HW2_CV2_dh3027

March 2, 2022

1 COMS W4732 Homework 2: Machine Learning Basics

Following the machine learning content covered in class, in this assignment we will explore some crucial concepts in gradient descent and backpropgation.

Specifically, in Section 1, we will work with a **feedforward neural network** (also known as a **multi-layer perceptron**) implemented solely in numpy, reflect on the associated details of the **forward** pass and implement the **backpropagation** parts of our layers to train our MLP. In Section 2, we will independently look at some gradient-based optimization techniques that are popular with convex functions and have been shown to be useful in finding sufficiently satisfactory optima on the loss manifolds on parametrized models.

Your job is to implement the sections marked with `TODO` to complete the tasks. Your tasks on this homework will be:

1.1 Section 1 (40 points)

Review the details on the chain rule and the backproprapagation step from lecture. You should take a look at the guide we provide to get familiar with the forward pass and backward pass equations for a Multi-Layer Perceptron. You are required to implement all parts marked with a **TO DO:**. The goal of this assignment is to leave you with a very through understanding of forward pass as well as the backpropgation mechanics of a Multi-layer Peceptron. Namely, you will be working with:

- a **linear** layer with **Leaky ReLU**
- a **linear** layer with a custom activation function that has a learnable parameter
- a **linear** output layer
- a **softmax cross-entropy** Loss layer

1.2 Section 2 (60 points).

We will introduce different gradient-based iterative optimization techniques which came from the domain of convex optimization and which have since been adapted for loss functions of modern neural networks that have millions of parameters. Try out a few of these to appreciate the improvements they make on each other. Specifically, we will look at:

- **Full Gradient Descent**
- **Stochastic Gradient Descent**
- **Stochastic Gradient Descent + Momentum**
- **AdaGrad**
- **Adaptive Moment Estimation (ADAM)**

1.2.1 About Submission

- Please submit the notebook (ipynb and pdf) including the output of all cells. Please note that you should export your completed notebook as a PDF (CV2_HW2_UNI.pdf) and upload it to GradeScope.
- Then, please submit the executable completed notebook (CV2_HW2_UNI.ipynb) to Cousework.
- For both files, 1) remember to replace with your own uni; 2) the completed notebook should show your code, as well as the final image you created.

1.2.2 Before your implementation

- Please check the packages listed in the **requirements.txt**. You can also use `pip install -r requirements.txt` to install the packages directly.

2 Section 1: Backpropagation

This assignment is aimed at leaving you with a very solid understanding of how backpropogation works in the context of a Neural Network. We have provided most of the code for a MLP (Multi-layer Perceptron) written completely in Numpy with some functions left for you to implement to get the network up and running.

On correct completion, you will be able to successfully train your MLP for any classification problem where the input feature vector is relatively low-dimensional. We've provided code that pre-processes and trains your MLP on the **Red Wine Classification** dataset for binary-classification. This is to enable you to quickly run and test your MLP.

2.1 Lets get to it!

As the building blocks of our MLP, we define three layer classes: Hidden, Output and Loss. Each of these inherit from our Base class and thus implement self.forward_pass(), self.backward_pass(), and contain self.update_weighhts(), which is already implemented.

Before we dive in, we would like to highlight the distinction between the Hidden Layer and the Hidden_Vondrick layer as you will see defined below in the code.

2.1.1 Hidden Layer

This is a standard Hidden Layer that uses Leaky ReLU as its activation function. Remember that Leaky ReLU is defined as a piecewise function:

$$g(x) = \begin{cases} x : & x > 0, \\ 0.01x : & x <= 0 \end{cases}$$

2.1.2 Hidden_Vondrick Layer (utilizes cuztom activation function with learnable parameter)

This layer is similar to the standard Hidden layer, but with one notable exception. We will use a custom activation function g'(x). Furthermore, we will make exponent parameter 'n' learnable, and update it also leveraging the chain rule and backpropgation.

$$g'(x) = \begin{cases} x^n : & x > 0, \\ 0.01x : & x <= 0 \end{cases}$$

2.2 Instructions

Go over the code for the MLP throughly and understand each update equation implemented as code. The forward_pass() method is implemented for you for every layer and thus you may find printing out variables and their shapes useful, before you begin implementing the backpropogation methods. Having a clear understanding of the forward and backward pass formulae is crucial for this Section. Your job is to implement only sections marked as **TO DO:** (7 in total) (The PDF in the zip file is for your reference)

```python
[90]: import numpy as np
      import random
      import pandas as pd
      import matplotlib.pyplot as plt
      %matplotlib inline
```

```python
[91]: import numpy as np
      import random
      import pandas as pd
      import matplotlib.pyplot as plt


      class Base:
          def __init__(self, input_dims: int, output_dims: int):
              self.input_dims = input_dims
              self.output_dims = output_dims

          def forward_pass(self):
              pass

          def backward_pass(self):
              pass

          def update_weights(self, W, b, del_W, del_b, learning_rate):
              W -= (learning_rate * del_W)
              b -= (learning_rate * del_b)
              return W, b


      class Hidden(Base):
          def __init__(self, input_dims: int, output_dims: int):
              super().__init__(input_dims, output_dims)

              self.W = np.random.random((input_dims, output_dims)) - 0.5
              self.c = np.random.random((1, output_dims)) - 0.5
```

```python
    def forward_pass(self, X):
        U = X @ self.W + self.c
        activations = self.leaky_relu(U)
        return activations

    def backward_pass(self, X, h, dLdh, alpha, learning_rate):
        """
        TO DO:  Finish this backward_pass method by completing the lines marked
→by the #s.
        Remember to use the helper functions update_weights(), leaky_relu, and
→leaky_relu_derivative wherever needed.
        """

        relu_derivative = self.leaky_relu_derivative(h)
        dLdU =  np.multiply(dLdh, relu_derivative)#
        dLdW =  np.dot(X.T, dLdU) + alpha*self.W#
        dLdc =  np.sum(dLdU,axis=0)/dLdU.shape[0]#
        for_prev = np.dot(dLdU,self.W.T) #

        self.W, self.c = self.update_weights(self.W, self.c, dLdW, dLdc,
→learning_rate)
        return for_prev
        """Note: for_prev is the gradient dL/dh' we pass onto the previous
→layer h' """

    def leaky_relu(self, inp):
        activation_mask = 1.0 * (inp > 0) + 0.01 * (inp < 0)
        activations = np.multiply(inp, activation_mask)
        return activations

    def leaky_relu_derivative(self, h):
        """
        TO DO: Implement the leaky_relu_derivative method.
        This should return a numpy ndarray of shape (batch_size x self.
→output_dims)
        """
        leaky_relu_derivative = 1.0*(h>0) + 0.01*(h<0)
        return leaky_relu_derivative


class Hidden_Vondrick(Base):
    def __init__(self, input_dims: int, output_dims: int):
        super().__init__(input_dims, output_dims)

        self.W = np.random.random((input_dims, output_dims))
        self.c = np.random.random((1, output_dims))
        self.U = None
```

```python
        self.vondrick_exponent = np.random.uniform(1.4,
                                                    2)  # The learnable
→exponent, called Vondrick Exponent, for our custom activation function is
→initilized from a Unifom(1.4,2) distribution. You may change this if you
→really want to, but keep it close to this range to ensure training stability.

        print("Intital Value of Vondrick Exponent: " + str(self.
→vondrick_exponent))

    def forward_pass(self, X):
        self.U = X @ self.W + self.c
        # Applys the custom activation elementwise
        activations = self.vondrick_activation(self.U)
        return activations

    def backward_pass(self, X, h, dLdh, alpha, learning_rate=0.0005):
        """
        # TO DO: Fill in this  backward pass method.
        """
        vondrick_derivative, dL_dexponent_scalar = self.
→vondrick_activation_derivative()
        dLdU = np.multiply(dLdh, vondrick_derivative)  #
        dLdW = np.dot(X.T, dLdU) + alpha * self.W  #
        dLdc = np.sum(dLdU, axis=0)  #
        for_prev = np.dot(dLdU, self.W.T)  #

        # Note, that for the purposes of training stablity, we have hard-coded
→the learning rate here to be 0.0005
        self.W, self.c = self.update_weights(self.W, self.c, dLdW, dLdc, 0.0005)

        # Gradient Descent on our learnable activation function parameter:
→Updating exponent, but clipping it's lower range to 1.01

        self.vondrick_exponent = max(1.01, self.vondrick_exponent - 0.001 *
→dL_dexponent_scalar)

        return for_prev

    def vondrick_activation(self, U):
        """
        TO DO:
        Implement this helper function that the forward pass uses compute to
→compute the activation map for the given input U.

        return activations: (batch_size x output_dims)
        """
```

```python
        vondrick = np.vectorize(lambda x: x**self.vondrick_exponent if x>0 else
 0.01*x)
        activations = vondrick(U)
        return activations

    def vondrick_activation_derivative(self):
        """
        # TO DO:
        Implement this helper function that uses the stored self.U to do a
 backward pass and return both dh/dU and dh/dexponent. Both should be numpy
 matrices dimensions batch_size x self.output_dims

        return activations: (batch_size x self.output_dims)
        """
        U_func = np.vectorize(lambda x: self.vondrick_exponent*x**(self.
 vondrick_exponent-1) if x>0 else 0.01)
        exp_func = np.vectorize(lambda x: np.log(x)*x**self.vondrick_exponent
 if x>0 else 0)
        derivative_wrt_U = U_func(self.U)
        derivative_wrt_exponent = np.sum(exp_func(self.U))

        return derivative_wrt_U, derivative_wrt_exponent


class Output(Base):

    def __init__(self, input_dims, output_dims):
        super().__init__(input_dims, output_dims)
        self.w = np.random.random((input_dims, output_dims)) - 0.5
        self.b = np.random.random((1, output_dims)) - 0.5

    def forward_pass(self, h):
        z = h @ self.w + self.b
        z = z - np.max(z, axis=1).reshape(z.shape[0],
                                          1)  # trick: subtracting maz z as
 softmax is not effected: prevents overflow when we do exponentation
        return z

    def backward_pass(self, h, dLdz, alpha, learning_rate):
        """
        # TO DO: Implement the backward pass for the output layer.
        Finally, update the Weight matrix and bias vector appropriately, and
 then return dLdh, which will be passed backed to previous layers during
 backpropgation
        """
        dLdw = np.dot(h.T, dLdz) + alpha * self.w
```

```python
        dLdb = np.sum(dLdz, axis=0)/dLdz.shape[0]
        dLdh = np.dot(dLdz, self.w.T)
        self.w, self.b = self.update_weights(self.w, self.b, dLdw, dLdb,
→learning_rate)
        return dLdh


class Loss(Base):

    def __init__(self, input_dims, output_dims):
        super().__init__(input_dims, output_dims)

    def forward_pass(self, z, y):
        temp = -z + np.log(np.sum(np.exp(z), axis=1)).reshape(z.shape[0],
                                                1)  # Computing
→Softmax Cross Entropy Loss terms for each z_i. Note dimensions of temp:
→batch_size x output layer output_dims
        L = temp[np.arange(z.shape[0]), y.flatten().astype(
            int)]  # Extracts Loss term corresponding only to ground truth
→class from each row (sample).
        L = np.mean(L)  # Mean Loss over the batch
        return L

    def backward_pass(self, z, y):
        # Recall the simplified expression we get for dL_i/dz_k= p_k- I(y_i=k)
→(Details in the guide)
        temp1 = np.zeros(z.shape)
        for i in range(z.shape[0]):
            true_class = int(y[i].item())
            temp1[i][true_class] = -1  # -1 is added to the loss term
→corresponding to the true class

        temp2 = np.exp(z) / np.sum(np.exp(z), axis=1).reshape(z.shape[0],
                                                1)  # Matrik of
→p_k terms, aka, elements replaced by softmaxed probabilities
        for_previous = temp1 + temp2
        return for_previous


class NN:
    def __init__(self):
        self.output_layer = self.loss_layer = None
        self.hidden_layers = []

    def add_layer(self, name, input_dims, output_dims):
        if name.lower() == 'hidden':
```

```python
                self.hidden_layers.append(Hidden(input_dims, output_dims))
            elif name.lower() == 'hidden_vondrick':
                self.hidden_layers.append(Hidden_Vondrick(input_dims, output_dims))
            elif name.lower() == 'output':
                self.output_layer = Output(input_dims, output_dims)
            elif name.lower() == 'loss':
                self.loss_layer = Loss(input_dims, output_dims)

    def forward_prop(self, X, y, alpha):
        hidden_outputs = []
        z = L = h = None
        for layer in self.hidden_layers:
            h = layer.forward_pass(X)
            hidden_outputs.append(h)
            X = h
        z = self.output_layer.forward_pass(h)
        L = self.loss_layer.forward_pass(z, y)
        for layer in self.hidden_layers:
            L += 0.5 * alpha * np.linalg.norm(layer.W) ** 2
        L += 0.5 * alpha * np.linalg.norm(self.output_layer.w) ** 2
        return hidden_outputs, z, L

    def backward_prop(self, X, hidden_outputs, z, y, alpha=0.01,␣
↪learning_rate=0.01):
        dLdz = self.loss_layer.backward_pass(z, y)
        for_previous = self.output_layer.backward_pass(hidden_outputs[-1],␣
↪dLdz, alpha, learning_rate)
        for i in range(len(self.hidden_layers) - 1, 0, -1):
            temp = self.hidden_layers[i].backward_pass(hidden_outputs[i - 1],␣
↪hidden_outputs[i], for_previous, alpha,
                                                        learning_rate)
            for_previous = temp
        self.hidden_layers[0].backward_pass(X, hidden_outputs[0], for_previous,␣
↪alpha, learning_rate)

    def train(self, X, y, epochs, batch_size, learning_rate, alpha,␣
↪show_training_accuracy=True):

        loss = []
        for epoch in range(epochs):
            predicted = self.predict(X)
            correct = 0
            for i in range(len(predicted)):
                if predicted[i] == y[i]:
                    correct += 1
            if show_training_accuracy:
```

```python
                print(f'the accuracy on the training data after epoch {epoch +␣
 ↪1} is {correct / X.shape[0]}')
            temp = total = 0
            for k in range(0, X.shape[0], batch_size):
                inp = X[k:k + batch_size]
                out = y[k:k + batch_size]

                hidden_outputs, z, L = self.forward_prop(inp, out, alpha)
                temp += L
                total += 1
                self.backward_prop(inp, hidden_outputs, z, out, alpha,␣
 ↪learning_rate)

            loss.append(temp / total)

        return loss

    def predict(self, X):
        """
        TO DO:
        Implement the predict() method that takes in a batch input X␣
 ↪(number_of_samples x feauture_vector_dims) and returns an nparray y of␣
 ↪predictions (number_of_samples x 1)

        """
        for layer in self.hidden_layers:
            h = layer.forward_pass(X)
            X = h

        z = self.output_layer.forward_pass(X)
        predictions = np.argmax(z, axis=1)


        return predictions

    def compute_accuracy(self, X, Y):
        predicted_Y = self.predict(X)
        correct = 0
        for i in range(len(predicted_Y)):
            if predicted_Y[i] == Y[i]:
                correct += 1
        return correct / len(Y)


def plot_loss(loss_li):
    # Given a list of losses over the epochs, plots the loss curve.
    plt.xlabel("epoch")
```

```
        plt.ylabel("loss")
        plt.title("loss of the neural network per epoch")
        plt.plot(loss_li)
        plt.show()
```

