IST 597 Assignment – 3

Vaibhav Singh

PSU ID – 997500644

Access ID – vxs5308

FASHION MNIST

I have created a generic class which covers all model combinations and have added a function to update parameters using the given algorithm.

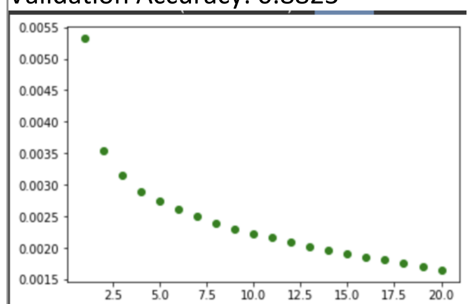
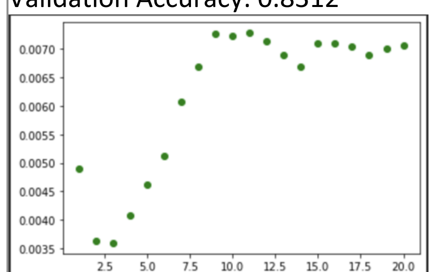
Updating the backward() and stochastic update function here.
Please refer the model from the notebook. I ended up spending a lot of time in trying the open source version sugesstions and couldn't eventually make that work. On the other hand, the design mentioned below was easy to code but was tough to debug. I went ahead with epsilon in the denominator. Initially, I didn't know that tensorflow pow function won't handle negative numbers for cube roots. Np.cbrt tends to handle signs as well. To retain the sign and the value, I have used tf.sign and tf.abs. I hope that approach is correct.

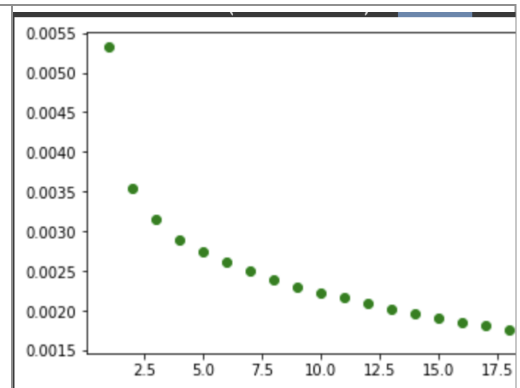
```
def backward(self, X_train, y_train):
    """
    backward pass
    """
    with tf.GradientTape() as tape:

      predicted = self.forward(X_train)
      current_loss = self.loss(predicted, y_train)

    grads = tape.gradient(current_loss, self.variables)

    if self.optimizer == "custom":
      self.Stochastic_optimization_update_params(grads, self.lr)

    else:
      if self.optimizer == "SGD":
        optimizer = tf.keras.optimizers.SGD(learning_rate = self.lr,
momentum=0.9)

      elif self.optimizer == "Adam":
```

```
        optimizer = tf.keras.optimizers.Adam(learning_rate= self.lr,
beta_1=0.9, beta_2=0.999, epsilon=1e-08)

      elif self.optimizer == "RMSprop":
        optimizer = tf.keras.optimizers.RMSprop(learning_rate= self.lr)
      else:
        print ("Invalid optimizer")
        sys.exit()

      optimizer.apply_gradients(zip(grads, self.variables))
def Stochastic_optimization_update_params(self, grads, lr=2e-5, beta1=0.9,
beta2=0.999, beta3=0.999987, epsilon=10**-8):
    #sys.exit()
    #print (f"\n\n\n First m : {self.m}")
    self.iterations += 1
    beta1_t = beta1 ** self.iterations
    beta2_t = beta2 ** self.iterations
    beta3_t = beta3 ** self.iterations

    self.m = [beta1 * m_ele + (1 - beta1) * grad_ele for m_ele, grad_ele
in zip(self.m, grads)]
    self.v = [beta2 * v_ele + (1 - beta2) * (grad_ele ** 2) for v_ele,
grad_ele in zip(self.v, grads)]
    self.u = [beta3 * u_ele + (1 - beta3) * (grad_ele ** 3) for u_ele,
grad_ele in zip(self.u, grads)]

    m_hat =[m_t / (1 - beta1_t) for m_t in self.m]
    v_hat =[v_t / (1 - beta2_t) for v_t in self.v]
    u_hat =[u_t / (1 - beta3_t) for u_t in self.u]

    v_hat_root2 = [tf.sign(v_hat_ele) * tf.sqrt(tf.abs(v_hat_ele)) for
v_hat_ele in v_hat]
    u_hat_root3 = [tf.sign(u_hat_ele) * tf.pow(tf.abs(u_hat_ele), (1/3))
for u_hat_ele in u_hat]

    sum_denom = [(v_hat_root2_ele + u_hat_root3_ele * epsilon) + epsilon
for v_hat_root2_ele, u_hat_root3_ele in zip(v_hat_root2, u_hat_root3)]
    new_vars_t = [vars_ele - lr * (m_hat_ele / sum_denom_ele) for
vars_ele, m_hat_ele, sum_denom_ele  in zip(self.variables, m_hat,
sum_denom)]

    #print (f"Variable: {self.variables}")
    for i in range(len(self.variables)):
      self.variables[i].assign(new_vars_t[i])
```

