variable(tf.random.normal([1, self.size_output]))
    # Define variables to be updated during backpropagation
    self.variables = [self.W1, self.W2, self.W3, self.b1, self.b2,
self.b3]

  def forward(self, X):
    """
    forward pass
    X: Tensor, inputs
    """
    if self.device is not None:
      with tf.device('gpu:0' if self.device=='gpu' else 'cpu'):
        self.y = self.compute_output(X)
    else:
      self.y = self.compute_output(X)

    return self.y

  def loss(self, y_pred, y_true):
    '''
    y_pred - Tensor of shape (batch_size, size_output)
    y_true - Tensor of shape (batch_size, size_output)
    '''
    #Use categorical cross entropy in place of MSE
    cce = tf.keras.losses.CategoricalCrossentropy()
    #loss = tf.nn.softmax_cross_entropy_with_logits(y_true, y_pred)
    loss = cce(y_true, y_pred)
    if self.reg_mode == "L2":
      #Applying L2 regularization
      #print ("#Applying L2 regularization")
      para_list = [self.W1, self.W2, self.W3]
      regularizer = sum([tf.reduce_sum(tf.square(_var)) for _var in
para_list])
      #loss = self.lambda_l2 * loss +  ((1 - self.lambda_l2)/
number_of_train_examples) * regularizer
```

```
      loss = loss +  ((self.lambda_l2/(2 * number_of_train_examples)) *
regularizer)
    return loss

  def backward(self, X_train, y_train):
    """
    backward pass
    """
    optimizer = tf.keras.optimizers.SGD(learning_rate=self.lr)
    #optimizer = tf.keras.optimizers.Adagrad(learning_rate=0.05)
    with tf.GradientTape() as tape:
      predicted = self.forward(X_train)
      current_loss = self.loss(predicted, y_train)
      grads = tape.gradient(current_loss, self.variables)
      optimizer.apply_gradients(zip(grads, self.variables))


  def compute_output(self, X):
    """
    Custom method to obtain output tensor during forward pass
    """
    # Cast X to float32
    X_tf = tf.cast(X, dtype=tf.float32)

    #Remember to normalize your dataset before moving forward

    # Compute values in hidden layer 1
    w1hat = tf.matmul(X_tf, self.W1) + self.b1
    h1hat = tf.nn.relu(w1hat)
    # Compute values in hidden layer 2
    w2hat = tf.matmul(h1hat, self.W2) + self.b2
    h2hat = tf.nn.relu(w2hat)
    if self.reg_mode == "dropout":
      # Adding dropout on layer 2
      #print ("# Adding dropout on layer 2")
      h2hat = tf.nn.dropout(h2hat, self.dropout)
    # Compute output
    output = tf.matmul(h2hat, self.W3) + self.b3
    #Now consider two things , First look at inbuild loss functions if
they work with softmax or not and then change this
    #Second add tf.Softmax(output) and then return this variable
    output = tf.nn.softmax(output)
    #print (f"Output : {output}")
    return output
```

Design regularization approaches, and analyze drop/boost in performance of your model. Report results with atleast 2 regularization variants (droput, L1 penalty, L2, L1+L2, Normalization, Noise). Do you see any tradeoff with respect to bias-variance? Report your findings.

Answer -

- I have implemented L2 regularization and dropout with the main model (MLP_combined) itself as you can see above. Initially, I had 3 different models but this new approach made the code more modular and proved handy for inference.
- With different set of hyperparameters, we get different sets of results. As expected, we see substantial difference in execution times as well.
- For most of my optimizations, I am getting good performance with dropout and very minimal improvement with L2 regularization.
- When the number of neurons in hidden layer is less(say 100) then L2 tends to outperform dropout and I will discuss that in the next question. Dropout however improves performs with increase in epochs.
- The performance across test and training set tends to stay almost the same hence I would say that variance is low in almost all cases and regularization doesn't affect that much.
- Bias within the data tends to improve with dropout but with optimal settings I am getting an average bias for all models

How did you perform hyper-parameter optimization? Report your approach, also show settings for best model.

Answer -

- I have tried various optimizations but I couldn't document all sorts of results.
- Cases with low batch size tends to take a lot of execution time and hence I couldn't update all those results.
- With high batch size, on increasing epochs the performance also improves a lot.
- To ease the effort in execution, I have made the code modular.
- Ideally, higher batch sizes work well higher learning rates.
- Dropout of 0.2 to 0.3 tends to work fine.
- I also tried various optimizers such as Adam, Adagrad, RMSProp but SGD itself performs the best in my findings
- With higher number of hidden layers(such as 512,128), accuracy improved much faster but reached minima very soon after which improvement is very gradual.
- In some set of hyperparameters, the default mode with no regularization tends to give low accuracies and the result is substantially improved with dropout.
- Higher lambda for l2 reg values yielded bad performance in general.

Model	Initial hyperparams	Hyperparams	Train Accuracy	Train Cross entropy Loss	Comments
default	epochs = 20 side_hidden_1 = size_hidden_2 = 100	batch size - 512 shuffle - 1000 learning rate - 0.05	72.10%	0.063	Getting better Test Cross entropy loss but similar Accuracy in test
L2	epochs = 20 side_hidden_1 = size_hidden_2 = 100	batch size - 512 shuffle - 1000 learning rate - 0.05 lambda - 0.05	74.416%	0.0081	Getting better Test Cross entropy loss but similar Accuracy in test
dropout	epochs = 20 side_hidden_1 = size_hidden_2 = 100	batch size - 512 shuffle - 1000 learning rate - 0.05 dropout - 0.3	68.72%	0.0098	Convergence is initially slower with this set of hidden layer combination.
dropout	epochs = 20 side_hidden_1 = size_hidden_2 = 100	batch size - 512 shuffle - 1000 learning rate - 0.05 dropout - 0.4	66.92%	0.0098	Convergence is initially slower with this set of hidden layer combination. However overall performance can improve a lot with more epochs since our batch size is high.
default	epochs = 20 side_hidden_1 = size_hidden_2 = 100	batch size - 64 shuffle - 100 learning rate - 0.05	74.91%	0.063	
dropout	epochs = 20 side_hidden_1 = size_hidden_2 = 100	batch size - 64 shuffle - 100 learning rate - dropout - 0.3	81.12%	0.047	**Accuracy with dropout is the best in this setting**
l2	epochs = 20 side_hidden_1 = size_hidden_2 = 100	batch size - 64 shuffle - 100 learning rate - 0.05	72.2%	0.07	
dropout	epochs = 250	batch size - 512 shuffle - 1000	80.88%	0.0059	**Accuracy and loss with dropout are the best in**

	side_hidden_1 = size_hidden_2 = 100	learning rate - 0.05 dropout - 0.3			**this setting. The convergence of loss is better here**

After running inference, I noticed that seeds tend to have a substantial impact on the results.

Report your results across multiple trials (minimum 10). Beside accuracy you will also report standard error and plot graph showing variance.

Answer –