3 Testing your MLP: Red Wine Quality Classification Dataset

3.1 More about the dataset:

https://archive.ics.uci.edu/ml/datasets/wine+quality

```
[92]: from sklearn.neural_network import MLPClassifier
      from sklearn.preprocessing import MinMaxScaler
      from sklearn.model_selection import train_test_split
```

```
[93]: # If you are using colab, pls refer to commands below for file uploading.
      # Otherwise, just ignore it.
      '''
      from google.colab import files
      uploaded = files.upload()
      '''
      wine_dataset = pd.read_csv('./winequality-red.csv')
```

```
[94]: #Converting Labels to a Binary Classification Problem
      def Convert_Labels(data):
          data.loc[:,'quality'] = np.where(data.loc[:,'quality']>=6, 1, 0)
          return data

      #Scales features to constrain them to lie within the default range (0,1)
      def DataScaler(data):
          scaler = MinMaxScaler()
          data = scaler.fit_transform(data)
          return data

      all_columns = list(wine_dataset)
      target = ['quality']
      print(all_columns)
      features = list(set(all_columns)-set(target))
      print(features)
      wine_dataset.loc[:,features] = DataScaler(wine_dataset.loc[:,features])

      wine_dataset.head()
```

```
['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH',
'sulphates', 'alcohol', 'quality']
['fixed acidity', 'density', 'pH', 'volatile acidity', 'free sulfur dioxide',
```

10

```
        'residual sugar', 'citric acid', 'total sulfur dioxide', 'alcohol', 'sulphates',
        'chlorides']
```

```
[94]:    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
      0       0.247788          0.397260         0.00        0.068493   0.106845
      1       0.283186          0.520548         0.00        0.116438   0.143573
      2       0.283186          0.438356         0.04        0.095890   0.133556
      3       0.584071          0.109589         0.56        0.068493   0.105175
      4       0.247788          0.397260         0.00        0.068493   0.106845

         free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
      0             0.140845              0.098940  0.567548  0.606299   0.137725
      1             0.338028              0.215548  0.494126  0.362205   0.209581
      2             0.197183              0.169611  0.508811  0.409449   0.191617
      3             0.225352              0.190813  0.582232  0.330709   0.149701
      4             0.140845              0.098940  0.567548  0.606299   0.137725

          alcohol  quality
      0  0.153846        5
      1  0.215385        5
      2  0.215385        5
      3  0.215385        6
      4  0.153846        5
```

```
[95]:  label_converted_dataset = Convert_Labels(wine_dataset)
       print(label_converted_dataset)
       #As you can see, the quality column (our labels) now has either 0 (for␣
       ↪quality<6) and 1 (for quality>=6)
```

```
           fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
      0          0.247788          0.397260         0.00        0.068493   0.106845
      1          0.283186          0.520548         0.00        0.116438   0.143573
      2          0.283186          0.438356         0.04        0.095890   0.133556
      3          0.584071          0.109589         0.56        0.068493   0.105175
      4          0.247788          0.397260         0.00        0.068493   0.106845
      ...             ...               ...          ...             ...        ...
      1594       0.141593          0.328767         0.08        0.075342   0.130217
      1595       0.115044          0.294521         0.10        0.089041   0.083472
      1596       0.150442          0.267123         0.13        0.095890   0.106845
      1597       0.115044          0.359589         0.12        0.075342   0.105175
      1598       0.123894          0.130137         0.47        0.184932   0.091820

           free sulfur dioxide  total sulfur dioxide   density        pH  \
      0                0.140845              0.098940  0.567548  0.606299
      1                0.338028              0.215548  0.494126  0.362205
      2                0.197183              0.169611  0.508811  0.409449
      3                0.225352              0.190813  0.582232  0.330709
      4                0.140845              0.098940  0.567548  0.606299
```

```
       ...                  ...                 ...       ...       ...
1594               0.436620         0.134276  0.354626  0.559055
1595               0.535211         0.159011  0.370778  0.614173
1596               0.394366         0.120141  0.416300  0.535433
1597               0.436620         0.134276  0.396476  0.653543
1598               0.239437         0.127208  0.397944  0.511811

      sulphates   alcohol   quality
0      0.137725  0.153846         0
1      0.209581  0.215385         0
2      0.191617  0.215385         0
3      0.149701  0.215385         1
4      0.137725  0.153846         0
...         ...       ...       ...
1594   0.149701  0.323077         0
1595   0.257485  0.430769         1
1596   0.251497  0.400000         1
1597   0.227545  0.276923         0
1598   0.197605  0.400000         1

[1599 rows x 12 columns]
```

```python
[96]: #Quick Sanity check that our dataset it indeed relatively balanced
      label_converted_dataset['quality'].mean()
```

```
[96]: 0.5347091932457786
```

```python
[97]: y_wine = label_converted_dataset.loc[:,'quality']
      X_wine = label_converted_dataset.drop(target,axis=1)
```

```python
[98]: X_wine_np= np.asarray(X_wine)
      y_wine_np= np.asarray(y_wine)
      X_wine_train, X_wine_test, y_wine_train, y_wine_test =␣
       →train_test_split(X_wine_np, y_wine_np, test_size=0.25, random_state=1)
```

Here, you compare your Neural Network's performance (Training and Test Accuracy) with that of Scikit-learn's built-in model MLPClassifier. We, of course, do not expect you to exceed their performance, but your accuracies should be reasonably close to MLPClassifier's.

For the Red Wine Dataset, you should be getting over 70% for both your training as well as test set accuracies. If your MLP's training accuracy hovers around 50%, that is a sign that your model is not learning and you need to go back and fix a bug in your implementation.

```python
[99]: wine_quality_classifier = MLPClassifier(solver='sgd',
                                              alpha=0.01,
                                              learning_rate_init=0.001,
                                              batch_size=16,
                                              hidden_layer_sizes=(12,8,2),
```

```
                                            random_state=1,
                                            max_iter=200
                                            )

        wine_quality_classifier.fit(X_wine_train, y_wine_train)
```

[99]: MLPClassifier(alpha=0.01, batch_size=16, hidden_layer_sizes=(12, 8, 2),
                    random_state=1, solver='sgd')

[100]: ```
       wine_quality_classifier.score(X_wine_test, y_wine_test)
       ```

[100]: 0.75

## 3.2 Evaluating your MLP on the Red Wine Dataset

First, we instantiate and train a standard MLP (that uses the RELU activation function).

[101]: ```
       my_wine_NN_1 = NN()
       num_epochs= 150
       lambda_reg= 0.01
       learning_rate= 0.001
       batch_size= 32
       ```

[102]: ```
       my_wine_NN_1.add_layer('Hidden', 11, 16) #Note that the first layer's weight
        →matrix must be 11 x k , as the input feature vector is 11-dimensional.
       my_wine_NN_1.add_layer('Hidden', 16, 12)
       my_wine_NN_1.add_layer('Hidden', 12, 8)
       my_wine_NN_1.add_layer('Output', 8, 2)
       my_wine_NN_1.add_layer('Loss', 0, 0)
       ```

