4.3.4 Momentum Optimizer

In order to improve this, a new version of optimizers was born – **Momentum Optimizers**. Essentially, this **idea** was born back in 1974 by Boris T. Polyak and the optimizers are focused on helping models converge as **fast** as possible. The idea comes from observing a ball rolling down a gentle slope. It will start off slowly, but it will gain **momentum**. So, the core of the idea is to focus on movement in the correct direction. This is done by reducing the **variance** in every other insignificant direction, thus accelerating gradient descent.

Moment Optimization introduces the **momentum vector**. This vector is used to "store" changes in previous gradients. It helps **accelerate** the stochastic gradient descent in the relevant direction and dampens oscillations. At each gradient step, the local gradient is **added** to the momentum vector. Then, parameters are updated just by subtracting the momentum vector from the current parameter values. Since the whole idea kinda comes from physics, a new hyperparameter β is introduced – **momentum**. It simulates the friction mechanism and regulates the momentum value so it doesn't explode. Typically, this value is set to 0.9. To sum up, momentum optimization is performed in two steps:

1. Calculating the momentum vector at each iteration using the formula:

$$m \leftarrow \beta m + \alpha \nabla MSE$$

where m is the momentum vector, β is the momentum, α is the learning rate, θ is the set of machine learning parameters and ∇MSE is the partial derivative of the cost function (**Mean Squared Error**, in this case).

2. Update the parameters by subtracting the momentum vector.

$$\theta \leftarrow \theta - m$$

4.3.4.1 Python Implementation

We implement this algorithm within *MyMomentumOptimizer* class:

```python
class MyMomentumOptimizer():
    def __init__(self, learning_rate, momentum = 0.9):
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.w = 0
        self.b = 0

        self.momentum_vector_w = 0
        self.momentum_vector_b = 0

    def _get_batch(self, X, y, batch_size):
        indexes = np.random.randint(len(X), size=batch_size)
        return X[indexes,:], y[indexes,:]

    def _get_momentum_vector(self, X_batch, y_batch):
        f = y_batch - (self.w * X_batch + self.b)

        self.momentum_vector_w = self.momentum * self.momentum_vector_w + \
              self.learning_rate * (-2 * X_batch.dot(f.T).sum() / len(X_batch))
        self.momentum_vector_b = self.momentum * self.momentum_vector_b + \
              self.learning_rate * (-2 * f.sum() / len(X_batch))

    def fit(self, X, y, batch_size = 32, epochs = 100):
        history = []

        for e in range(epochs):

            indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)

            self._get_momentum_vector(X_batch, y_batch)

            self.w -= self.momentum_vector_w
            self.b -= self.momentum_vector_b

            loss = mean_squared_error(y_batch, (self.w * X_batch + self.b))

            if e % 100 == 0:
                print(f"Epoch: {e}, Loss: {loss})")

            history.append(loss)

        return history

    def predict(self, X):
        return self.w * X + self.b
```

Ugh, that is a lot of code; let's segment it and explain piece by piece. In the **constructor** of this class, we initialize the necessary attributes and set **hyperparameter** values. The learning rate and momentum are set, and algorithm parameters *w* and *b* are initialized to 0. The same goes for momentum vectors. Note that we could put all the parameters of the algorithm (*w and b*) within one array, but we wanted everything to be as clear as possible. The code can, of course, be improved.

```
def __init__(self, learning_rate, momentum = 0.9):
      self.learning_rate = learning_rate
      self.momentum = momentum
            self.w = 0
      self.b = 0
            self.momentum_vector_w = 0
      self.momentum_vector_b = 0
```