Apart from this, I have written a generic train and test model function as well which gets utilized later.

Following are the results that I had obtained in the initial testing and I further used Colab pro for execution(Hence the improve in speed).

Optimizer	Regularization mode	Optimizer Hyperparameters	CPU time	Final Loss and Accuracy and plot
SGD	Default – no regularization	lr=0.1, momentum=0.9	219.55 s	Train Accuracy: 0.9203 Average Cross Entropy:= 0.0016488372802734375 Validation Accuracy: 0.8823 
ADAM	Default – no regularization	lr=0.001 beta_1=0.9, beta_2=0.999, epsilon=1e-08	275.42	Train Accuracy: 0.8481 Average Cross Entropy:= 0.0070640948486328l2 Validation Accuracy: 0.8312 
RMS Prop	Default – no regularization	lr=0.001 momentum=0.9	302.26	Train Accuracy: 0.8232 Average Cross Entropy:= 0.0484434521484375 Validation Accuracy: 0.8156
Custom	Default – no regularization	lr=0.001 beta1=0.9, beta2=0.999,	280.28	Train Accuracy: 0.9321 Average Cross Entropy:= 0.001183359375 Validation Accuracy: 0.8824

		beta3=0.999987, epsilon=10**-8		

SGD and Custom optimizer always tend to perform well while RMSprop loss doesn't tend to monotonically decrease usually. I will provide more statistical data in the inference.

Inference across 10 trials

The assignment demanded optimization benchmarking in terms of Speed, Stability and Robustness. I referred to the following paper to learn more about it and hopefully I can utilize that knowledge beyond the scope of this assignment.
https://arxiv.org/pdf/1709.08242.pdf

In the interest of time, I have just targeted the following as suggested though.

- Speed
  - Average execution times for training across trials
- Robustness
  - Average accuracy of prediction of test data across trials
- Stability
  - Average variance in loss(cce) across epochs across seeds for every model

Pain points –

- Colab crashes! Need to find faster ways of execution as suggested.
- 120 iterations still seem far too many.

Now, given the amount of data that gets generated, I wanted to have a clean JSON which accumulates all possible data for further data analysis.

