PMDS603P Deep Learning Lab Experiment 5

August 2025

1 Work to do today

Note: Make a single PDF file of the work you are doing in a Jupyter notebook. Upload with the proper format. Please mention your name and roll no properly with the Experiment number on the first page of your submission.

Question1. Use the cifar10 dataset and do necessary pre-processing, and split the data into training, validation, and testing sets.

Create a new model using a sequential class with appropriate hidden layers and output layer neurons. Choose appropriate activation functions like sigmoid and relu, etc. And also an appropriate one in the output layer. Choose the error function appropriately and usue SGD as the optimizer. Include early stopping technique in your model and run the model for 500 epochs. Try to come up with a better model with decent accuracy.

Question 2: We will try to include some techniques for optimizing the training part. Let us see how one can include a learning rate schedule for SGD, what we discussed in the theory class.

Now, first we will try to see the linear learning rate scheduler we discussed in class. here, τ is taken as 5 and $\epsilon_0 = 0.01$ and $\epsilon_\tau = 0.001$. If you run this code, you can see the way in which we generate the learning rate for different iterations. Compare with what we did in theory and check whether they are the same.

```python
import tensorflow as tf

lr_schedule = tf.keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=0.01,
    decay_steps=5,
    end_learning_rate=0.001,
    power=1.0,
    cycle=False
)

for step in range(10):
    print("Step", step, "LR=", lr_schedule(step).numpy())
```

Now, to incorporate this into the previous Question 1. You have to create a schedule and define the parameters as given below, and use it inside the SGD optimizer you are using in your previous problem.

```
lr_schedule = tf.keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=0.01,  # starting LR
    decay_steps=5,               # steps to reach end LR
    end_learning_rate=0.001,     # final LR after decay
    power=1.0,                    # 1.0 → linear decay
    cycle=False
)

optimizer = SGD(learning_rate=lr_schedule)
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The polynomialdecay scheduler works in this fashion where power = 1 is the linear case.

$$LR(t) = \text{end\_lr} + (\text{initial\_lr} - \text{end\_lr}) \times \left(1 - \frac{t}{\text{decay\_steps}}\right)^{\text{power}}$$

We can go for quadratic as well. Quadratic decay (power=2.0) → starts slower than linear, then accelerates and drop near the end. This means:

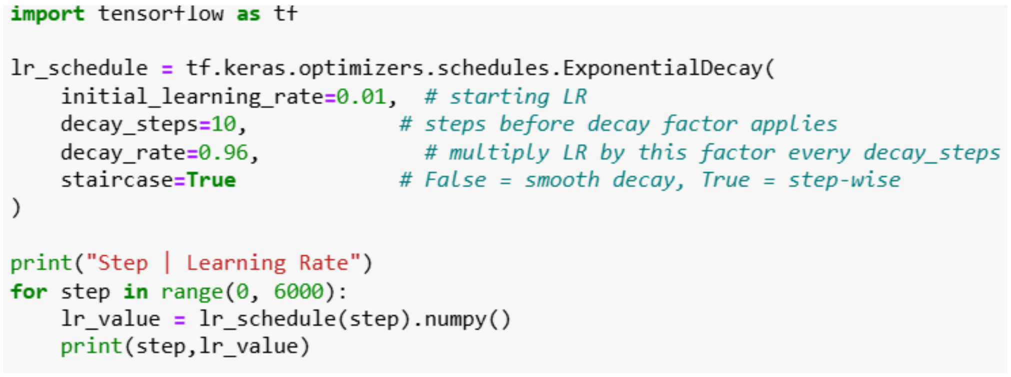
Early training: LR is higher for longer means the model can explore more, make bigger updates.
Later training: LR drops faster towards the end means fine-tuning happens in smaller steps.

Question 3: Try cifar10 dataset classification problem with polynomialdecay with power = 2 case and see if there are any improvements.

Question 4: Next is an exponentialdecay scheduler which uses this rule for decay. The decay_rate is again a hyperparameter.

$$LR(t) = \text{initial\_lr} \times (\text{decay\_rate})^{\frac{t}{\text{decay\_steps}}}$$

You can print and check to see the learning rates how its getting modified.

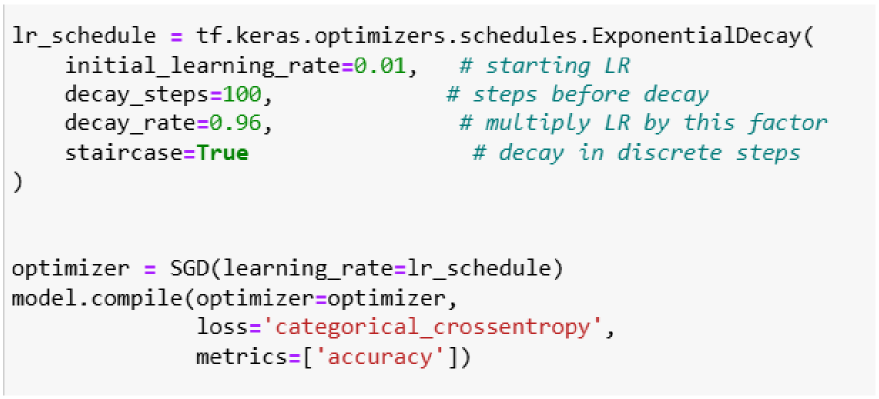


