Optimization

1 Remarks on optimization codes in Keras

Keras provides the *SGD* class that implements the stochastic gradient descent optimizer with a learning rate and momentum. First, an instance of the class must be created and configured, then specified to the "optimizer" argument when calling the fit() function on the model. The default learning rate is 0.01 and no momentum is used by default.

```
from keras.optimizers import SGD
...
opt = SGD()
model.compile(..., optimizer=opt)
```

The learning rate can be specified via the "lr" argument and the momentum can be specified via the "momentum" argument.

```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9)
model.compile(..., optimizer=opt)
```

The class also supports learning rate decay via the "decay" argument. With learning rate decay, the learning rate is calculated each update (e.g. end of each mini-batch) as follows:

$$\text{lrate} = \text{initial\_lrate} * (1 / (1 + \text{decay} * \text{iteration}))$$

Where `lrate` is the learning rate for the current epoch, initial_lrate is the learning rate specified as an argument to SGD, decay is the decay rate which is greater than zero and iteration is the current update number

```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9, decay=0.01)
model.compile(..., optimizer=opt)
```

Keras provides the *ReduceLROnPlateau* that will adjust the learning rate when a plateau in model performance is detected, e.g. no change for a given number of training epochs. This callback is designed to reduce the learning rate after the model stops improving with the hope of fine-tuning model weights. The ReduceLROnPlateau requires you to specify the metric to monitor during training via the "monitor" argument, the value that the learning rate will be multiplied by via the "factor" argument and the "patience" argument that specifies the

number of training epochs to wait before triggering the change in learning rate. For example, we can monitor the validation loss and reduce the learning rate by an order of magnitude if validation loss does not improve for 100 epochs:

```
# snippet of using the ReduceLROnPlateau callback
from keras.callbacks import ReduceLROnPlateau
rlrop = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience
    =100)
model.fig(..., callbacks=[rlrop])
```

Keras also provides *LearningRateScheduler* callback that allows you to specify a function that is called each epoch in order to adjust the learning rate.

You can define your Python function that takes two arguments (epoch and current learning rate) and returns the new learning rate.

```
# snippet of using the LearningRateScheduler callback
from keras.callbacks import LearningRateScheduler

def my_learning_rate(epoch, lrate):
  return lrate

lrs = LearningRateScheduler(my_learning_rate)
model.fit(..., callbacks=[lrs])
```

Adaptive Learning Rate Gradient Descent

Keras also provides a suite of extensions of simple stochastic gradient descent that support adaptive learning rates. Because each method adapts the learning rate, often one learning rate per model weight, little configuration is often required.
Three commonly used adaptive learning rate methods include:

- RMSPROP Optimiser:

  ```
  from keras.optimizers import RMSprop
  ...
  opt = RMSprop()
  model.compile(..., optimizer=opt)
  ```

- Adagrad Optimiser

  ```
  from keras.optimizers import Adagrad
  ...
  opt = Adagrad()
  model.compile(..., optimizer=opt)
  ```

- Adam Optimiser

  ```
  from keras.optimizers import Adam
  ...
  opt = Adam()
  model.compile(..., optimizer=opt)
  ```

2 Wine Quality Data Set

The to-be analyzed data set is related to white variants of the Portuguese "Vinho Verde" wine. We will use this data set as a classification task through this exercise. The attribute information are as:

1 - fixed acidity

2 - volatile acidity

3 - citric acid

4 - residual sugar

5 - chlorides

6 - free sulfur dioxide

7 - total sulfur dioxide

8 - density

9 - pH

10 - sulphates

11 - alcohol

Output variable or Target (based on sensory data):

12 - quality (score between 0 and 10)

The goal is to learn a model for predicting **wine quality** using its given parameters.

Exe. 1 First upload the `winequality-white.csv`. It is available in the platform. Then upload it in the panda data frame format with ; separator.

```
import pandas as pd
import io

wine = pd.read_csv(io.BytesIO(uploaded['winequality-white.csv']),
    sep=';')
```

Shuffle the `wine` data set using `sklearn.utils.shuffle()`. For the sake of simplicity for this exercise, get only the first 1000 elements of the wine dataframe.