[103]: ```
       loss_wine_li_1= my_wine_NN_1.train(X_wine_train, y_wine_train, num_epochs,
        →batch_size, learning_rate, lambda_reg)
       plot_loss(loss_wine_li_1)
       ```
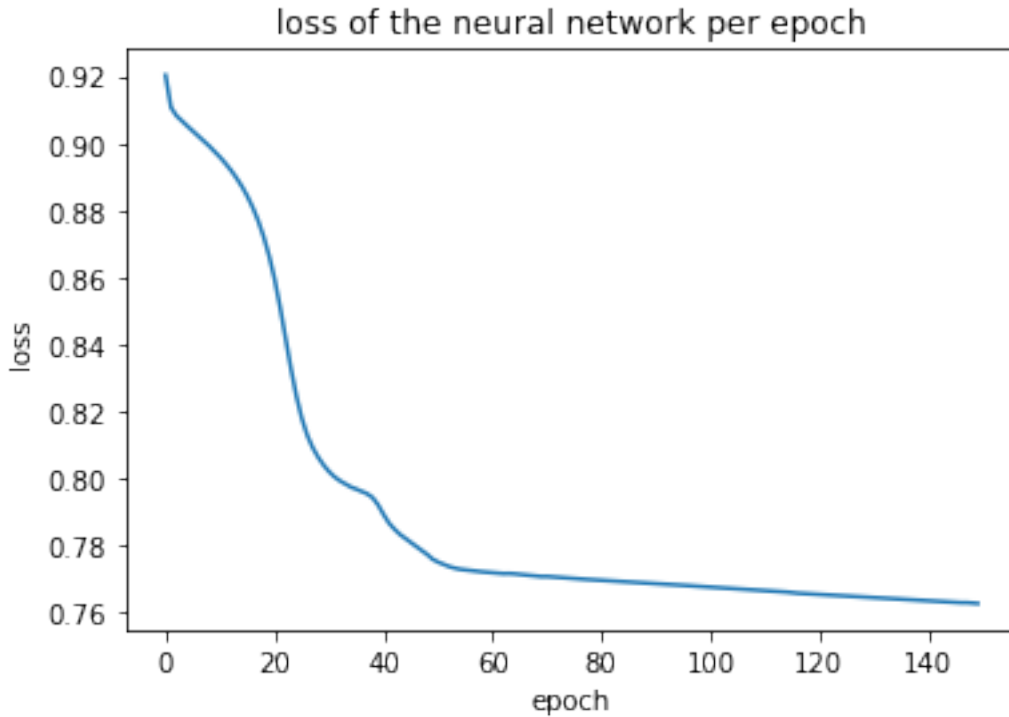
```
the accuracy on the training data after epoch 1 is 0.46371976647206004
the accuracy on the training data after epoch 2 is 0.5004170141784821
the accuracy on the training data after epoch 3 is 0.5195996663886572
the accuracy on the training data after epoch 4 is 0.5271059216013344
the accuracy on the training data after epoch 5 is 0.5321100917431193
the accuracy on the training data after epoch 6 is 0.5437864887406172
the accuracy on the training data after epoch 7 is 0.5504587155963303
the accuracy on the training data after epoch 8 is 0.5521267723102585
the accuracy on the training data after epoch 9 is 0.5512927439532944
the accuracy on the training data after epoch 10 is 0.5554628857381151
the accuracy on the training data after epoch 11 is 0.5521267723102585
the accuracy on the training data after epoch 12 is 0.5537948290241869
the accuracy on the training data after epoch 13 is 0.5587989991659716
```

```
the accuracy on the training data after epoch 14 is 0.5629691409507923
the accuracy on the training data after epoch 15 is 0.567139282735613
the accuracy on the training data after epoch 16 is 0.5704753961634695
the accuracy on the training data after epoch 17 is 0.5796497080900751
the accuracy on the training data after epoch 18 is 0.5829858215179317
the accuracy on the training data after epoch 19 is 0.6030025020850709
the accuracy on the training data after epoch 20 is 0.6296914095079232
the accuracy on the training data after epoch 21 is 0.6547122602168474
the accuracy on the training data after epoch 22 is 0.6638865721434529
the accuracy on the training data after epoch 23 is 0.6922435362802335
the accuracy on the training data after epoch 24 is 0.6989157631359466
the accuracy on the training data after epoch 25 is 0.7122602168473728
the accuracy on the training data after epoch 26 is 0.7206005004170142
the accuracy on the training data after epoch 27 is 0.7214345287739783
the accuracy on the training data after epoch 28 is 0.7222685571309424
the accuracy on the training data after epoch 29 is 0.7297748123436196
the accuracy on the training data after epoch 30 is 0.7306088407005839
the accuracy on the training data after epoch 31 is 0.7322768974145121
the accuracy on the training data after epoch 32 is 0.7381150959132611
the accuracy on the training data after epoch 33 is 0.737281067556297
the accuracy on the training data after epoch 34 is 0.7364470391993327
the accuracy on the training data after epoch 35 is 0.7347789824854045
the accuracy on the training data after epoch 36 is 0.7356130108423686
the accuracy on the training data after epoch 37 is 0.7381150959132611
the accuracy on the training data after epoch 38 is 0.7397831526271893
the accuracy on the training data after epoch 39 is 0.7397831526271893
the accuracy on the training data after epoch 40 is 0.7397831526271893
the accuracy on the training data after epoch 41 is 0.7381150959132611
the accuracy on the training data after epoch 42 is 0.7397831526271893
the accuracy on the training data after epoch 43 is 0.7364470391993327
the accuracy on the training data after epoch 44 is 0.7364470391993327
the accuracy on the training data after epoch 45 is 0.7347789824854045
the accuracy on the training data after epoch 46 is 0.737281067556297
the accuracy on the training data after epoch 47 is 0.737281067556297
the accuracy on the training data after epoch 48 is 0.737281067556297
the accuracy on the training data after epoch 49 is 0.7414512093411176
the accuracy on the training data after epoch 50 is 0.7414512093411176
the accuracy on the training data after epoch 51 is 0.74395329441201
the accuracy on the training data after epoch 52 is 0.7456213511259383
the accuracy on the training data after epoch 53 is 0.7456213511259383
the accuracy on the training data after epoch 54 is 0.74395329441201
the accuracy on the training data after epoch 55 is 0.7456213511259383
the accuracy on the training data after epoch 56 is 0.7472894078398665
the accuracy on the training data after epoch 57 is 0.7481234361968306
the accuracy on the training data after epoch 58 is 0.7481234361968306
the accuracy on the training data after epoch 59 is 0.7472894078398665
the accuracy on the training data after epoch 60 is 0.7472894078398665
the accuracy on the training data after epoch 61 is 0.7481234361968306
```

```
the accuracy on the training data after epoch 62 is 0.74812343619068306
the accuracy on the training data after epoch 63 is 0.74812343619068306
the accuracy on the training data after epoch 64 is 0.7472894078398665
the accuracy on the training data after epoch 65 is 0.74812343619068306
the accuracy on the training data after epoch 66 is 0.74812343619068306
the accuracy on the training data after epoch 67 is 0.74812343619068306
the accuracy on the training data after epoch 68 is 0.74812343619068306
the accuracy on the training data after epoch 69 is 0.74812343619068306
the accuracy on the training data after epoch 70 is 0.7472894078398665
the accuracy on the training data after epoch 71 is 0.7472894078398665
the accuracy on the training data after epoch 72 is 0.7472894078398665
the accuracy on the training data after epoch 73 is 0.7472894078398665
the accuracy on the training data after epoch 74 is 0.7472894078398665
the accuracy on the training data after epoch 75 is 0.7464553794829024
the accuracy on the training data after epoch 76 is 0.7464553794829024
the accuracy on the training data after epoch 77 is 0.7456213511259383
the accuracy on the training data after epoch 78 is 0.7456213511259383
the accuracy on the training data after epoch 79 is 0.7456213511259383
the accuracy on the training data after epoch 80 is 0.7456213511259383
the accuracy on the training data after epoch 81 is 0.7456213511259383
the accuracy on the training data after epoch 82 is 0.7447873227689742
the accuracy on the training data after epoch 83 is 0.7447873227689742
the accuracy on the training data after epoch 84 is 0.7447873227689742
the accuracy on the training data after epoch 85 is 0.7447873227689742
the accuracy on the training data after epoch 86 is 0.7447873227689742
the accuracy on the training data after epoch 87 is 0.7447873227689742
the accuracy on the training data after epoch 88 is 0.7456213511259383
the accuracy on the training data after epoch 89 is 0.7456213511259383
the accuracy on the training data after epoch 90 is 0.7456213511259383
the accuracy on the training data after epoch 91 is 0.7464553794829024
the accuracy on the training data after epoch 92 is 0.7472894078398665
the accuracy on the training data after epoch 93 is 0.7472894078398665
the accuracy on the training data after epoch 94 is 0.74812343619068306
the accuracy on the training data after epoch 95 is 0.74812343619068306
the accuracy on the training data after epoch 96 is 0.74812343619068306
the accuracy on the training data after epoch 97 is 0.74812343619068306
the accuracy on the training data after epoch 98 is 0.74812343619068306
the accuracy on the training data after epoch 99 is 0.74812343619068306
the accuracy on the training data after epoch 100 is 0.7464553794829024
the accuracy on the training data after epoch 101 is 0.7464553794829024
the accuracy on the training data after epoch 102 is 0.7464553794829024
the accuracy on the training data after epoch 103 is 0.7464553794829024
the accuracy on the training data after epoch 104 is 0.7464553794829024
the accuracy on the training data after epoch 105 is 0.7464553794829024
the accuracy on the training data after epoch 106 is 0.7464553794829024
the accuracy on the training data after epoch 107 is 0.7464553794829024
the accuracy on the training data after epoch 108 is 0.7464553794829024
the accuracy on the training data after epoch 109 is 0.7456213511259383
```

```
the accuracy on the training data after epoch 110 is 0.7456213511259383
the accuracy on the training data after epoch 111 is 0.7456213511259383
the accuracy on the training data after epoch 112 is 0.7456213511259383
the accuracy on the training data after epoch 113 is 0.7456213511259383
the accuracy on the training data after epoch 114 is 0.7456213511259383
the accuracy on the training data after epoch 115 is 0.7464553794829024
the accuracy on the training data after epoch 116 is 0.7464553794829024
the accuracy on the training data after epoch 117 is 0.7456213511259383
the accuracy on the training data after epoch 118 is 0.7456213511259383
the accuracy on the training data after epoch 119 is 0.7456213511259383
the accuracy on the training data after epoch 120 is 0.7456213511259383
the accuracy on the training data after epoch 121 is 0.7456213511259383
the accuracy on the training data after epoch 122 is 0.7456213511259383
the accuracy on the training data after epoch 123 is 0.7464553794829024
the accuracy on the training data after epoch 124 is 0.7464553794829024
the accuracy on the training data after epoch 125 is 0.7464553794829024
the accuracy on the training data after epoch 126 is 0.7464553794829024
the accuracy on the training data after epoch 127 is 0.7464553794829024
the accuracy on the training data after epoch 128 is 0.7464553794829024
the accuracy on the training data after epoch 129 is 0.7464553794829024
the accuracy on the training data after epoch 130 is 0.7464553794829024
the accuracy on the training data after epoch 131 is 0.7464553794829024
the accuracy on the training data after epoch 132 is 0.7464553794829024
the accuracy on the training data after epoch 133 is 0.7464553794829024
the accuracy on the training data after epoch 134 is 0.7464553794829024
the accuracy on the training data after epoch 135 is 0.7464553794829024
the accuracy on the training data after epoch 136 is 0.7464553794829024
the accuracy on the training data after epoch 137 is 0.7464553794829024
the accuracy on the training data after epoch 138 is 0.7464553794829024
the accuracy on the training data after epoch 139 is 0.7464553794829024
the accuracy on the training data after epoch 140 is 0.7464553794829024
the accuracy on the training data after epoch 141 is 0.7456213511259383
the accuracy on the training data after epoch 142 is 0.7456213511259383
the accuracy on the training data after epoch 143 is 0.7456213511259383
the accuracy on the training data after epoch 144 is 0.7456213511259383
the accuracy on the training data after epoch 145 is 0.7464553794829024
the accuracy on the training data after epoch 146 is 0.7464553794829024
the accuracy on the training data after epoch 147 is 0.7464553794829024
the accuracy on the training data after epoch 148 is 0.7464553794829024
the accuracy on the training data after epoch 149 is 0.7456213511259383
the accuracy on the training data after epoch 150 is 0.7456213511259383
```

loss of the neural network per epoch

```
[104]: print(my_wine_NN_1.compute_accuracy(X_wine_test, y_wine_test))
```

```
0.74
```

```
[105]: my_wine_NN_2 = NN()
       num_epochs= 100
       lambda_reg= 0.01
       learning_rate= 0.001
       batch_size= 32
```

```
[106]: my_wine_NN_2.add_layer('Hidden', 11, 16) #Note that the first layer's weight␣
       ↪matrix must be 11 x k , as the input feature vector is 11-dimensional.
       my_wine_NN_2.add_layer('Hidden', 16, 12)
       my_wine_NN_2.add_layer('Hidden_Vondrick', 12, 8)
       my_wine_NN_2.add_layer('Output', 8, 2)
       my_wine_NN_2.add_layer('Loss', 2, 2)
```

```
Intital Value of Vondrick Exponent: 1.5879557254216647
```

```
[107]: loss_wine_li_2= my_wine_NN_2.train(X_wine_train, y_wine_train, num_epochs,␣
       ↪batch_size, learning_rate, lambda_reg)
       plot_loss(loss_wine_li_2)
```

```
the accuracy on the training data after epoch 1 is 0.45454545454545453
```