For this purpose, I have designed the code such it generates a JSON in three layer hierarchical format as shown below. This eased my data analysis by a huge margin.

```
optimizers = learning_rates.keys()

test_ds = tf.data.Dataset.from_tensor_slices((X_test, y_test)).batch(256)
####################################################

for optimizer in optimizers:
  inference_stats_dict = {
    "default":{},
    "L2": {},
    "dropout": {}
  }
  print (f"#####Running trials for optimizer: {optimizer.upper()}#####")
  for reg_type in reg_type_list:
    print (f"## START: Current Model: {reg_type} ##")
    test_cce_list = []
    test_acc_list = []
    cpu_time_list = []
    train_acc_list_dict = {}
    val_acc_list_dict = {}
    train_cce_list_dict = {}

    for iter in tqdm(range(inf_iters)):
      print(f"\n** Inference Iteration: {iter} **")
      cur_seed = random_seeds[iter]
      print (f"\n#Training {reg_type} model, optimizer {optimizer} with
seed {cur_seed}#")
      np.random.seed(cur_seed)
      tf.random.set_seed(cur_seed)


      # Model initialization as per regularization type - regularization
code triggers only if knob is on
      mlp = MLP(size_input, size_hidden_1, size_hidden_2, size_output,
reg_mode = reg_type, lambda_l2 = lambda_val, dropout = dropout_val, lr =
learning_rates[optimizer], optimizer=optimizer, device=device_type)
      cputime, train_acc_list, val_acc_list, train_cce_list = \
                train_model(mlp, NUM_EPOCHS, seed=cur_seed,
shuffle_size=shuffle_size, batch_size=batch_size)

      train_acc_list_dict[cur_seed] = train_acc_list
      val_acc_list_dict[cur_seed] = val_acc_list
      train_cce_list_dict[cur_seed] = train_cce_list
      cpu_time_list.append(cputime)

      # Testing the data
```

```
      (cce_test, acc_test) = test_model(mlp)
      print(f"seed: {cur_seed}, Test Cross Entropy loss: {cce_test},
Accuracy: {acc_test}")

      test_cce_list.append(cce_test)
      test_acc_list.append(acc_test)

    inference_stats_dict[reg_type]["test_cce_list"] = test_cce_list
    inference_stats_dict[reg_type]["test_acc_list"] = test_acc_list
    inference_stats_dict[reg_type]["cputime_list"] = cpu_time_list
    inference_stats_dict[reg_type]["train_acc_list_dict"] =
train_acc_list_dict
    inference_stats_dict[reg_type]["val_acc_list_dict"] =
val_acc_list_dict
    inference_stats_dict[reg_type]["train_cce_list_dict"] =
train_cce_list_dict

    print (f"## END: Current Model: {reg_type}##")
    print (f"Current inference results: \n{inference_stats_dict}")
  print (f"Inference stat dict for optimizer : {optimizer}:
\n{inference_stats_dict}")
  inference_stats_total_dict[optimizer] = inference_stats_dict
  print (f"Current status of dictionary: {inference_stats_total_dict}")
  with open('fmnist_stats.json', 'w') as convert_file:
     convert_file.write(json.dumps(str(inference_stats_total_dict)))
  ################################
```

Following is an abstract design from JSON visualizer


```
{
  "custom" : {
    "default" : {
      "test_cce_list" : [ ],
      "test_acc_list" : [ ],
      "cputime_list" : [ ],
      "train_acc_list_dict" : {
        "36549" : [
          0.908079981803894,
          0.940020247764587,
          0.9561399817466736,
          0.9651399850845337,
          0.9708399772644043,
          0.9759200215339661,
          0.979640007019043,
          0.982200026512146,
          0.98444002866745,
          0.98602002859115б
        ],
        "13790" : [ ],
        "41463" : [ ],
        "52126" : [ ],
        "1272" : [ ],
        "60548" : [ ],
        "14578" : [ ],
        "86558" : [ ],
        "96271" : [ ],
        "57229" : [ ]
      },
      "val_acc_list_dict" : { },
      "train_cce_list_dict" : { }
    },
    "L2" : { },
    "dropout" : { }
  },
  "Adam" : {
    "default" : { },
    "L2" : { },
    "dropout" : { }
  },
  "RMSprop" : {
    "default" : { },
    "L2" : { },
    "dropout" : { }
  },
  "SGD" : { }
}
```

Following are the hyperparameters that I have used –
size_input = 784
size_hidden_1 = 128
size_hidden_2 = 128
size_output = 10

lambda_val = 0.7
dropout_val = 0.3
NUM_EPOCHS = 10 # reduced to 10 to save overall time
shuffle_size = 25
batch_size = 128
#batch_size = 512
learning_rates = {
    "SGD": 0.1,
    "Adam": 0.001,
    "RMSprop": 0.001,
    "custom": 0.001
}

I have written code snippets to further analyze the generated JSON and obtain tables and plots from that data. Please refer the colab for that.