Exe. 2 By using the following code, scatter plot your entire recovered dataset.

```
import matplotlib.pyplot as plt

plt.scatter(wine['fixed acidity'],
            wine['alcohol'],
            c= wine['quality'],
            cmap='viridis')

plt.xlabel("fixed acidity")
plt.ylabel("alcohol")
plt.show()
```

You can see that the classes are not linearly separable (separable by a line), causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different "good enough" candidate solutions.

Effect of Learning Rate and Momentum

In this section, we will develop a Multilayer Perceptron (MLP) model to address the blobs classification problem and investigate the effect of different learning rates and momentum.

Learning Rate Dynamics

Exe. 3 Using the `iloc` in panda dataframe get the first 11 columns of *wine* dataset as `X` and the final column as `y`. Then convert the dataframe to matrix using `X.values` and `y.values`.
Finally convert `y` values to the categorical types using the following code i.e. for the y labels between 0 and 10, 2 should be represented as a 10 dimensional vector of 0 and 1s as: $[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$. This is called one hot encoding of the target variable so that we can develop a model that predicts the probability of an example belonging to each class.

```
from keras.utils import to_categorical
y = to_categorical(y)
```

Exe. 4 Divide the data set to train and test set by selecting the first 80% of elements as train and the rest as the test set.

```
n_train = # this value should be computed by you
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

check the `trainX, testX, trainy, testy` shapes, and see if they are coherent.
Exe. 5 Define a simple multi layer perceptron (MLP) model that expects 11 input variables from the wine dataset, has a single hidden layer with 50 nodes, and an output layer with 10 nodes to predict the probability for each of the ten classes. Nodes in the hidden layer will use the rectified linear activation function (ReLU), whereas nodes in the output layer will use the softmax activation function.
Exe. 6 using the stochastic gradient descent optimizer and a learning rate be specified by you. The model will be trained to minimize cross entropy.

```
from keras.optimizers import SGD
# compile model
lrate = # choose a learning rate between 0 and 1 for your test
model.compile(loss='categorical_crossentropy', optimizer=keras.
    optimizers.SGD(learning_rate= lrate), metrics=['accuracy'])
```

Exe. 7 Fit the model for 200 training epochs, found with a little trial and error, and the test set will be used as the validation dataset so we can get an idea of the generalization error of the model during training.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy),
     epochs=200, verbose=0)
```

Exe. 8 Once fit, we will plot the accuracy of the model on the train and test sets over the training epochs using the following code:

```
# plot learning curves
plt.plot(history.history['accuracy'], label='train',color = 'r')
plt.plot(history.history['val_accuracy'], label='test', color = 'b'
    )
plt.title('lrate='+str(lrate), pad=-50)
```

Exe. 9 write a function namely `fit_model(trainX, trainy, testX, testy, lrate)` that ties the exercises 5 to 8 together. In general, it will fit a model and plot its performance given the train and test datasets as well as a specific learning rate to evaluate.

Exe. 10 In this exercise, we evaluate learning rates on a logarithmic scale from $1E-0$ (1.0) to $1E-7$ (0.000001) and create line plots for each learning rate by calling the `fit_model()` function.

```
# create learning curves for different learning rates
learning_rates = [1E-0, 1E-1, 1E-2, 1E-3, 1E-4, 1E-5, 1E-6, 1E-7]
for i in range(len(learning_rates)):
  # determine the plot number
  plot_no = 420 + (i+1)
  plt.subplot(plot_no)
  # fit model and plot learning curves for a learning rate
  fit_model(trainX, trainy, testX, testy, learning_rates[i])
  # show learning curves
  plt.show()
```

Classification accuracy on the training dataset is marked in blue, whereas accuracy on the test dataset is marked in orange. Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times.

