IST 597 Assignment – 4

Vaibhav Singh

PSU ID – 997500644

Access ID – vxs5308

FASHION MNIST - MLP

I have created a generic class which covers both pre-activation and post-activation models and have accordingly added a function for batch normalization handling train and test phases.

We know that batch normalization expects some additional learnable parameters for each layer

- beta_[layer]
- gamma_[layer]

I have updated the newly added variables to the trainable variable list to ensure that backpropagation updates them via apply_gradients

For every activation (pre or post), we calculate the mean and variance and then use them to calculate the normalized value for the batch using the following formula –

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}.$$

Gamma and Beta for every layer is assigned to ones and zeros respectively to ensure that the formula works as expected. An epsilon term is added in the denominator to avoid possible errors. Effectively, in my implementation I have used the following formula since variance is actually σ^2

$$BN(x) = \gamma \frac{x - mean}{\sqrt{var + \varepsilon}} + \beta$$

During testing phase, I am using the moving mean and variance which was tracked during training and calculating batch norm values accordingly.

In order to calculate moving mean and variance, I have added a hyperparameter alpha_mov in the class which controls the impact of the current statistics and the old statistics.

The idea used for calculation is as follows –
moving_[mean/var] = α * old_moving_[mean/var] + (1 - α) * current_moving_[mean/var]

These values are later fixed for computation during validation or testing phase.
Optimizer used *SGD with learning rate of 0.1 and momentum 0.9*

Batch normalization function

```
  def batch_norm_train(self, Z, beta, gamma):
    # Z : result of pre or post activation
    #Calculating mean and variance
    # if test_mean is None then trigger training logic
    if self.test_mean is not None:
      mean_mini_batch = self.test_mean
      var_mini_batch = self.test_var
    else:
      mean_mini_batch = tf.reduce_mean(Z)
      var_mini_batch = tf.math.reduce_variance(Z)
      self.mov_mean.assign(self.alpha_mov * self.mov_mean + (1 - self.alpha_mov) *
mean_mini_batch)
      self.mov_var.assign(self.alpha_mov * self.mov_var + (1 - self.alpha_mov) *
var_mini_batch)

    Z_hat = tf.divide(tf.subtract(Z, mean_mini_batch), tf.sqrt(tf.add(var_mini_batch,
self.epsilon)))
    batch_norm = tf.add(tf.multiply(gamma, Z_hat), beta)
```

I have added two functions for pre activation and post activation respectively. Overall, batch norm has been applied on all three layers and their respective beta and gamma gets trained during backprop.

Further, I have coded separate functions for training and testing the model for ease of inference. Finally, I have reused my inference model from previous assignment as it captures all possible details that might be needed to gather intuition across the 3 seeds.

Inference across 3 trials

The suggested documentation mentions that batch normalization works best with batch sizes in the range 50-100, but for the purpose of this assignment I went ahead with batch size of 128 during inference. I tried few execution with batch_size = 64 and obtained similar results. Since, hyperparameter tuning wasn't the objective of the assignment so I have skipped further trials around these params

Following are the hyperparameters that I have used –
size_input = 784
size_hidden_1 = 128
size_hidden_2 = 128
side_hidden_3 = 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
learning_rates = {
   "SGD": 0.1,
}
```

I have written reused code snippets from previous assignment to further analyze the generated JSON and obtain intuitions 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##
```

	SGD_preact	SGD_postact
0	113.801344	113.142443
1	113.618386	112.915427
2	113.441578	113.206895

```
#Average CPU time per model#
SGD_preact_avg     113.620436
SGD_postact_avg    113.088255
```

The CPU time tends to stay almost similar across the seeds and between both pre and post activation models.

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

```
##Accuracy stats across trials##
   SGD_preact  SGD_postact
0      0.8744       0.8696
1      0.8731       0.8642
2      0.8681       0.8622

#Average Accuracy per model#
SGD_preact_avg     0.871867
SGD_postact_avg    0.865333
```

pre-activation models usually tend to perform slightly better than post-activation models across all seeds but the difference is comparable.

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#
SGD_preact     1.194319e-10
SGD_postact    9.581537e-10
```

The variance is loss is substantial in case of post-activation models when compared with those of pre-activation models.

MNIST – CNN

The idea stays the same in MNIST but here I went ahead with the standard Keras implementation of batch norm since the provided code scaffold also followed the same design.

So, I have made use of `tf.keras.layers.BatchNormalization()` to add a Batch Normalization layer which is placed accordingly as per pre and post activation modes