```
############## Setting values as per dataset ##############
size_input = 784
size_hidden_1 = 100
size_hidden_2 = 100
size_output = 10
number_of_train_examples = X_train.shape[0]
number_of_test_examples = X_test.shape[0]
print (f"{number_of_test_examples},{number_of_train_examples}")
#########################################################
'''
Additional steps done above
1. Reshape to flatten
2. Normalization
3. One hot encoding
'''
#########################################################
reg_type_list = ["default", "L2", "dropout"]
inference_stats_dict = {"default":{}, "L2": {}, "dropout": {}}
lambda_val = 0.05
dropout_val = 0.3
NUM_EPOCHS = 20
random_seeds = list(np.random.randint(low=2000, high=4000, size=10))
inf_iters = 10
device_type = 'gpu'
shuffle_size = 1000
batch_size = 512
learning_rate = 0.05
test_ds = tf.data.Dataset.from_tensor_slices((X_test, y_test)).batch(100)
```

```
[67]  import pandas as pd
      df = pd.DataFrame()
      df["seed"] = random_seeds
      df["default_cce"] = inference_stats_dict["default"]["cce_list"]
      df["L2_cce"] = inference_stats_dict["L2"]["cce_list"]
      df["dropout_cce"] = inference_stats_dict["dropout"]["cce_list"]

      df["default_acc"] = inference_stats_dict["default"]["acc_list"]
      df["L2_acc"] = inference_stats_dict["L2"]["acc_list"]
      df["dropout_acc"] = inference_stats_dict["dropout"]["acc_list"]
```

```
df
```

	seed	default_cce	L2_cce	dropout_cce	default_acc	L2_acc	dropout_acc
0	3058	0.074838	0.050778	0.026508	0.5342	0.6847	0.8314
1	3717	0.058118	0.056247	0.057397	0.6372	0.6514	0.6419
2	2781	0.088476	0.098492	0.058324	0.4499	0.3908	0.6363
3	2563	0.029573	0.057030	0.039482	0.8153	0.6470	0.7524
4	2120	0.061539	0.062051	0.052544	0.6160	0.6157	0.6714
5	2717	0.085984	0.090801	0.067630	0.4650	0.4369	0.5779
6	3823	0.072293	0.072833	0.039442	0.5498	0.5493	0.7522
7	2690	0.073540	0.072031	0.039175	0.5418	0.5535	0.7548
8	3371	0.084251	0.071207	0.063647	0.4765	0.5597	0.6031
9	3155	0.073906	0.074139	0.036693	0.5408	0.5414	0.7694

Capturing a screenshot from the results obtained after running trials for 10 different seeds.
In this process, I ran overall 30 iterations of training and testing and documented every result above.

Updating mean and variance for losses and accuracies as was suggested in the announcement –