Two private methods **_get_batch** and **_get_momentum_vector**, are very important. The *_get_batch* method is used to pick the batch from the input and output datasets. The *_get_momentum_vector* method does the dirty work we defined in the momentum algorithm. Here, the gradient for each parameter is calculated and used to update the momentum vector for each parameter.

```
def _get_batch(self, X, y, batch_size):
      indexes = np.random.randint(len(X), size=batch_size)
      return X[indexes,:], y[indexes,:]

def _get_momentum_vector(self, X_batch, y_batch):
      f = y_batch - (self.w * X_batch + self.b)
            self.momentum_vector_w = self.momentum * self.momentum_vector_w +
\
                          self.learning_rate * (-2 * X_batch.dot(f.T).sum() /
len(X_batch))
      self.momentum_vector_b = self.momentum * self.momentum_vector_b + \
                          self.learning_rate * (-2 * f.sum() / len(X_batch))
```

Finally, in the *fit* method, the actual training is performed. For each epoch, this algorithm first fetches the batch and calculates the momentum vectors for each parameter. In the end, *w* and *b* are **updated** using these vectors. Apart from that, we calculate the loss, print it out, and store it in history. This is done just so we can follow what is happening during the **training** process. The method *predict* just predicts values for the input.

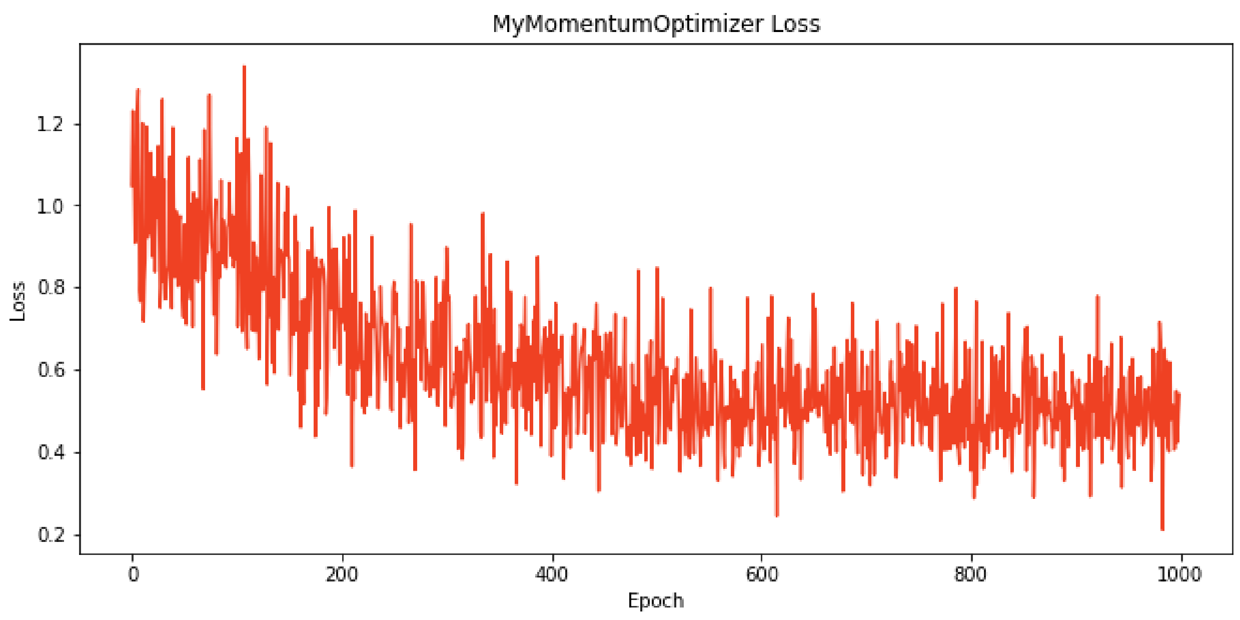
```
def fit(self, X, y, batch_size = 32, epochs = 100):
      history = []
            for e in range(epochs):
                        indexes = np.random.randint(len(X), size=batch_size)
            X_batch, y_batch = self._get_batch(X, y, batch_size)
                        self._get_velocity(X_batch, y_batch)
                        self.w -= self.velocity_w
            self.b -= self.velocity_b
                  loss = mean_squared_error(y_batch, (self.w * X_batch +
self.b))

            if e % 100 == 0:
                  print(f"Epoch: {e}, Loss: {loss})")
                        history.append(loss)

      return history
                  def predict(self, X):
      return self.w * X + self.b
```