17

```
the accuracy on the training data after epoch 2 is 0.5404503753127606
the accuracy on the training data after epoch 3 is 0.5446205170975813
the accuracy on the training data after epoch 4 is 0.5479566305254379
the accuracy on the training data after epoch 5 is 0.5521267723102585
the accuracy on the training data after epoch 6 is 0.5537948290241869
the accuracy on the training data after epoch 7 is 0.5613010842368641
the accuracy on the training data after epoch 8 is 0.5863219349457881
the accuracy on the training data after epoch 9 is 0.6063386155129274
the accuracy on the training data after epoch 10 is 0.6296914095079232
the accuracy on the training data after epoch 11 is 0.6422018348623854
the accuracy on the training data after epoch 12 is 0.6580483736447039
the accuracy on the training data after epoch 13 is 0.6730608840700584
the accuracy on the training data after epoch 14 is 0.6914095079232694
the accuracy on the training data after epoch 15 is 0.69557964970809
the accuracy on the training data after epoch 16 is 0.7172643869891576
the accuracy on the training data after epoch 17 is 0.7180984153461217
the accuracy on the training data after epoch 18 is 0.7214345287739783
the accuracy on the training data after epoch 19 is 0.7206005004170142
the accuracy on the training data after epoch 20 is 0.7222685571309424
the accuracy on the training data after epoch 21 is 0.7247706422018348
the accuracy on the training data after epoch 22 is 0.7222685571309424
the accuracy on the training data after epoch 23 is 0.7222685571309424
the accuracy on the training data after epoch 24 is 0.7222685571309424
the accuracy on the training data after epoch 25 is 0.7214345287739783
the accuracy on the training data after epoch 26 is 0.7289407839866555
the accuracy on the training data after epoch 27 is 0.7306088407005839
the accuracy on the training data after epoch 28 is 0.7289407839866555
the accuracy on the training data after epoch 29 is 0.7272727272727273
the accuracy on the training data after epoch 30 is 0.7289407839866555
the accuracy on the training data after epoch 31 is 0.7297748123436196
the accuracy on the training data after epoch 32 is 0.7297748123436196
the accuracy on the training data after epoch 33 is 0.7297748123436196
the accuracy on the training data after epoch 34 is 0.7306088407005839
the accuracy on the training data after epoch 35 is 0.731442869057548
the accuracy on the training data after epoch 36 is 0.7322768974145121
the accuracy on the training data after epoch 37 is 0.731442869057548
the accuracy on the training data after epoch 38 is 0.7322768974145121
the accuracy on the training data after epoch 39 is 0.731442869057548
the accuracy on the training data after epoch 40 is 0.7339449541284404
the accuracy on the training data after epoch 41 is 0.7356130108423686
the accuracy on the training data after epoch 42 is 0.7356130108423686
the accuracy on the training data after epoch 43 is 0.7381150959132611
the accuracy on the training data after epoch 44 is 0.7397831526271893
the accuracy on the training data after epoch 45 is 0.7414512093411176
the accuracy on the training data after epoch 46 is 0.7414512093411176
the accuracy on the training data after epoch 47 is 0.7431192660550459
the accuracy on the training data after epoch 48 is 0.7431192660550459
the accuracy on the training data after epoch 49 is 0.7431192660550459
```

```
the accuracy on the training data after epoch 50 is 0.7447873227689742
the accuracy on the training data after epoch 51 is 0.74395329441201
the accuracy on the training data after epoch 52 is 0.74395329441201
the accuracy on the training data after epoch 53 is 0.7447873227689742
the accuracy on the training data after epoch 54 is 0.7464553794829024
the accuracy on the training data after epoch 55 is 0.7472894078398665
the accuracy on the training data after epoch 56 is 0.7481234361968306
the accuracy on the training data after epoch 57 is 0.7472894078398665
the accuracy on the training data after epoch 58 is 0.7489574645537949
the accuracy on the training data after epoch 59 is 0.7489574645537949
the accuracy on the training data after epoch 60 is 0.7481234361968306
the accuracy on the training data after epoch 61 is 0.7489574645537949
the accuracy on the training data after epoch 62 is 0.7481234361968306
the accuracy on the training data after epoch 63 is 0.7472894078398665
the accuracy on the training data after epoch 64 is 0.7472894078398665
the accuracy on the training data after epoch 65 is 0.7472894078398665
the accuracy on the training data after epoch 66 is 0.7489574645537949
the accuracy on the training data after epoch 67 is 0.7506255212677231
the accuracy on the training data after epoch 68 is 0.749791492910759
the accuracy on the training data after epoch 69 is 0.7489574645537949
the accuracy on the training data after epoch 70 is 0.7489574645537949
the accuracy on the training data after epoch 71 is 0.7489574645537949
the accuracy on the training data after epoch 72 is 0.7489574645537949
the accuracy on the training data after epoch 73 is 0.7489574645537949
the accuracy on the training data after epoch 74 is 0.7489574645537949
the accuracy on the training data after epoch 75 is 0.7489574645537949
the accuracy on the training data after epoch 76 is 0.7489574645537949
the accuracy on the training data after epoch 77 is 0.7489574645537949
the accuracy on the training data after epoch 78 is 0.749791492910759
the accuracy on the training data after epoch 79 is 0.7506255212677231
the accuracy on the training data after epoch 80 is 0.7506255212677231
the accuracy on the training data after epoch 81 is 0.7514595496246872
the accuracy on the training data after epoch 82 is 0.7514595496246872
the accuracy on the training data after epoch 83 is 0.7514595496246872
the accuracy on the training data after epoch 84 is 0.7514595496246872
the accuracy on the training data after epoch 85 is 0.7514595496246872
the accuracy on the training data after epoch 86 is 0.7514595496246872
the accuracy on the training data after epoch 87 is 0.7514595496246872
the accuracy on the training data after epoch 88 is 0.7506255212677231
the accuracy on the training data after epoch 89 is 0.7514595496246872
the accuracy on the training data after epoch 90 is 0.7506255212677231
the accuracy on the training data after epoch 91 is 0.7531276063386155
the accuracy on the training data after epoch 92 is 0.7531276063386155
the accuracy on the training data after epoch 93 is 0.7531276063386155
the accuracy on the training data after epoch 94 is 0.7531276063386155
the accuracy on the training data after epoch 95 is 0.7522935779816514
the accuracy on the training data after epoch 96 is 0.7522935779816514
the accuracy on the training data after epoch 97 is 0.7514595496246872
```

```
the accuracy on the training data after epoch 98 is 0.7522935779816514
the accuracy on the training data after epoch 99 is 0.7522935779816514
the accuracy on the training data after epoch 100 is 0.7514595496246872
```

loss of the neural network per epoch
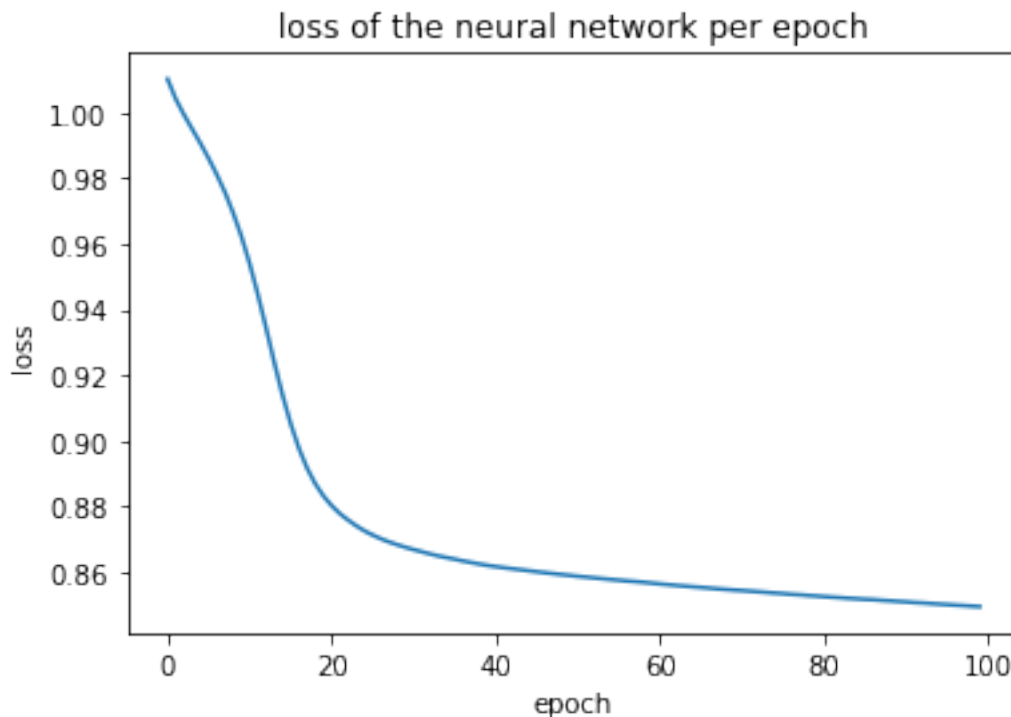
[108]: `print(my_wine_NN_2.compute_accuracy(X_wine_test, y_wine_test))`

```
0.7675
```

3.3 Section 2: Optimization

You now have intuition for how the backpropagation procedure updates every single node or layer in the neural network with the gradient of the loss function with respect to the specific parameters. Luckily, you will not have to repeat this tedious enumeration in the future as autograd packages can help you track and organize the gradient tracking process. Even better, most modern neural network libraries like PyTorch and TensorFlow have their own autograd versions which abstract gradient calculations into a single function call on your loss function, instantly tracking along a computational neural network graph to quickly update gradients.

Equally important to the machine learning pipeline is the process of optimization: actually using the calculated **gradient** at the current values and moving to the next values, which are closer to the optimal arguments to our function. A version of **stochastic gradient descent** was already used in section 1 to train the multilayer perceptron.

In a perfect world, once we have an analytical formulation for the gradient of a function, we can go back to the classic technique from Calc 2/Calc 3 of setting the gradient to 0 and calculating the

values of the variables at the optimal location. Indeed take

$$f(x, y) = x^2 + y^2$$

We have that:

$$\nabla f(x, y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

yielding the location of the optima at

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

However, this technique does not work well with most functions: it is the exception rather than norm to set values to 0 to get and classify the optima. Moreover as the order of the partial derivatives increases, the resulting polynomials become harder and harder to solve (even if a solution is possible) and the problem quickly becomes computationally intractable. Second order methods (Hessian matrices) are required to further classify if these points are in any way useful (saddle points for example would be severely detrimental stopping points for our optimization problem) and are notoriously difficult computationally.

We will now proceed to look at some well-known gradient-based iterative algorithms that have successfully been deployed in training deep learning models. In a typical machine learning pipeline, these optimizers will only be useful after the backpropagation stage is complete. At this stage we have that the network parameters θ have an associated gradient with respect to the loss $\frac{\delta \mathcal{L}}{\delta \theta}$. This step involves using the gradient to nudge θ towards an optimum value, i.e, the θ that would yield the lowest possible loss.

In reality, loss functions that are encountered in neural networks are parametrized by hundreds, thousands and millions of parameters, and hence it is not always easy to visualize or study the exact properties of optimization algorithms on them. Typically such optimization functions are designed for **convex** functions, which are characterized, amidst others by **Jensen's inequality**, meaning that the function always lies below any surface connecting two points on this surface, or precisely for a function $f : X \mapsto Y$:

$$f(\theta \vec{x} + (1 - \theta)\vec{y}) \leq \theta \vec{x} + (1 - \theta)\vec{y}$$

for $x, y \in X$ and $0 \leq \theta \leq 1$. In $\mathbb{R}^3$ the **bowl** or **sphere** function is the archetypical convex function.

$$f(x, y) = x^2 + y^2$$

While convex theory gives convenient bounds on gradient based optimization, it is not enough to stop here as we do not expect the loss function for our machine learning models to be convex. Optimization research then focuses on studying the behavior of algorithms on test functions that accentuate some of the possible problematic optimization scenarios we might run into on a loss manifold in higher dimensions. For example, one potential issue we have studied in class is that in

an iterative optimization process, we might get stuck in a local optima. A potential test function that could especially be indicative of if an optimization algorithm handles this issue, is the following function, which we will call the **mult** function:

$$f(x, y) = \sin(\sqrt{(x^2 + y^2)})$$

Another issue could be a point which has different signs to its curvature in different directions, but locally has no gradient, aka a saddle point. A "test" function to effectively evaluate an algorithm's performance on saddle points could be

$$f(x, y) = x^3 + 3xy^2$$

or otherwise known as the **monkey saddle**.

As a final example, it might be concerning if a point has a very high gradient in one direction but extremely low gradient aka a high condition number is associated with its eigendecomposition of the Hessian (if you didn't understand this last line, that's fine- the Hessian only comes into play while giving a proof of convergence for convex functions for gradient descent and such theory is beyond the scope of this class. An alternate way of thinking why this is an issue is because it will cause a zig-zag convergence to the optima when we can potentially save a lot of iterations by just taking a step along one axis). A function to test convergence performance emperically for this issue could be one shaped like a taco shell. A well known function of this kind is the **Matyas** function.

$$f(x, y) = 0.26(x^2 + y^2) - 0.48xy$$

Bonus point for figuring out who the **Matyas** function is named after, because I looked forever in the hopes of adding a half-clever note on who Matyas was to improve the readability of this homework with a casual fun fact that has nothing to do with machine learning, but I ended up getting lost online and achieving nothing for 30 mins.