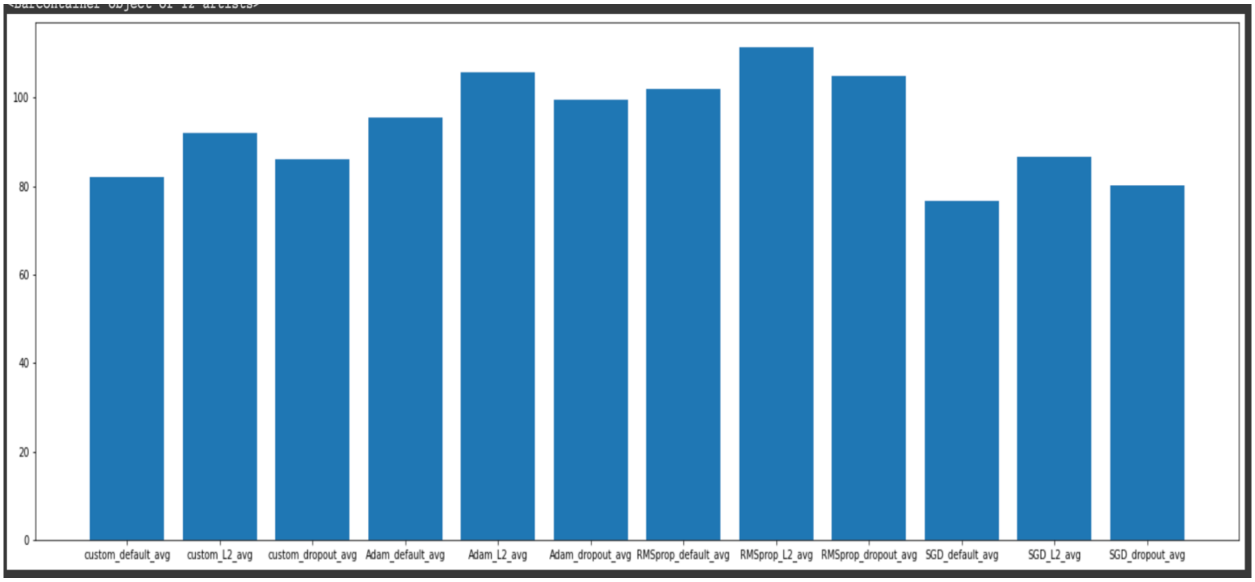
Following are some of the generated results: -

Speed can be gauged by assessing the approximate CPU times across trials. Here are some captured results.

##CPU time stats across trials##

	custom_default	custom_L2	custom_dropout	Adam_default	Adam_L2	Adam_dropout	RMSprop_default	RMSprop_L2	RMSprop_dropout	SGD_default	SGD_L2	SGD_dropout
0	83.710175	92.067148	85.839592	95.606847	105.518696	99.556717	102.029196	110.794125	104.194310	77.655035	87.164152	79.824654
1	82.010244	91.867406	85.949927	95.605773	105.165371	99.269905	102.152241	111.117599	104.563838	76.703920	87.133283	80.350165
2	81.757674	91.760205	86.745251	95.954747	105.581319	99.270465	101.639173	111.206578	104.713480	76.450304	86.756881	80.107274
3	81.803616	91.670838	86.833428	95.967432	105.584304	99.610166	102.468657	112.332230	104.696564	76.529816	86.446709	80.221324
4	82.099605	91.244393	86.140921	95.381497	105.826796	99.712230	102.209926	111.733275	104.582003	76.279214	86.661093	80.284512
5	81.570583	91.909757	86.026944	95.321669	107.002815	99.403781	102.011270	111.489617	104.976014	76.605995	86.455837	79.962278
6	81.503945	92.344973	85.876193	95.598744	106.332536	99.624222	101.916068	111.293188	104.896109	76.650741	86.240966	80.087236
7	82.228379	92.330935	86.084583	95.530583	105.906677	99.678480	101.989442	110.940096	105.041513	76.732656	86.454679	80.407833
8	83.025987	92.550790	85.870790	95.365998	105.597836	99.939083	102.132959	112.948312	106.781350	76.933826	86.124023	80.038608
9	81.346270	92.187693	86.072386	95.231651	105.354260	99.600650	100.941441	110.735229	105.057623	76.570743	86.622722	80.360065

```
#Average CPU time per model#
custom_default_avg      82.105648
custom_L2_avg           91.993414
custom_dropout_avg      86.144002
Adam_default_avg        95.556494
Adam_L2_avg            105.787061
Adam_dropout_avg        99.566570
RMSprop_default_avg    101.949037
RMSprop_L2_avg         111.459025
RMSprop_dropout_avg    104.950280
SGD_default_avg         76.711225
SGD_L2_avg              86.606034
SGD_dropout_avg         80.164395
Name: 0, dtype: float64
```



Our custom optimizer always performs well in terms of overall execution time. But SGD performs the best. A possible reason might be that SGD has higher learning rate of 0.1 in our case. RMSprop tends to take the most time across all runs.