Question: Check your results for various learning rate vale. Which learning rate is more suitable for your model? which one is not helpful?

Momentum Dynamics

Momentum can smooth the progression of the learning algorithm and, in turn, can accelerate the training process.

Exe. 11 choose the best learning rate from the previous exercise which converged to a reasonable solution. Modify the `fit_model(trainX, trainy, testX, testy, momentum))` function to take a "momentum" argument instead of a learning rate argument, that can be used in the configuration of the SGD class and reported on the resulting plot. And keep the learning rate as your best parameter of the exe. 10.

Exe. 12 It is common to use momentum values close to 1.0, such as 0.9 and 0.99. Demonstrate the dynamics of the model without momentum compared to

the model with momentum values of 0.5 and the higher momentum values. Similar to Exe. 10, demonstrate different accuracy changing vs. epochs graphs on various momentum values as `momentums = [0.0, 0.5, 0.9, 0.99]` and using the modified `fit_model()` function. Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times.
Normally, the addition of momentum does accelerate the training of the model. In all cases where momentum is used, the accuracy of the model on the holdout test dataset is more stable, showing less volatility over the training epochs. How are your results? how do you interpret them?

Learning Rate Decay

The SGD class provides the "decay" argument that specifies the learning rate decay.
Exe. 13 Implement a learning rate decay function using the following formula:

$$initialrate \times (1.0/(1.0 + decay \times iteration))$$

for three input variables `initial_rate, decay` and `iteration`. This function will be used to calculate the learning rate over multiple updates with different decay values. Run the following code:

```
decays = [1E-1, 1E-2, 1E-3, 1E-4]
lrate = 0.01
n_updates = 200
for decay in decays:
  # calculate learning rates for updates
  lrates = [decay_lrate(lrate, decay, i) for i in range(n_updates)]
  # plot result
  plt.plot(lrates, label=str(decay))
plt.legend()
plt.show()
```

Running the example creates a line plot showing learning rates over updates for different decay values. After checking results, we can seen that in all cases, the learning rate starts at the initial value of 0.01. And a small decay value of $1E-4$ has almost no effect, whereas a large decay value of $1E-1$ has a dramatic effect, We can see that the change to the learning rate is not linear. We can also see that changes to the learning rate are dependent on the batch size, after which an update is performed.
Exe. 14 similar to Exe. 11, modify the `fit_model()` function to take a 'decay' argument which can be used to configure decay for SGD class i.e. `keras.optimizers.SGD(learning_rate= , decay= )`
Exe. 15 Similar to exe. 10 and 12, evaluate the accuracy changing of train ans test sets for $[1E-1, 1E-2, 1E-3, 1E-4]$ decay values and their effect on model accuracy. Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times.
Normally, the large decay values decay the learning rate too rapidly for the

model on the problem and result in poor performance. The larger decay values do result in worse performance. How are your results and interpretations?

Effect of Adaptive Learning Rates

Learning rates and learning rate schedules are both challenging to configure and critical to the performance of a deep learning neural network model.

Keras provides a number of different popular variations of stochastic gradient descent with adaptive learning rates, such as:

- Adaptive Gradient Algorithm (AdaGrad).
- Root Mean Square Propagation (RMSprop).
- Adaptive Moment Estimation (Adam).

Each provides a different methodology for adapting learning rates for each weight in the network. There is no single best algorithm, and the results of racing optimization algorithms on one problem are unlikely to be transferable to new problems. We can study the dynamics of different adaptive learning rate methods on the wine quality dataset.

Exe. 16 Update the `fit_model(trainX, trainy, testX, testy, optimizer)` function to take the name of an optimization algorithm to evaluate, which can be specified to the "optimizer" argument when the MLP model is compiled. The default parameters for each method will then be used.

Exe. 17 Similar to exe. 10, 12 and 15 explore the three methods of RMSprop, AdaGrad and Adam and compare their behavior to simple stochastic gradient descent with a static learning rate. Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times. What are your interpretations? which method learns better? which one is the worst?