Implement the said functions below as bowl, mult, monkey and matyas and use the plot function below to visualize what they look like. Add a comment on each to explain what there utility might be as test functions.

```
[290]:  from mpl_toolkits.mplot3d import Axes3D
        from matplotlib.colors import LogNorm
        import matplotlib.colors
        from matplotlib import animation, rc
        from IPython.display import HTML
        import matplotlib.pyplot as plt
        import autograd.numpy as np
        from autograd import grad
        from autograd import elementwise_grad as egrad
        from scipy.optimize import minimize
        from collections import defaultdict
        from itertools import zip_longest
        from functools import partial
        import numpy
```

```python
def bowl():
    '''
    TODO

    Implement the bowl function as defined above.

    Add comment here explaining why it is a reasonable test function.

    Answer:
    It is a convex function that we can easily find its optimal point even␣
↪without the SGD method. We can just solve the derivative setting to zero and␣
↪find the optimal point. SGD methods can also easily find the optimal point␣
↪of this function.
    '''

    def f_bowl(x, y):
        out =  x**2 + y**2# CODE HERE

        return out

    def opt_bowl():
        return np.array([0., 0.])

    return f_bowl, opt_bowl


def mult():
    '''
    TODO

    Implement the mult function as defined above.

    Add comment here explaining why it is a reasonable test function.

    Answer:
    This function has a local minimum point at the center of the axis. The SGD␣
↪method could find this local minimum point and get stuck here. But this is␣
↪not the point we want as it is not the global minimum point.
    '''

    def f_mult(x, y):

        out = np.sin((x ** 2 + y ** 2) ** 0.5) # CODE HERE

        return out
```

```python
    def opt_mult():
        return np.array([0., 0.])

    return f_mult, opt_mult


def monkey():
    '''
    TODO

    Implement the monkey saddle function as defined above.

    Add comment here explaining why it is a reasonable test function.

    Answer:
    There is a saddle point at the center of the axis. The two directions along␣
↪two axis at the saddle point are different sign(positive and negtive), and␣
↪the gradient at the saddle point is zero. So the SGD methods could also get␣
↪stuck at this saddle point.
    '''

    def f_monkey(x, y):
        out = x**3 + 3*x*y**2 # CODE HERE

        return out

    def opt_monkey():
        return np.array([0., 0.])

    return f_monkey, opt_monkey


def matyas():
    '''
    TODO

    Implement the mult function as defined above.

    Add comment here explaining why it is a reasonable test function.

    Answer:
    The gradient along one direction is changing very quickly while the␣
↪gradient is changing very slowly along the perpendicular direction. So the␣
↪SGD could have a lot of movements along the direction whose gradient is␣
↪changing quickly but move slowly toward the direction with smooth gradient.
    '''
```

```python
    def f_matyas(x, y):
        out = 0.26 * (x ** 2 + y ** 2) - 0.48 * x * y # CODE HERE

        return out

    def opt_matyas():
        return np.array([0., 0.])

    return f_matyas, opt_matyas


def plot_func(f):
    func, opt = f()

    # Set grid parameters
    xmin = -4.5
    xmax = 4.5
    ymin = -4.5
    ymax = 4.5
    step = 0.2

    x, y = np.meshgrid(np.arange(xmin, xmax + step, step), np.arange(ymin, ymax␣
 ↪+ step, step))
    z = func(x, y)
    cp = opt()
    optima = cp.reshape(-1, 1)

    fig = plt.figure(figsize=(12, 6), dpi=100)
    ax = fig.add_subplot(1, 2, 1, projection='3d')
    ax.plot_surface(x, y, z, rstride=5, cstride=5, alpha=0.5, cmap=plt.cm.
 ↪plasma)
    cset = ax.contourf(x, y, z, 25, zdir='z', offset=-1, alpha=0.6, cmap=plt.cm.
 ↪coolwarm)
    out1 = func(*optima)

    ax.plot(*optima, out1, 'r*', markersize=10)

    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$z$')
```

```python
[291]: plot_func(monkey)
       plot_func(mult)
       plot_func(matyas)
       plot_func(bowl)
```
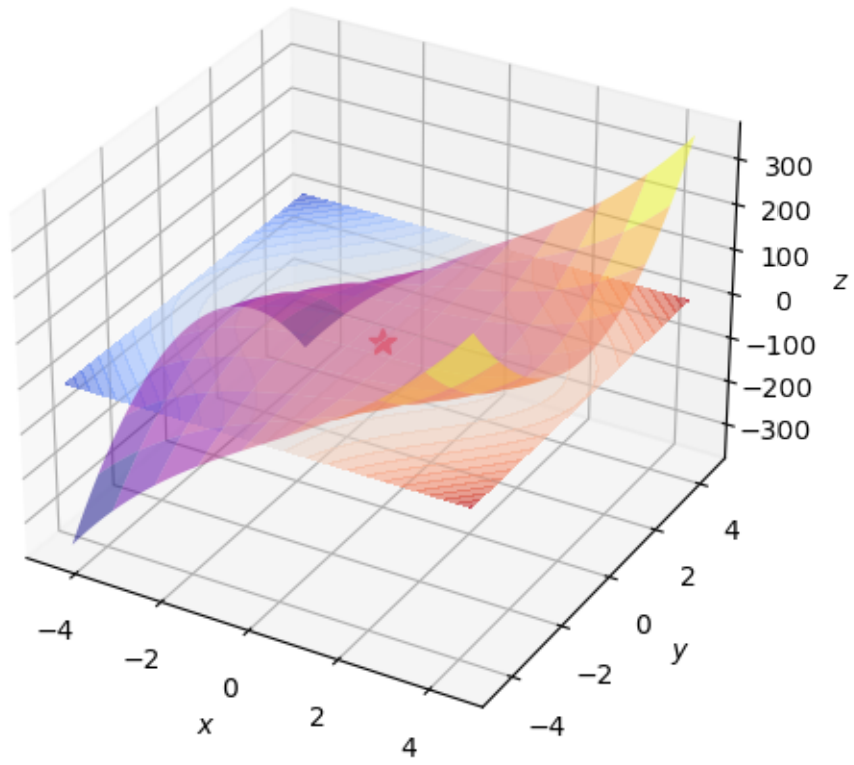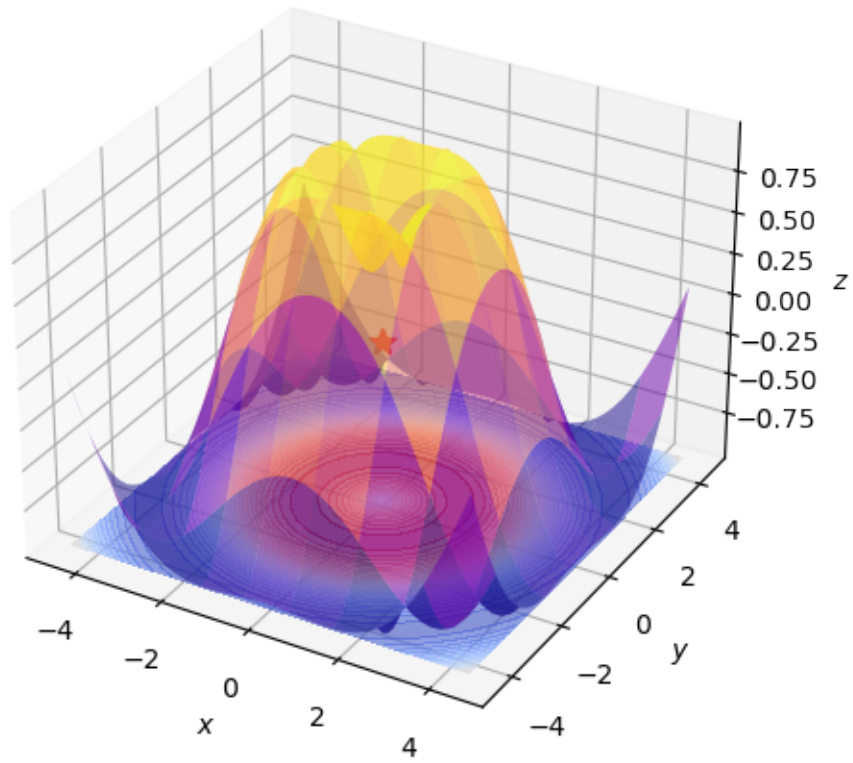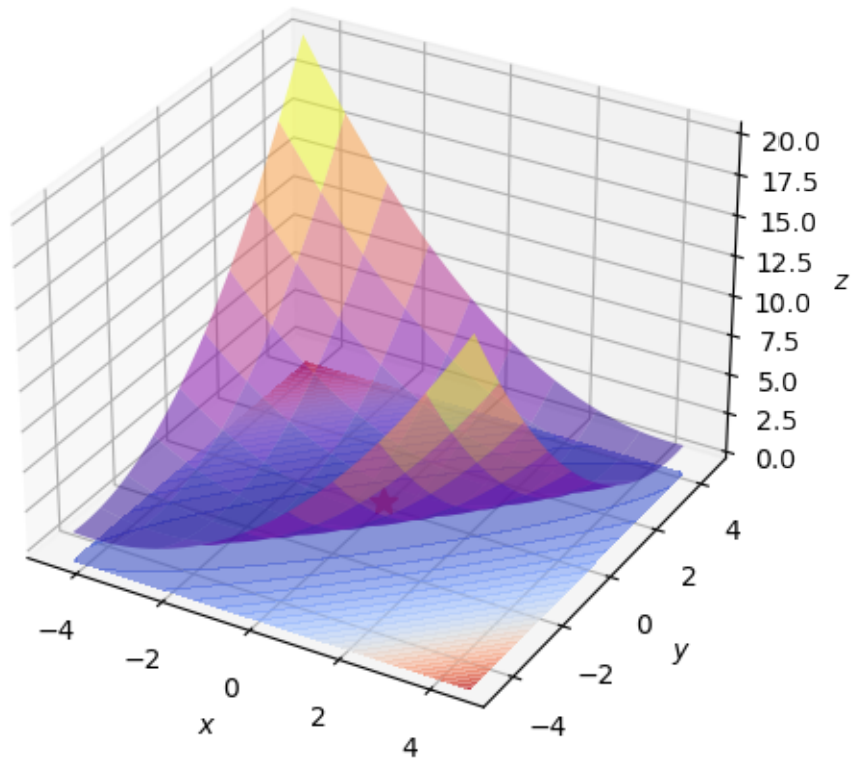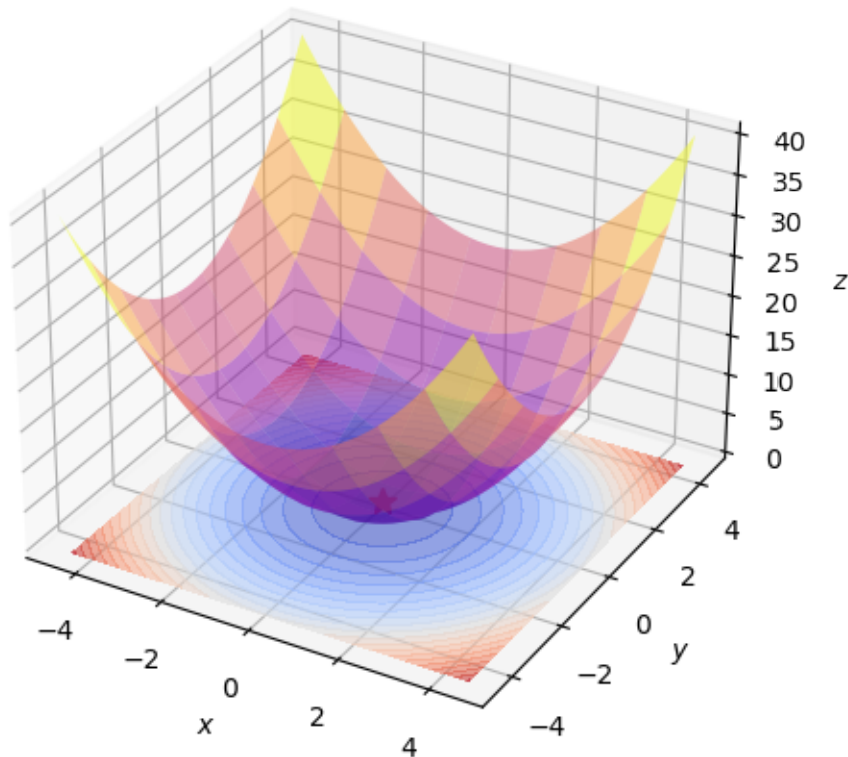
3.4 Gradient Descent

$$\theta = \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$$

We computed the gradient with respect to each of the parameters and make an update in the opposite direction of the local gradient. α which is the learning rate, is a hyperparam that controls how quickly gradient descent converges. If it is too low, too many updates may be required, especially as gradients are small, and if it is too large you may overshoot the optima. In the context of batch gradient descent- at each epoch where the above update is ran- batch gradient descent requires that the entire loss function be computed at each stage through a pass through the entire training data so that the calculation of the $\mathcal{L}(\theta)$ and its subsequent gradient be accurate. As a result, this method is usually very slow. Additionally, it is only guaranteed to converge to the global minimum for convex functions such as the bowl functions and this is often not the case in machine learning, in which case it is guaranteed to only converge to local minima.

Implement a version of gradient descent below.

```
[310]: def gradient_descent(x, y, dx, dy, hparams):
    '''
    x: value of x before update
    y: value of y before update
```

```
    dx: derivative wrt x
    dy: derivative wrt y
    hparams must contain alpha


    TODO


    Implement the update rule and return the new value of x, y after the update
    '''


    alpha = hparams['alpha']

    #CODE HERE
    x = x - alpha*dx
    y = y - alpha*dy
    return x, y
```

3.5   Stochastic Gradient Descent

$$\theta = \theta - \alpha \nabla_\theta \mathcal{L}(\theta; x^i, y^i)$$

This method is very similar to full gradient descent, except that instead of calculating the loss function over all training examples, the loss function is only calculated over a single training data point at a time. As such the estimate of $\mathcal{L}(\theta)$ and its gradient is not precise, but in return, SGD is much faster, and additionally reduces redundant time for recomputing gradients for similar examples.

Gradients received can be highly erratic because they are not calculated over the full dataset, and as a result the optimization path will often zig zag and occasionally spiral out of control. It has been shown that with enough control over the learning rate, sgd and batch gradient descent often achieve the same results, but SGD does it much faster in the context of machine learning. A commonly used variant of gradient descent that sits between full and stochastic gradient descent is mini-batch gradient descent where the gradient and loss function are calculated over a randomly chosen fixed size batch of training examples.

Implement a version of stochastic gradient descent below. Since we are not actually using a dataset or even a data distribution to generate our "loss" manifold that we are trying to optimize over, you should simulate the effect of approximation using gaussian noise on the gradients. Feel free to use the gauss function imported below for the same.

```
[311]: from random import gauss

       def stochastic_gradient_descent(x, y, dx, dy, hparams):
           '''
           x: value of x before update
           y: value of y before update
           dx: derivative wrt x
           dy: derivative wrt y
           hparams must contain alpha
```

```python
    # TODO

    Implement the update rule and return the new value of x, y after the update
    '''

    alpha = hparams['alpha']

    #CODE HERE
    dx = gauss(dx, 1)
    dy = gauss(dy, 1)
    x = x - alpha * dx
    y = y - alpha * dy
    return x, y
```

3.6 Stochastic Gradient Descent and Momentum

$$\nu_t = \gamma\nu_{t-1} + \alpha\nabla_\theta\mathcal{L}(\theta; x^i, y^i)$$

$$\theta = \theta - \nu_t$$

Stochastic Gradient Descent suffers very heavily if the loss function changes very quickly in one direction and slowly in another direction (eg. taco shell function) In this case the direction of the gradient does not align with the direction toward the minimum and a zig zag motion is direction with slower gradients. This is the case where the Hessian of the loss function wrt parameters has a high condition number, aka the eigenvalues of the representative matrix have a high ratio between the highest and the lowest eigenvalue.