To check Robustness, we can have a look at the test accuracies across trials. Here are some captured results,

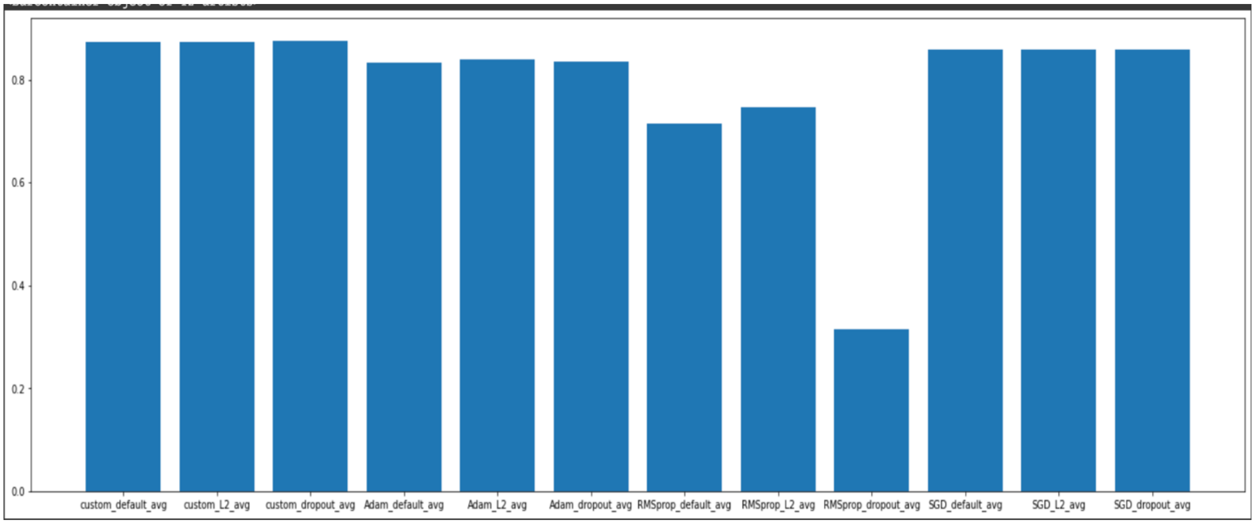
##Accuracy stats across trials##

	custom_default	custom_L2	custom_dropout	Adam_default	Adam_L2	Adam_dropout	RMSprop_default	RMSprop_L2	RMSprop_dropout	SGD_default	SGD_L2	SGD_dropout
0	0.8717	0.8737	0.8752	0.8211	0.8216	0.8421	0.7271	0.7719	0.2492	0.8680	0.8683	0.8647
1	0.8754	0.8721	0.8761	0.8383	0.8535	0.8317	0.7491	0.7435	0.2413	0.8557	0.8572	0.8501
2	0.8745	0.8749	0.8694	0.8357	0.8464	0.8333	0.7260	0.6278	0.3499	0.8513	0.8519	0.8588
3	0.8714	0.8753	0.8738	0.8234	0.8446	0.8395	0.7478	0.7475	0.3198	0.8481	0.8478	0.8500
4	0.8728	0.8742	0.8777	0.8303	0.8344	0.8201	0.7440	0.7876	0.2812	0.8519	0.8509	0.8542
5	0.8751	0.8773	0.8776	0.8500	0.8437	0.8409	0.7276	0.7015	0.3258	0.8633	0.8642	0.8648
6	0.8747	0.8769	0.8800	0.8255	0.8439	0.8182	0.6842	0.7738	0.3549	0.8616	0.8633	0.8607
7	0.8702	0.8713	0.8765	0.8549	0.8520	0.8584	0.5322	0.8046	0.4594	0.8608	0.8621	0.8609
8	0.8711	0.8719	0.8771	0.8192	0.8341	0.8504	0.7530	0.7168	0.2870	0.8577	0.8595	0.8618
9	0.8694	0.8724	0.8781	0.8272	0.8267	0.8211	0.7503	0.7941	0.2709	0.8625	0.8636	0.8563

#Average Accuracy per model#

custom_default_avg	0.87263
custom_L2_avg	0.87400
custom_dropout_avg	0.87615
Adam_default_avg	0.83256
Adam_L2_avg	0.84009
Adam_dropout_avg	0.83557
RMSprop_default_avg	0.71413
RMSprop_L2_avg	0.74691
RMSprop_dropout_avg	0.31394
SGD_default_avg	0.85809
SGD_L2_avg	0.85888
SGD_dropout_avg	0.85823