```
for ele in inference_stats_dict:
  print (f"## Model: {ele} ###")
  print (f"Mean of cross entropy : {np.mean(inference_stats_dict[ele]['cce_list'])}")
  print (f"Variance of cross entropy : {np.var(inference_stats_dict[ele]['cce_list'])}")
  print (f"Mean of accuracies : {np.mean(inference_stats_dict[ele]['acc_list'])}")
  print (f"Variance of accuracies : {np.var(inference_stats_dict[ele]['acc_list'])}")
  print ()
```

```
## Model: default ###
Mean of cross entropy : 0.07025182403564453
Variance of cross entropy : 0.0002702885774382875 5
Mean of accuracies : 0.56265
Variance of accuracies : 0.0103792525

## Model: L2 ###
Mean of cross entropy : 0.07056093688964844
Variance of cross entropy : 0.0002062116748892102 8
Mean of accuracies : 0.5630400000000001
Variance of accuracies : 0.0078768164

## Model: dropout ###
Mean of cross entropy : 0.04808411865234376
Variance of cross entropy : 0.0001659007697437442 4
Mean of accuracies : 0.6990799999999999
Variance of accuracies : 0.006309677600000001
```

From the results, we can see that we get the most consistent best results for dropout for the following 10 seeds

Dataset - Fashion MNIST

The model stays the same for Fashion MNIST but the hyperparameters change. Unfortunately, I couldn't document all possible values and hence updating the final 10 trials I performed with my set of settings

```
df
```

	seed	default_cce	L2_cce	dropout_cce	default_acc	L2_acc	dropout_acc
0	4225	0.115223	0.115758	0.090128	0.2849	0.2839	0.4389
1	4555	0.084675	0.080690	0.052094	0.4742	0.5004	0.6743
2	4193	0.085767	0.091453	0.075550	0.4646	0.4335	0.5290
3	4027	0.114549	0.115045	0.059787	0.2888	0.2883	0.6259
4	4747	0.086422	0.072184	0.060580	0.4625	0.5532	0.6230
5	4106	0.099603	0.100119	0.073309	0.3818	0.3809	0.5426
6	4079	0.075448	0.085236	0.074884	0.5296	0.4725	0.5337
7	4285	0.104943	0.115676	0.067916	0.3482	0.2836	0.5756
8	4648	0.107314	0.102253	0.069067	0.3330	0.3672	0.5689
9	4912	0.118343	0.116457	0.064417	0.2655	0.2795	0.5966

```
## Model: default ###
Cross entropy loss across trials : [0.11522347412109375, 0.08467548828125,
0.08576741333007812, 0.1145487060546875, 0.08642153930664062,
0.09960307006835938, 0.07544818725585938, 0.104943408203125,
0.1073142578125, 0.11834266357421876]
Accuracy across trials : [0.2849, 0.4742, 0.4646, 0.2888, 0.4625, 0.3818,
0.5296, 0.3482, 0.333, 0.2655]
```

```
## Model: L2 ###
Cross entropy loss across trials : [0.11575765380859375,
0.08069005737304688, 0.09145341186523437, 0.11504547119140625,
0.07218370971679687, 0.10011902465820313, 0.08523603515625,
0.11567591552734376, 0.10225344848632813, 0.1164573974609375]
Accuracy across trials : [0.2839, 0.5004, 0.4335, 0.2883, 0.5532, 0.3809,
0.4725, 0.2836, 0.3672, 0.2795]
```

```
## Model: dropout ###
Cross entropy loss across trials : [0.09012791137695313,
0.05209363403320313, 0.07554984130859375, 0.059787115478515625,
0.06057958984375, 0.07330875244140625, 0.07488440551757812,
0.06791614379882813, 0.06906673583984375, 0.0644167724609375]
Accuracy across trials : [0.4389, 0.6743, 0.529, 0.6259, 0.623, 0.5426,
0.5337, 0.5756, 0.5689, 0.5966]
```

```python
for ele in inference_stats_dict:
  print (f"## Model: {ele} ###")
  print (f"Mean of cross entropy : {np.mean(inference_stats_dict[ele]['cce_list'])}")
  print (f"Variance of cross entropy : {np.var(inference_stats_dict[ele]['cce_list'])}")
  print (f"Mean of accuracies : {np.mean(inference_stats_dict[ele]['acc_list'])}")
  print (f"Variance of accuracies : {np.var(inference_stats_dict[ele]['acc_list'])}")
  print ()
```

```
## Model: default ###
Mean of cross entropy : 0.09922882080078126
Variance of cross entropy : 0.00020765916375456756
Mean of accuracies : 0.38331000000000004
Variance of accuracies : 0.007880282899999999

## Model: L2 ###
Mean of cross entropy : 0.09948721252441406
Variance of cross entropy : 0.00024313515187576748
Mean of accuracies : 0.3843
Variance of accuracies : 0.009292995999999998

## Model: dropout ###
Mean of cross entropy : 0.06877309020996095
Variance of cross entropy : 0.00010057566869095305
Mean of accuracies : 0.5708499999999999
Variance of accuracies : 0.0038481864999999997
```