The use of the word momentum is a metaphor in this context, as such a method allows the navigation of shallow local optima or the navigation of ravines using the build-up of gradients from the navigation. The idea is that we maintain what is a 'velocity' term ν at each timestep that keeps track of how much gradient has so far been encountered. Hence this value builds up in each direction and even in cases when gradients received in training are poor (eg. around saddle points, or local minima), the algorithm is able to escape such points (much like a ball rolling down a hill). The term for gamma, which is a hyperparameter, can be thought of as friction for the build-up of this velocity, as it decides how much of the previous velocity to count at a certain timestep. Even in the case of a ravine, the zig-zag motion of stochastic gradient descent would reduce as the buildup term in one direction would carry us smoothly through the low-gradient sensitive dimension. The momentum vectors also help cancel/smooth some of the noise that results from approximating gradients using a single data point, aka, the stochastic way.

This method has its own flaws: it might settle in extremely deep minima. Such minima are not desirable even as global minima, as on a data manifold they may be "too good" to be true; representative of a kind of overfit. An example test function for this issue is the **Easom function**. The good news is that with some tuning of **gamma** more often that never SGD + momentum will settle in shallow, wide minima.

```python
[312]: def easom():
           '''
```

```
    TODO

    Implement the easom function as defined above.

    Add comment here explaining why it is a reasonable test function.
    '''

    def f_easom(x, y):
        #CODE HERE
        out = -np.cos(x)*np.cos(y)*np.exp(-((x-np.pi)**2 + (y-np.pi)**2))
        return out

    def opt_easom():

        return np.array([np.pi, np.pi])


    return f_easom, opt_easom

plot_func(easom)
```
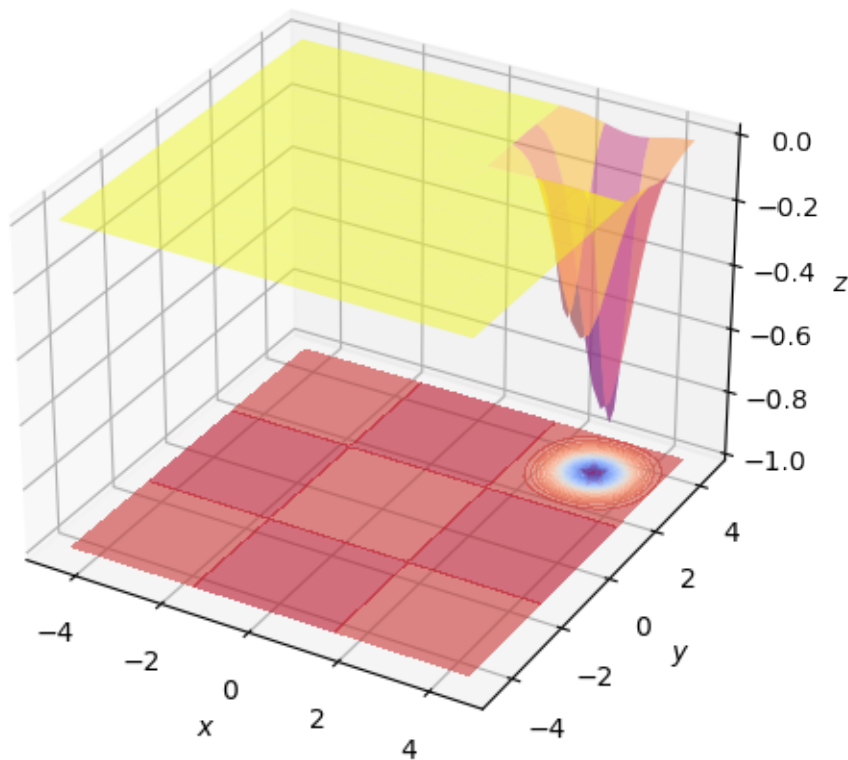
Implement your version of stochastic gradient descent with momentum below.

```python
[313]: def momentum(x, y, dx, dy, v_x, v_y, hparams):
           '''
           x: value of x before update
           y: value of y before update
           dx: derivative wrt x
           dy: derivative wrt y
           v_x: velocity parameter wrt x
           v_y: velocity parameter wrt y
           hparams must contain alpha and gamma

           # TODO

           Implement the update rule and return the new value of x, y, v_x, v_y after
           the update. Don't forget to add the gaussian noise for the stochasticity
           '''

           alpha = hparams['alpha']
           gamma = hparams['gamma']

           #CODE HERE
           v_x = gamma*v_x + alpha*gauss(dx, 1)
           v_y = gamma*v_y + alpha*gauss(dy, 1)
           x = x - v_x
           y = y - v_y
           return x, y, v_x, v_y
```

3.7   Nesterov Momentum

$$\nu_t = \gamma\nu_{t-1} + \alpha\nabla_\theta\mathcal{L}(\theta - \gamma\nu_{t-1})$$

$$\theta = \theta - \nu_t$$

A problem with momentum is that there is no way to control the slow-down of the optimizer even as we approach a minima- the optimizer isn't smart enough to decide whether or not it needs to continue up a slope once it has reached a (local or global) minima. A version of momentum, called Nesterov momentum helps deal with this problem by first changing parameters in the direction of the accumulated gradient, estimating the destination gradient and then making an update in that direction.

You do not have to implement nesterov updates.

3.8   AdaGrad

Adagrad is another approach toward solving some of the same problems that momentum attempts to solve. However, it does more by giving the optimizer the ability to adapt updates for individual parameters depending on their importance. It adapts its learning rate to make higher updates

(higher learning rate) for dimensions/features with higher values and lower updates (low learning rate) for dimensions/features with lower values.

$$G_{i,i}^{t+1} = G_{i,i}^t + (\frac{\delta \mathcal{L}(\theta)}{\delta \theta_i})^2$$

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{G_{i,i} + \epsilon}} \nabla_\theta \mathcal{L}(\theta_{t,i})$$

Here G starts off as a 0 matrix and builds up squared gradients for each feature/dimension which is later used to scale the value for the parameter updates. ϵ is generally a very small number to ensure that division by zero does not occur in case the initialization point provides no gradient. The biggest advantage of adagrad is that it takes away the need to manually tune the learning rate as is required with SGD, and provides feature-tuned learning rates. However, because of the way adagrad adjusts learning rates it sometimes tends to work well only with convex problems, as with non-convex problems in runs into issues with how to overshooting local minima/saddle points as it slows down rapidly due to accumulating squared gradients. It also provides really slow initial updates if the gradients at the initialization point are really high.

Implement your variant of Adagrad below.

```python
[314]:  def adagrad(x, y, dx, dy, v_x, v_y, hparams):
            '''
            x: value of x before update
            y: value of y before update
            dx: derivative wrt x
            dy: derivative wrt y
            v_x: velocity parameter wrt x
            v_y: velocity parameter wrt y
            hparams must contain eps and alpha

            # TODO

            Implement the update rule and return the new value of x, y, v_x, v_y after
            the update. Don't forget to add the gaussian noise for the stochasticity.
            '''

            eps = hparams['eps']
            alpha = hparams['alpha']

            #CODE HERE
            v_x = v_x + (dx)**2
            v_y = v_y + (dy)**2
            x = x - alpha*dx/((v_x + eps)**0.5)
            y = y - alpha*dy/((v_y + eps)**0.5)
            return x, y, v_x, v_y
```

RMSProp and **AdaDelta** are two different algorithms that combat the aggressive learning rate reduction that comes with AdaGrad.

3.9 Adaptive Momentum Estimation

This is the **best of all worlds** update algorithm. The second line (f) accumulated gradients much like momentum, while the third line (s) accumulates gradient squared for adjusting the learning rates like AdaGrad. These have to be adjusted in lines 3 and 4, because they are initialized to 0 at the beginning of optimization and since f is in the numerator of the final update we cannot afford to multiply by a really small number, neither can we afford to divide by a really small number. β_1 and β_2 must be less than 1 and are typically initialized high values such as 0.9 or 0.99. Yhe unbiasing operations on line 4 and 5 above help bring up the values of f and s early in training.

$$g_{t,i} = \nabla_\theta \mathcal{L}(\theta_{t,i})$$
$$f_{t,i} = \beta_1 f_{t-1,i} + (1 - \beta_1)g_{t,i}$$

$$s_{t,i} = \beta_2 s_{t-1,i} + (1 - \beta_2)g_{t,i}^2$$
$$\hat{f}_{t,i} = \frac{f_{t,i}}{1 - \beta_1^t}$$
$$\hat{s}_{t,i} = \frac{s_{t,i}}{1 - \beta_2^t}$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha \hat{f}_{t,i}}{\sqrt{\hat{s}_{t,i} + \epsilon}}$$

Implement your function for adam below. Follow the comments closely for interpretation of inputs and outputs.

```
[315]: def adam(x, y, dx, dy, f_x, f_y, s_x, s_y, i, hparams):
           '''
           x: value of x before update
           y: value of y before update
           dx: derivative wrt x
           dy: derivative wrt y
           f_x, f_y: first order gradient accumulators
           s_x, s_y: second order gradient accumulators
           i: number of iteration
           hparams must contain alpha, eps, beta_1 and beta_2

           # TODO

           Implement the update rule and return the new value of x, y, f_x, f_y, s_x,
           s_y, after the update. Don't forget to add the gaussian noise for
           the stochasticity.
           '''

           eps = hparams['eps']
           alpha = hparams['alpha']
           beta_1 = hparams['beta_1']
           beta_2 = hparams['beta_2']
```

```python
    #CODE HERE

    f_x = beta_1*f_x + (1-beta_1)*dx
    f_y = beta_1*f_y + (1-beta_1)*dy
    s_x = beta_2*s_x + (1-beta_2)*dx**2
    s_y = beta_2*s_y + (1-beta_2)*dy**2
    i = 0.001 if i == 0 else i
    f_x_hat = f_x/(1-beta_1**i)
    f_y_hat = f_y/(1-beta_1**i)
    s_x_hat = s_x/(1-beta_2**i)
    s_y_hat = s_y/(1-beta_2**i)
    x = x - alpha*f_x_hat/((s_x_hat + eps)**0.5)
    y = y - alpha*f_y_hat/((s_y_hat + eps)**0.5)
    return x, y, f_x, f_y, s_x, s_y
```

Study in detail the class Optimizer that has been implemented below, because you will be using it to study the effect of key hyperparameters on the optimization process, and the difference that the bells and whistles on Gradient Descent can make. The **fit** function calls the methods you have implemented above depending on how you initialize the class.

```python
[316]: from random import seed
       import math

       seed(1)

       class Optimizer:

           def __init__(self, x_init, y_init, method, func, hparams):

               '''
               x_init: Initialization x point
               y_init: Initialization y point
               method: adam, adagrad, sgrad, ... check the fit function below
               func: function to optimize [mult, easom, monkey, bowl, matyas, ...]
               hparams: alpha, gamma, eps, beta, beta_1, beta_2
               '''

               f, optima = func()

               self.x = x_init
               self.y = y_init
               self.hparams = hparams
               self.first = True
               self.iter = 0
               self.x_list = []
               self.y_list = []
```

```python
        cp = optima()
        self.cp_x = np.asscalar(np.array([0]))
        self.cp_y = np.asscalar(np.array([1]))

        self.method = method
        self.f = f
        self.f_gradx = grad(f, 0)
        self.f_grady = grad(f, 1)
        self.count = 0

    def distance(self, x, y):
        '''
        L2 Norm
        '''

        return math.sqrt((x-self.cp_x)**2 + (y-self.cp_y)**2)

    def cgrad(self, x, y):
        '''
        Uses autograd
        '''

        return self.f_gradx(x, y), self.f_grady(x, y)

    def fit(self, epochs):

        '''
        Epochs: max number of updates to be made
        '''


        self.x_list = []
        self.y_list = []
        self.z_list = []

        if self.method == 'grad':

            '''
            USES
            alpha: learning rate'
            '''

            for i in range(epochs):

                self.x_list.append(self.x)
                self.y_list.append(self.y)
```

```python
                self.z_list.append(self.f(self.x, self.y))

                dx, dy = self.cgrad(self.x, self.y)
                print('dx:',dx,'dy:',dy)
                self.x, self.y = gradient_descent(self.x, self.y, dx, dy, self.
→hparams)

                if (self.distance(self.x, self.y) < 1 and self.first):
                    self.iter = i
                    self.first = False


        if self.method == 'sgrad':

            '''
            USES
            alpha: learning rate
            '''

            for i in range(epochs):

                self.x_list.append(self.x)
                self.y_list.append(self.y)
                self.z_list.append(self.f(self.x, self.y))

                dx, dy = self.cgrad(self.x, self.y)

                self.x, self.y = stochastic_gradient_descent(self.x, self.y,
→dx, dy, self.hparams)

                if (self.distance(self.x, self.y) < 1 and self.first):
                    self.iter = i
                    self.first = False


        if self.method == 'grad_momentum':

            '''
            USES
            alpha: learning rate
            gamma: momentum factor
            '''

            v_x, v_y = 0., 0.
            for i in range(epochs):
```

```python
                self.x_list.append(self.x)
                self.y_list.append(self.y)
                self.z_list.append(self.f(self.x, self.y))

                dx, dy = self.cgrad(self.x, self.y)

                self.x, self.y, v_x, v_y = momentum(self.x, self.y, dx, dy,␣
→v_x, v_y, self.hparams)

                if (self.distance(self.x, self.y) < 1 and self.first):
                    self.iter = i
                    self.first = False

        if self.method == 'adagrad':

            '''
            USES
            alpha: learning rate
            eps:
            '''

            v_x, v_y = 0., 0.
            for i in range(epochs):

                self.x_list.append(self.x)
                self.y_list.append(self.y)
                self.z_list.append(self.f(self.x, self.y))

                dx, dy = self.cgrad(self.x, self.y)

                self.x, self.y, v_x, v_y = adagrad(self.x, self.y, dx, dy, v_x,␣
→v_y,self.hparams)

                if (self.distance(self.x, self.y) < 1 and self.first):
                    self.iter = i
                    self.first = False

        if self.method == 'adam':

            '''
            USES
            alpha: learning rate
            beta_1:
            beta_2:
            eps:
            '''
```

```
            f_x, f_y = 0., 0.
            s_x, s_y = 0., 0.
            for i in range(epochs):

                self.x_list.append(self.x)
                self.y_list.append(self.y)
                self.z_list.append(self.f(self.x, self.y))

                dx, dy = self.cgrad(self.x, self.y)

                self.x, self.y, f_x, f_y, s_x, s_y = adam(self.x, self.y, dx,
    ↪dy, f_x, f_y, s_x, s_y, i, self.hparams)

                if (self.distance(self.x, self.y) < 1 and self.first):
                    self.iter = i
                    self.first = False
```