Name: 0, dtype: float64

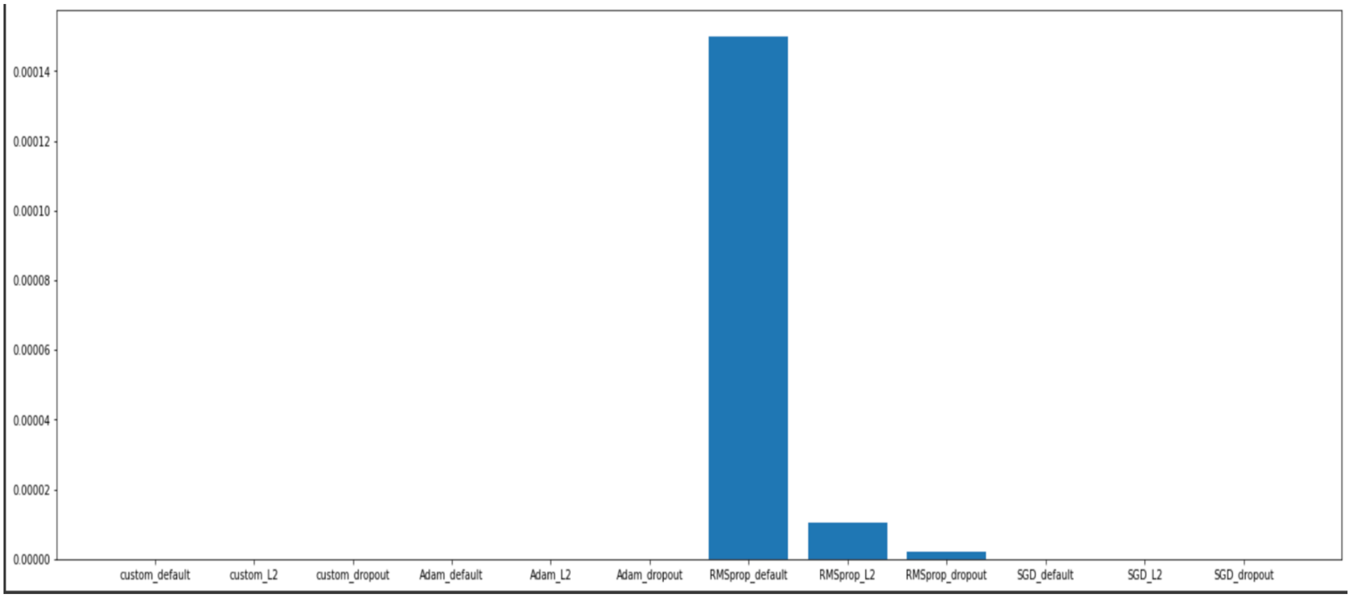


RMS Prop with dropout shows the worst performance in terms of robustness. It performs bad consistently. On the other hand, our custom optimizer tends to provide the most robust results along with SGD.

Lastly, to check stability we can check how the models perform in terms of variance in average loss over epochs across all seeds

```
#Average Variance in loss across seeds per model#
custom_default     2.124437e-09
custom_L2          2.271414e-09
custom_dropout     2.859384e-10
Adam_default       1.471043e-08
Adam_L2            8.894657e-09
Adam_dropout       9.679884e-09
RMSprop_default    1.500571e-04
RMSprop_L2         1.044649e-05
RMSprop_dropout    2.034020e-06
SGD_default        1.498629e-09
SGD_L2             1.550205e-09
SGD_dropout        1.342304e-09
```

RMS Prop tends to show the least stability and worst with dropout. I haven't tried to optimize this hyperparameter much since it wasn't needed for this assignment as much the discussion.



The same can be seen in the above plot. RMSProp tends to have the most variance in training loss across seeds per epoch.