```
import tensorflow as tf

lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.01,  # starting LR
    decay_steps=10,            # steps before decay factor applies
    decay_rate=0.96,             # multiply LR by this factor every decay_steps
    staircase=True             # False = smooth decay, True = step-wise
)

print("Step | Learning Rate")
for step in range(0, 6000):
    lr_value = lr_schedule(step).numpy()
    print(step,lr_value)
```

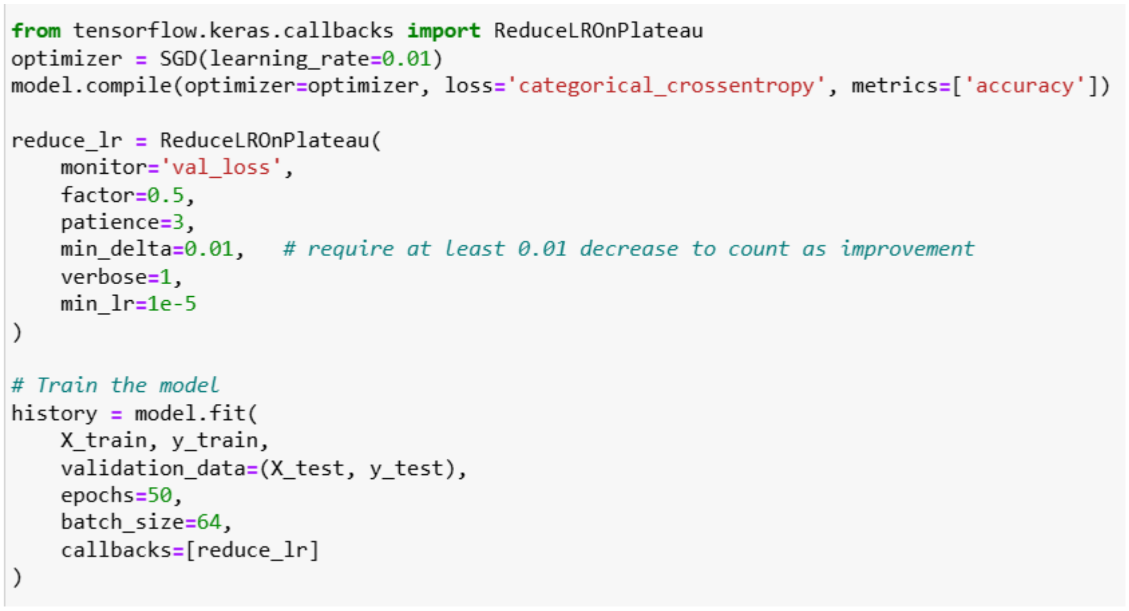
Further, you can incorporate in the previous problem by including these. Choose appropriate values for parameters.



```
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.01,   # starting LR
    decay_steps=100,             # steps before decay
    decay_rate=0.96,              # multiply LR by this factor
    staircase=True                 # decay in discrete steps
)


optimizer = SGD(learning_rate=lr_schedule)
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Question 5a: We can also use ReduceLRonPlateau to deal with the training part in a different way compared to last two types. Based on the validation loss we can decide when to reduce the learning rate if you are stuck with training part.

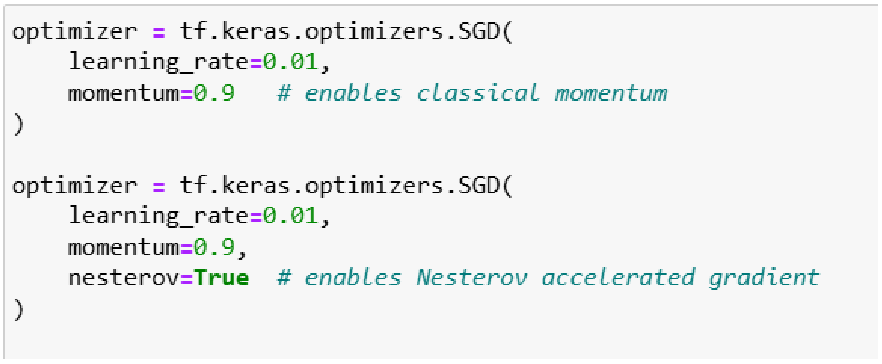


```
from tensorflow.keras.callbacks import ReduceLROnPlateau
optimizer = SGD(learning_rate=0.01)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    min_delta=0.01,   # require at least 0.01 decrease to count as improvement
    verbose=1,
    min_lr=1e-5
)

# Train the model
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=50,
    batch_size=64,
    callbacks=[reduce_lr]
)
```

Question 5: Now for adding momentum and nesterov momemtum you can try these in your question 1 soln and check your results in both the cases.



```
optimizer = tf.keras.optimizers.SGD(
    learning_rate=0.01,
    momentum=0.9   # enables classical momentum
)

optimizer = tf.keras.optimizers.SGD(
    learning_rate=0.01,
    momentum=0.9,
    nesterov=True  # enables Nesterov accelerated gradient
)
```

Challenging question: Try to hand-code how momentum and Nesterov momentum can be incorporated in a simple neural network model with two inputs, x1 and x2,(take random values) in the input layer and one neuron in the output layer. Note that there is no hidden layer. Use sigmoid as activation in the output layer.(use the error as mean squared error). The pseudocode is given for help as a separate sheet in Moodle. Use only if required.