The code at the bottom provides helper functions to help you visualize your results and play around with your optimizer implementations.

```
[317]: def animate(i):
           '''Plotting helper'''

           i = int(i*(epochs/frames))
           line1.set_data(optim.x_list[:i+1], optim.y_list[:i+1])
           line1.set_3d_properties(optim.z_list[:i+1])
           line2.set_data(optim.x_list[:i+1], optim.y_list[:i+1])
           line2.set_3d_properties(np.zeros(i+1) -1)
           title.set_text('Epoch: {: d}, Error: {:.4f}'.format(i, optim.z_list[i]))

           return line1, line2, title



       def plot_function(epochs, frames, func, xmin = -4.5, xmax= 4.5, ymin = -4.5,
        ↪ymax = 4.5, step = 0.2, option = '3d'):

           '''
           Plot the optimization
           epochs: number of optimization improvements
           frames: number of displayed frames
           func: function to plot. eg. mult, monkey, ....
           xmin: Min value of x to be displayed
           xmax: Max value of x to be displayed
           ymin: Min value of y to be displayed
           ymax: Max value of y to be displayed
```

```python
    step: Split between xmin and xmax, ymin and ymax
    option: keep this option '3d'

    '''

    x, y = np.meshgrid(np.arange(xmin, xmax + step, step), np.arange(ymin, ymax
→+ step, step))
    f, opt = func()
    z = f(x, y)
    cp = opt()

    optima = cp.reshape(-1, 1)

    if (option == '3d'):

        fig = plt.figure(figsize=(12,6), dpi = 100)
        ax = fig.add_subplot(1,2,1,projection='3d')

        ax.plot_surface(x, y, z,  rstride=5, cstride=5, alpha = 0.5, cmap=plt.
→cm.plasma)
        cset = ax.contourf(x, y, z, 25, zdir='z', offset=-1, alpha=0.6,
→cmap=plt.cm.coolwarm)
        out1 = f(*optima)
        ax.plot(*optima, out1 , 'r*', markersize=10)

        ax.set_xlabel('$x$')
        ax.set_ylabel('$y$')
        ax.set_zlabel('$z$')


        i = 0

        for i in range(epochs):
            line1, = ax.plot(optim.x_list[:i+1], optim.y_list[:i+1], optim.
→z_list[:i+1], color ='black', marker = '.')

            line2, = ax.plot(optim.x_list[:i+1], optim.y_list[:i+1], np.
→zeros(i+1)-1, color ='red', marker='.')


    if (option == '2d'):

        fig = plt.figure(dpi = 100)
        ax = plt.subplot(111)

        cset = ax.contourf(x, y, z, 25, zdir='z', offset=-1, alpha=0.6,
→cmap=plt.cm.bwr)
```

```
        dz_dx = egrad(f, argnum=0)(x, y)
        dz_dy = egrad(f, argnum=1)(x, y)
        ax.quiver(x, y, x- dz_dx, y-dz_dy, alpha = 0.5)
        ax.plot(*optima, 'r*', markersize = 18)

        ax.set_xlabel('$x$')
        ax.set_ylabel('$y$')


    ax.set_xlim((xmin, xmax))
    ax.set_ylim((ymin, ymax))

    plt.show
```

[318]:
```
'''
Test plot: Here, we plot optimization with adam on bowl function for 200␣
 ↪updates,
initialized at (6., 6.).
'''

frames = 20
epochs = 200

func = bowl
init_x, init_y = 6., 6.

hparams = {'alpha': 0.1, 'eps': 1e-7, 'beta_1': 0.8, 'beta_2': 0.99}
optim = Optimizer(init_x, init_y, 'adam', func, hparams)
optim.fit(epochs)

plot_function(epochs, frames, func)
```
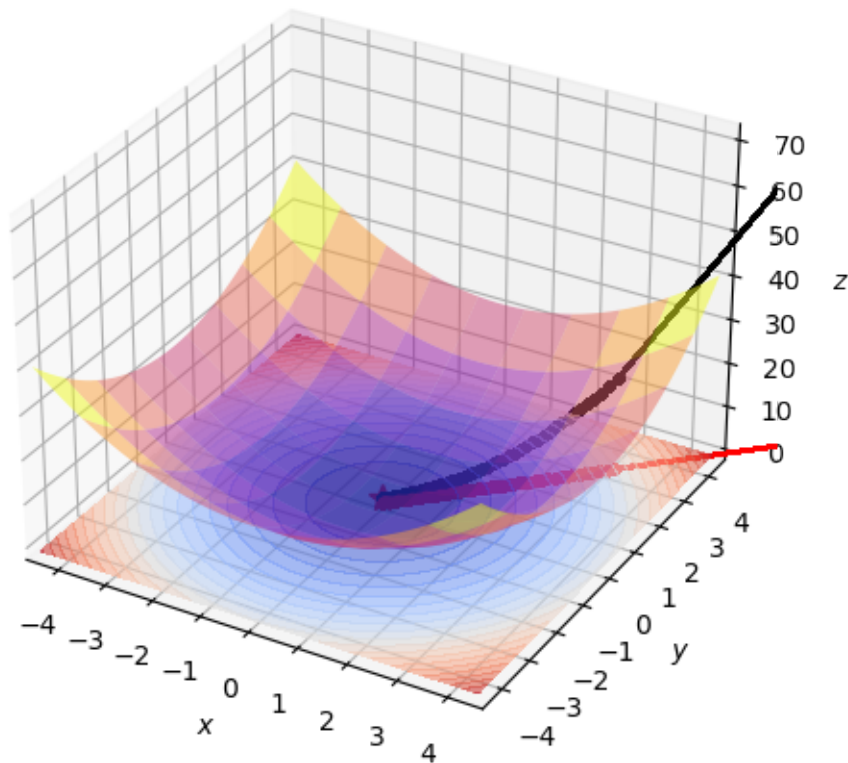
4   Importance of learning rate

Call the plotting function and Optimizer as shown above and make two plots to show how too high or too low of a learning rate could be a problem for stochastic gradient descent on the bowl function. Keep the number of iterations (epochs) constant across the two to demonstrate the effect. You are recommended to use the bowl function.

```
[319]:  frames = 20
        epochs = 20

        '''
        TODO

        Make a plot to show what happens if the learning rate is too high with sgrad.

        Recommended hparams:

        initialization: (6., 6.)
        alpha: 2
        '''
```

```
func = bowl
#CODE HERE
init_x, init_y = 6., 6.

hparams = {'alpha': 2}
optim = Optimizer(init_x, init_y, 'sgrad', func, hparams)
optim.fit(epochs)

plot_function(epochs, frames, func)
```
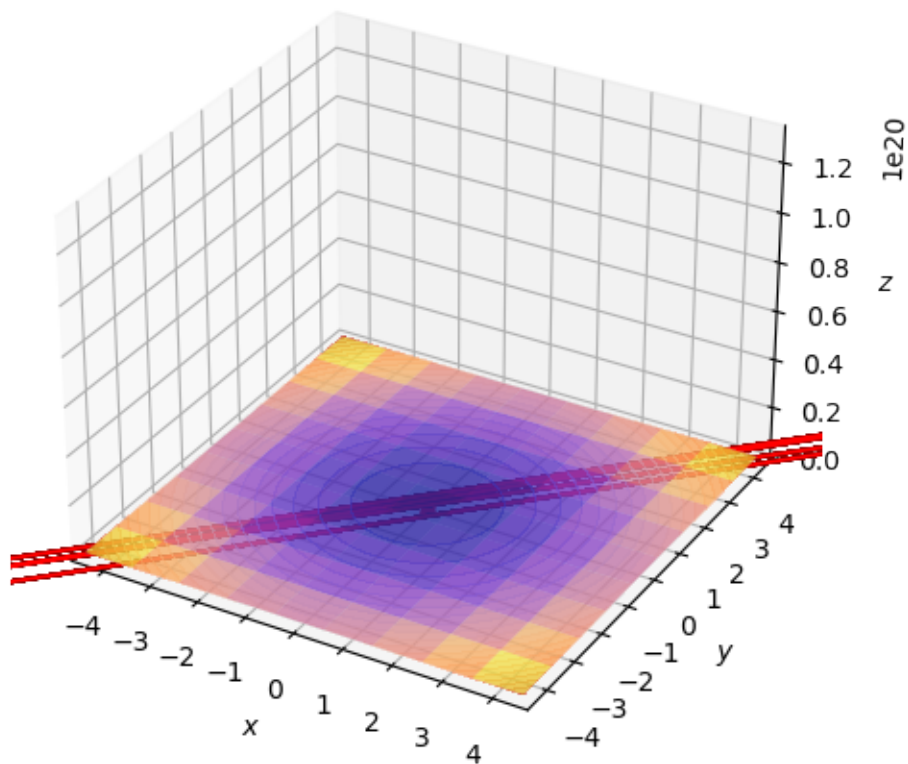

[327]:
```
frames = 20
epochs = 20

'''
TODO

Make a plot to show what happens if the learning rate is too low with sgrad.

Recommended hparams:
```

```
initialization: (6., 6.)
alpha: 0.01
'''

func = bowl
#CODE HERE
init_x, init_y = 6., 6.

hparams = {'alpha': 0.01}
optim = Optimizer(init_x, init_y, 'sgrad', func, hparams)
optim.fit(epochs)

plot_function(epochs, frames, func)
```
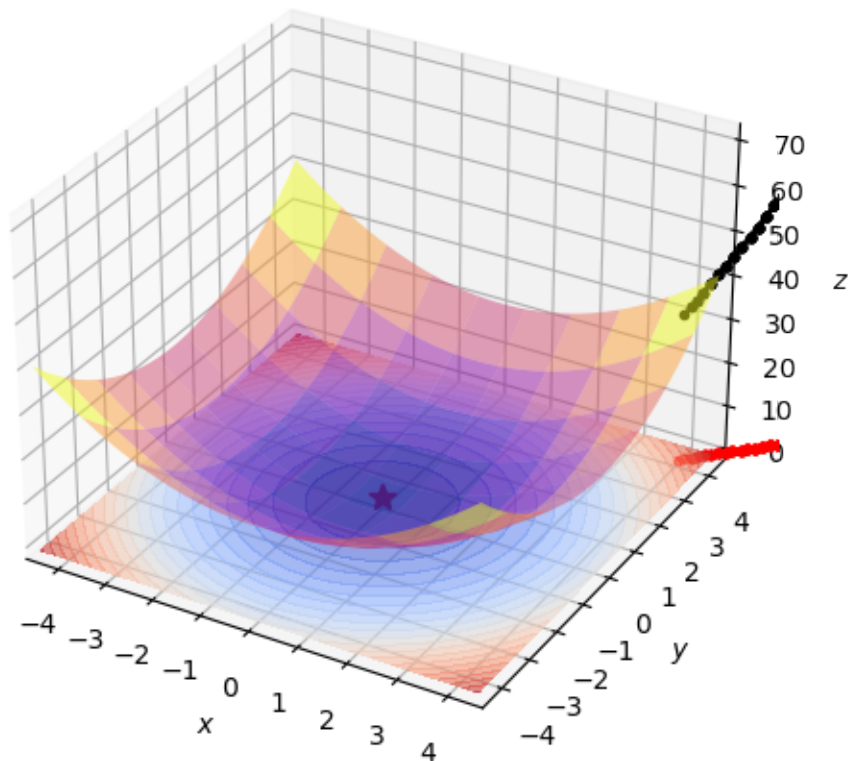