MNIST

I have used the same hyperparameters as above for this dataset as well except batch size which I had set to 256 for faster execution. Batch size of 128 was providing better accuracies though

Capturing the data similar to what's done above –

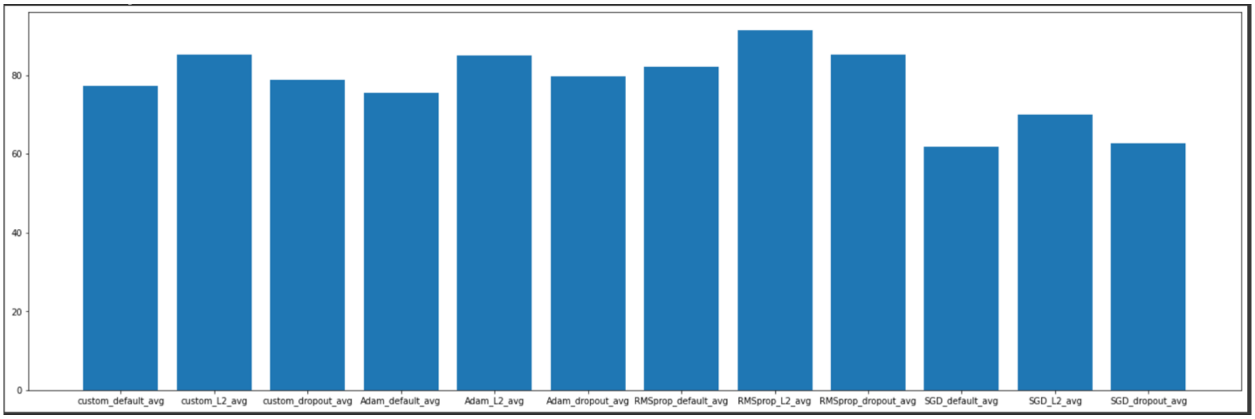
Speed

##CPU time stats across trials##

	custom_default	custom_L2	custom_dropout	Adam_default	Adam_L2	Adam_dropout	RMSprop_default	RMSprop_L2	RMSprop_dropout	SGD_default	SGD_L2	SGD_dropout
0	81.056575	85.393682	78.904905	74.610472	84.867258	79.378205	81.281124	91.615340	85.436424	65.754269	70.943370	62.247257
1	78.000971	84.796916	78.222516	74.751670	85.366175	79.195696	83.830038	92.434929	84.904968	60.496041	73.915814	62.512293
2	76.557593	86.687062	79.428418	74.857095	85.007599	79.868427	81.850318	90.261486	85.436703	60.345958	68.304167	62.168037
3	76.730594	88.913436	78.288000	74.753294	85.110517	79.491824	81.415945	92.283919	85.376212	61.187734	71.875464	66.342970
4	76.021193	86.038294	80.275937	76.372594	83.722287	79.478431	81.768977	91.522833	85.289327	61.342974	68.191761	62.414368
5	76.339379	83.172057	79.989098	75.923751	85.736207	79.670385	84.292369	92.861003	84.922684	61.285855	68.192899	62.846280
6	77.513205	83.218222	77.326227	75.589902	85.417860	79.408391	81.187147	90.990103	85.181624	65.267457	67.805723	62.184449
7	77.102227	84.729416	77.628460	75.211616	83.761706	79.770710	81.273321	90.709859	86.006834	61.836177	68.000276	62.113513
8	76.863275	85.259063	80.735075	75.780097	86.187139	80.111211	83.334080	90.604212	84.815243	60.350822	75.481711	61.322238
9	76.425348	84.667145	77.836261	76.314282	85.315907	80.454849	81.395342	91.234927	83.671408	60.241013	67.812474	63.013760

#Average CPU time per model#

custom_default_avg	77.261036
custom_L2_avg	85.287529
custom_dropout_avg	78.863490
Adam_default_avg	75.416477
Adam_L2_avg	85.049266
Adam_dropout_avg	79.682813
RMSprop_default_avg	82.162866
RMSprop_L2_avg	91.451861
RMSprop_dropout_avg	85.104143
SGD_default_avg	61.810830
SGD_L2_avg	70.052366
SGD_dropout_avg	62.716517



SGD tends to work the fastest owing mostly to its learning_rate.
Our custom algorithm also tends to have comparable performance. RMSProp is consistently the slowest on average