```
self.conv1 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
self.bn1 = tf.keras.layers.BatchNormalization()
self.conv2 = tf.keras.layers.Conv2D(64, 3,padding='same', activation=None)
self.bn2 = tf.keras.layers.BatchNormalization()
self.pool1 = tf.keras.layers.MaxPool2D()
self.conv3 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
self.bn3 = tf.keras.layers.BatchNormalization()
self.conv4 = tf.keras.layers.Conv2D(64, 3, padding='same', activation=None)
self.bn4 = tf.keras.layers.BatchNormalization()
```

This version of Keras batch normalization allows us to pass an argument `training = <True/False>` to signal that the execution is in train or test phase. Keras ensures that the running stats are used when training is set to False. As a result of the addition of 4 batch normalization layers, there are 24 variables that Keras tracks overall which includes weights(kernels) and biases for each layer (8 variables) and gamma, beta, moving mean and moving variance(4 * 4 = 16) for the normalization layers.

The final model handles both pre and post activation implementations via two different functions and controlled by a knob – *batch_norm*

Just like MLP, I have created two separate functions for training and testing any model and eventually added an inference model which utilizes the history data that was captured during training. I am updating the test results to the actual per model history for ease of coding.

Inference across 3 trials

test accuracies across seeds


```
{
  "postact":[
    0.970823,
    0.97682315,
    0.95377696
  ],
  "preact":[
    0.9658489,
    0.98344964,
    0.9601051
  ]
}
```


Variance in test accuracies across seeds between pre and post act : (9.86381e-05, 9.52996e-05)

As can be seen above, there's very less variance in accuracies across different seeds and on average both models show similar performance across all seeds

Test losses across seeds


```
{
  "postact":[
    0.035087366,
    0.03538838,
    0.022961156
  ],
  "preact":[
    0.03114769,
    0.031808954,
    0.040787797
  ]
}
```


Variance in test loss across seeds between pre and post act: (1.9332061e-05, 3.350794e-05)

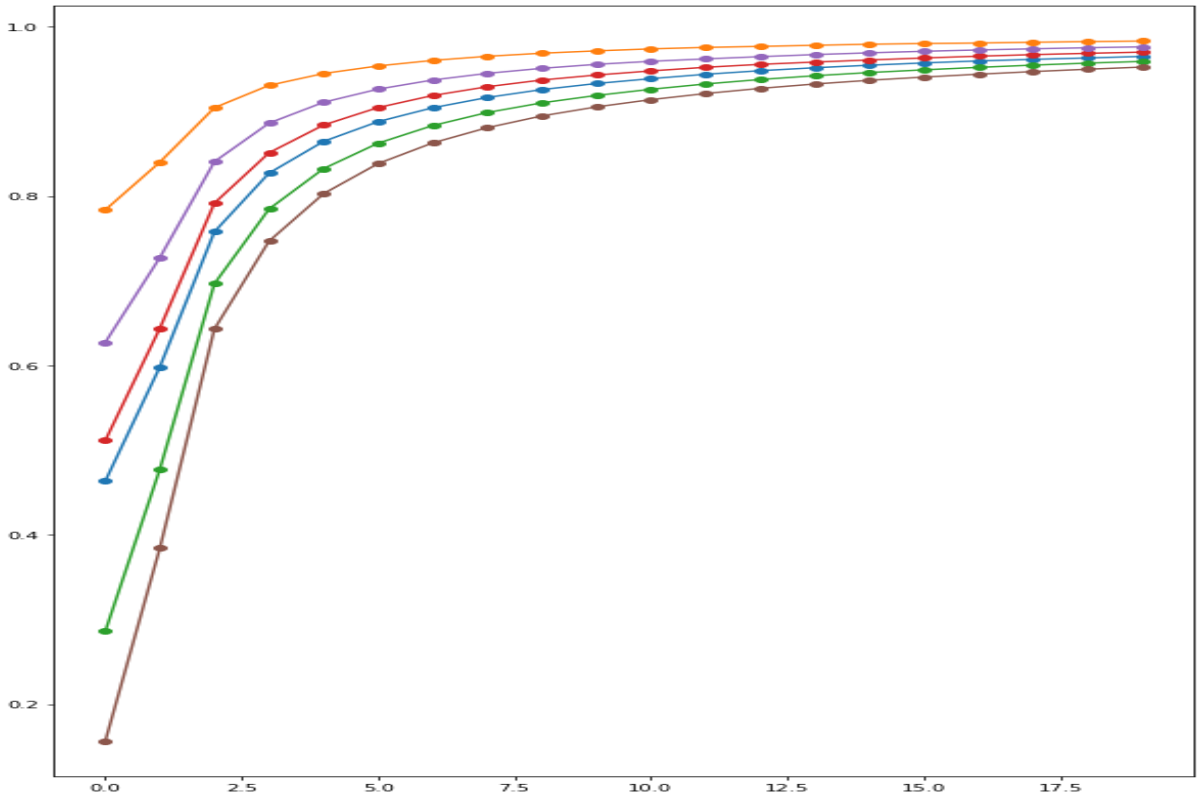
Overall, there's slightly more variance in losses across seeds in post activation for test results.

Average Train losses across seeds


```
{
  "postact":[
    0.108825244,
    0.07337411,
    0.23868184
  ],
  "preact":[
    0.09407832,
    0.046354193,
    0.10335843
  ]
}
```


Variance in average loss across seeds between pre and post act:
(0.00062368845, 0.0050495737)

accuracy vs epoch plot during training phase



Accuracies start off differently for different seeds but eventually they all converge to similar best accuracy by the end of 10th epoch.