Now when we understand what is happening in this class, let's run this on loaded data:

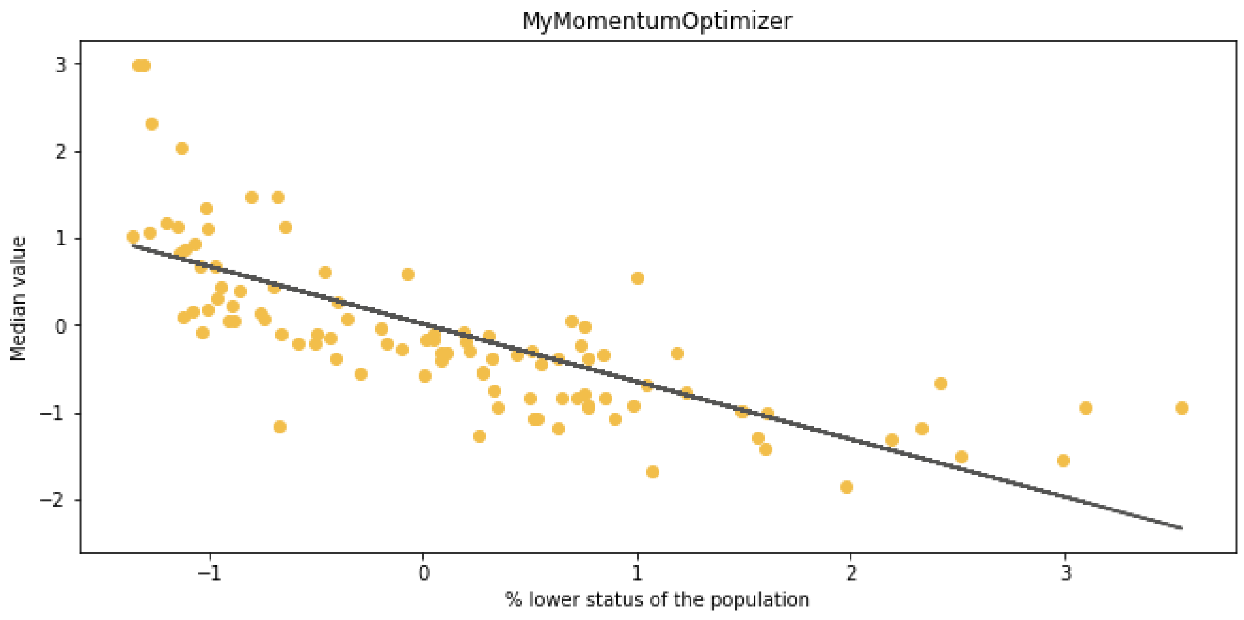
```
model = MyMomentumOptimizer(learning_rate = 0.0001)
history = model.fit(X_train, y_train, batch_size = 128, epochs = 1000)

predictions = model.predict(X_test)
```

```
Epoch: 0, Loss: 1.023970350020749)
Epoch: 100, Loss: 0.8915269977932345)
Epoch: 200, Loss: 0.6160710091923012)
Epoch: 300, Loss: 0.7748865586040383)
Epoch: 400, Loss: 0.7624729068019713)
Epoch: 500, Loss: 0.7386694964887148)
Epoch: 600, Loss: 0.4590078364639633)
Epoch: 700, Loss: 0.45645337845296935)
Epoch: 800, Loss: 0.548081143220535)
Epoch: 900, Loss: 0.5855308679385105)
```

We can note that this algorithm goes to the global minimum quite quickly, which is what we wanted. Also, note that it is almost the same as the momentum optimizer. It goes fast "ahead" and then backs up. If we plot the loss history, we get something like this:



And if we plot the model it looks something like this:



Seems quite cool that we were able to get really good results with this approach, taken into consideration that we are using Linear Regression as our chosen algorithm.

4.3.5 Nesterov Accelerated Gradient

Yurii Nesterov noticed, back in 1983, that it is possible to improve momentum-based