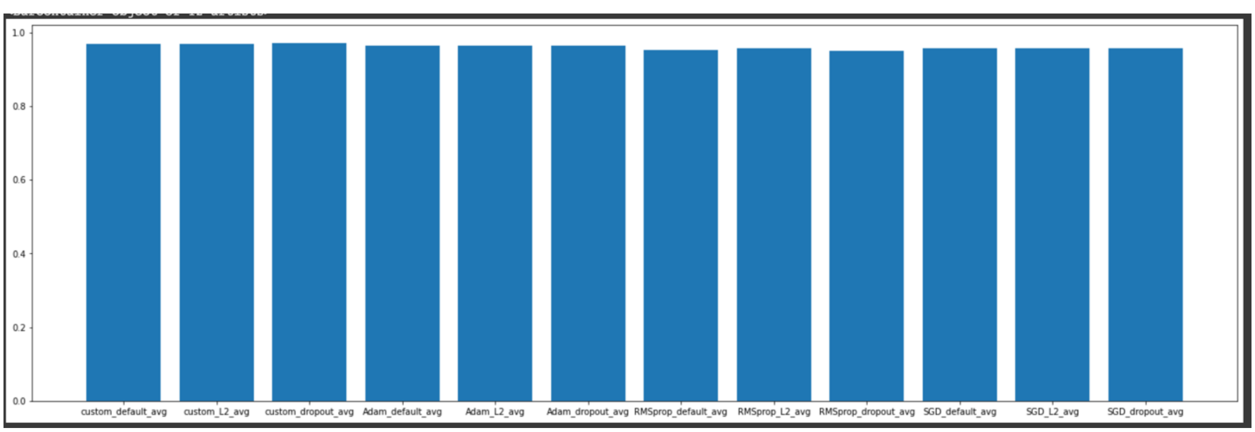
Robustness

##Accuracy stats across trials##

	custom_default	custom_L2	custom_dropout	Adam_default	Adam_L2	Adam_dropout	RMSprop_default	RMSprop_L2	RMSprop_dropout	SGD_default	SGD_L2	SGD_dropout
0	0.9716	0.9706	0.9722	0.9653	0.9651	0.9641	0.9600	0.9542	0.9555	0.9488	0.9474	0.9558
1	0.9693	0.9687	0.9709	0.9658	0.9645	0.9644	0.9526	0.9546	0.9431	0.9514	0.9516	0.9563
2	0.9660	0.9688	0.9715	0.9558	0.9616	0.9623	0.9554	0.9438	0.9412	0.9499	0.9498	0.9570
3	0.9651	0.9670	0.9723	0.9605	0.9608	0.9658	0.9575	0.9593	0.9570	0.9602	0.9603	0.9592
4	0.9688	0.9701	0.9709	0.9637	0.9633	0.9638	0.9566	0.9656	0.9442	0.9604	0.9605	0.9564
5	0.9710	0.9707	0.9721	0.9661	0.9649	0.9632	0.9585	0.9580	0.9497	0.9567	0.9569	0.9572
6	0.9678	0.9678	0.9716	0.9640	0.9635	0.9665	0.9555	0.9666	0.9453	0.9570	0.9590	0.9571
7	0.9687	0.9681	0.9731	0.9625	0.9591	0.9656	0.9299	0.9374	0.9557	0.9545	0.9541	0.9567
8	0.9711	0.9700	0.9702	0.9654	0.9660	0.9649	0.9470	0.9631	0.9525	0.9609	0.9608	0.9575
9	0.9667	0.9670	0.9688	0.9638	0.9656	0.9632	0.9355	0.9559	0.9528	0.9618	0.9623	0.9594

#Average Accuracy per model#

custom_default_avg	0.96861
custom_L2_avg	0.96888
custom_dropout_avg	0.97136
Adam_default_avg	0.96329
Adam_L2_avg	0.96344
Adam_dropout_avg	0.96438
RMSprop_default_avg	0.95085
RMSprop_L2_avg	0.95585
RMSprop_dropout_avg	0.94970
SGD_default_avg	0.95616
SGD_L2_avg	0.95627
SGD_dropout_avg	0.95726



On an average, all models are approximately robust to change in seeds for MNIST data.

Stability

```
#Average Variance in loss across seeds per model#
custom_default     1.056462e-09
custom_L2          7.386467e-10
custom_dropout     2.057086e-10
Adam_default       1.417280e-09
Adam_L2            1.389877e-09
Adam_dropout       4.274127e-10
RMSprop_default    5.485912e-09
RMSprop_L2         1.038087e-09
RMSprop_dropout    1.731371e-09
SGD_default        8.375600e-10
SGD_L2             8.415907e-10
SGD_dropout        8.812967e-10
```

Average variance across loss is relatively stable compared to FMNIST.

Overall, for better performance, I tried HE initialization since it performs well with ReLU but skipped it later. Apart from that, I tried log scaling for learning rate hyperparameter tuning. Based on the suggestions later, I dropped the extended tuning process based on inference.