[321]:
```
'''
TODO

Make a plot to show what happens if the learning rate is just right with sgrad.
```

```
Recommended hparams:

initialization: (6., 6.)
alpha: 0.1
'''

frames = 20
epochs = 20

func = bowl
#CODE HERE
init_x, init_y = 6., 6.

hparams = {'alpha': 0.1}
optim = Optimizer(init_x, init_y, 'sgrad', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```
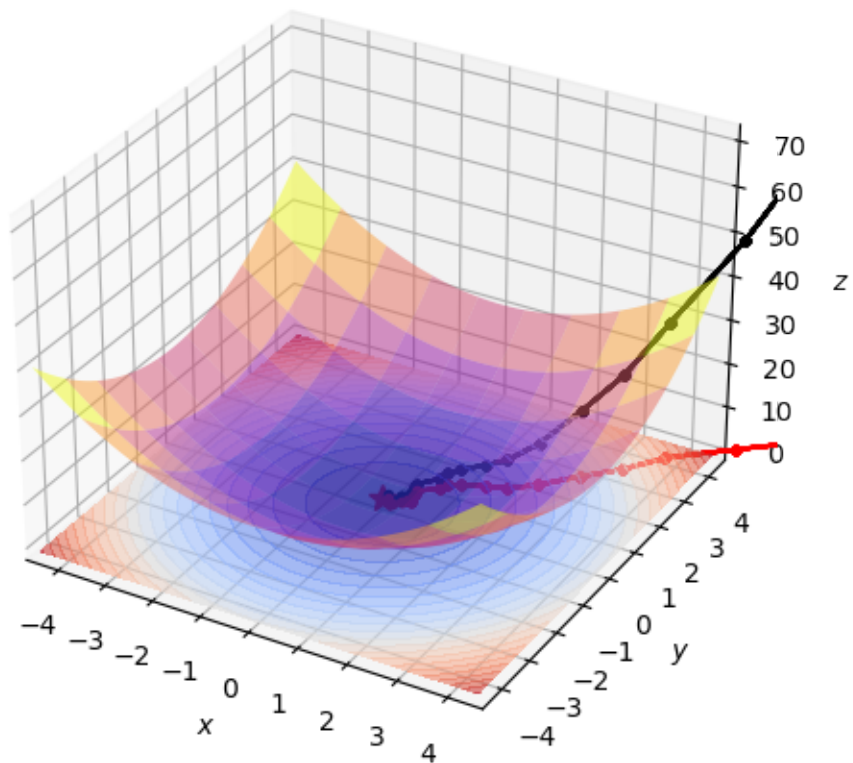
5 SGD and Matyas: momentum helps

Make two plots to show how momentum possibly helps with navigating a taco-shell function like Matyas, where gradient is low in one direction and high in another direction. Keep all parameters except the type of optimizer constant.

```
[322]:  frames = 20
        epochs = 20

        '''
        TODO

        Make a plot to show what happens if stochastic gradient descent is used on a
        function like this.

        Recommended hparams:

        initialization: (4., -4.)
        alpha: 0.01
        '''

        func = matyas
        #CODE HERE
        init_x, init_y = 4., -4.

        hparams = {'alpha': 0.01}
        optim = Optimizer(init_x, init_y, 'sgrad', func, hparams)
        optim.fit(epochs)
        plot_function(epochs, frames, func)
```
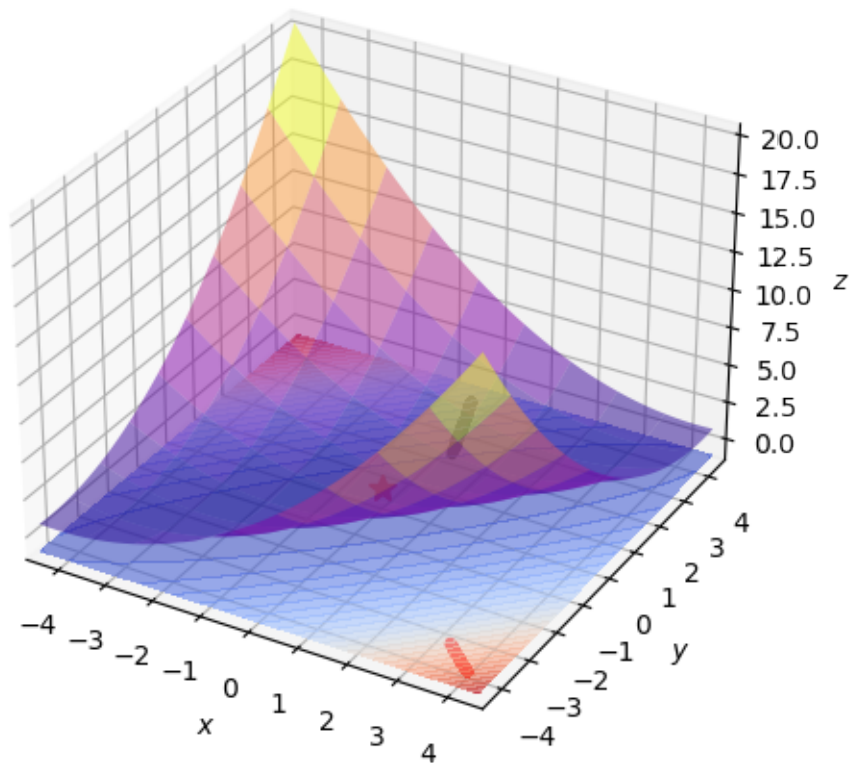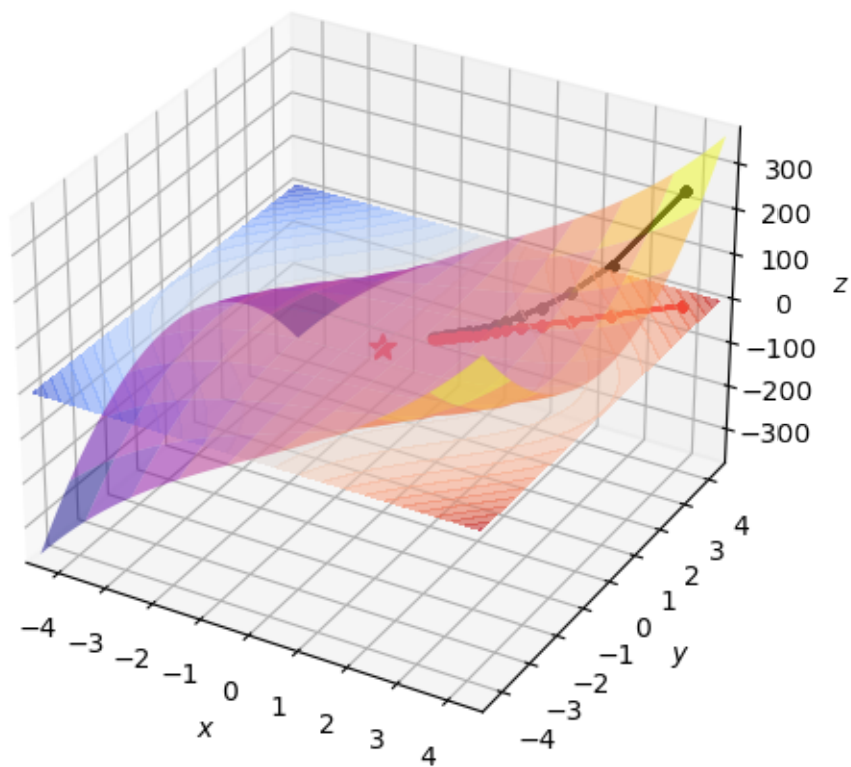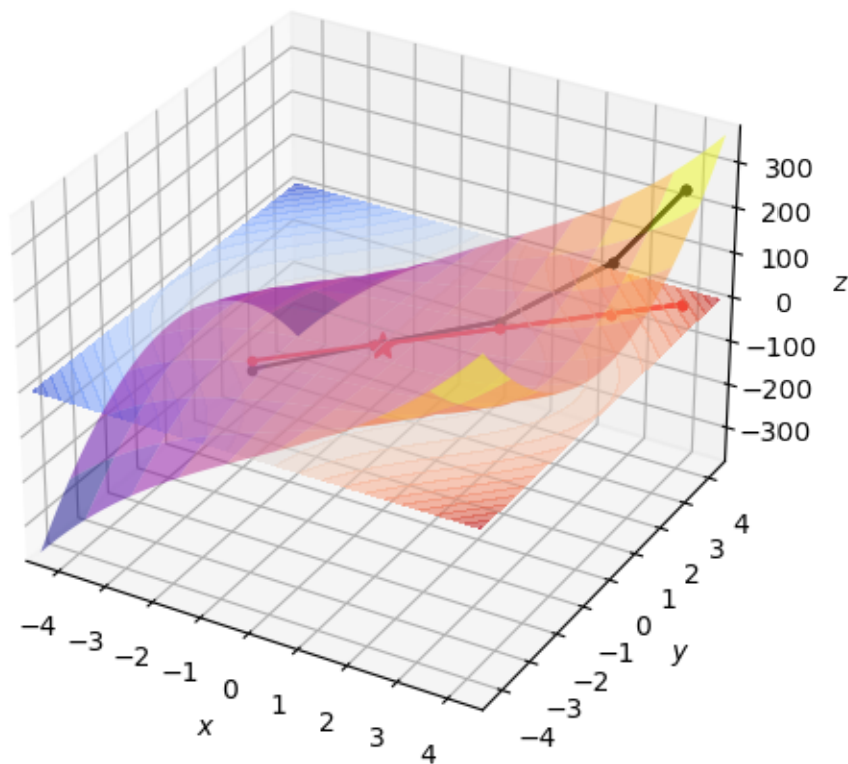
[328]:
```
frames = 20
epochs = 20

'''
TODO

Make a plot to show what happens if stochastic gradient descent with momentum
is used on a function like this.

Recommended hparams:

initialization: (4., -4.)
alpha: 0.01
gamma: 0.99
'''

func = matyas
#CODE HERE
init_x, init_y = 4., -4.
```

```
hparams = {'alpha': 0.01, 'gamma': 0.99}
optim = Optimizer(init_x, init_y, 'grad_momentum', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```
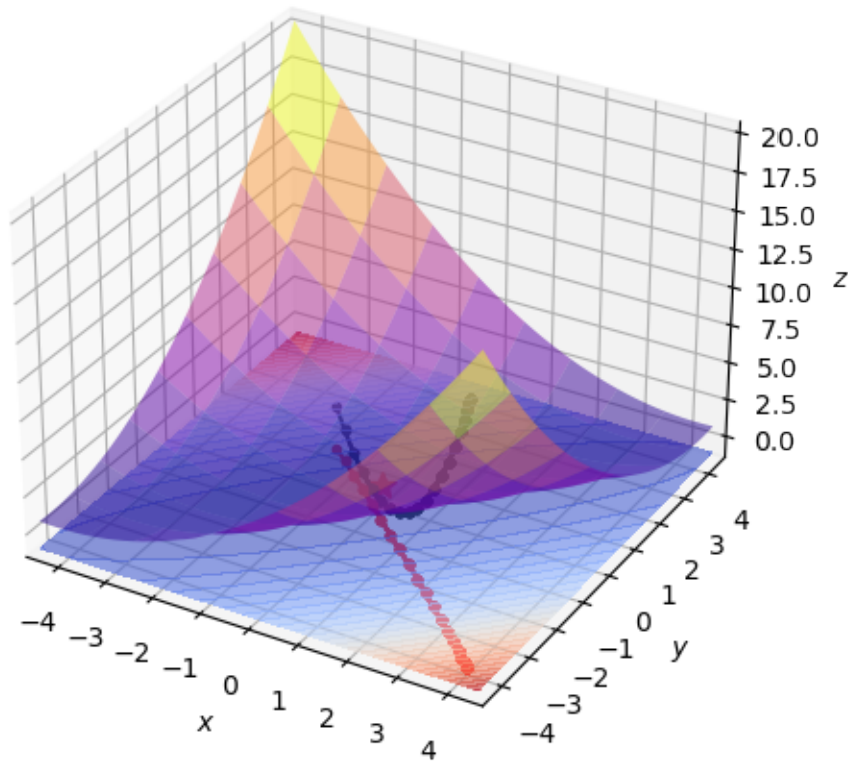

6 Exploring Saddle Points

Make three plots to show how SGD, momentum, and adagrad behave on the monkey function. A good optimization algorithm will move past the flat region and continue its trajectory downward. Do not change the learning rate as this has been artificially reduced to make the distinction clear on this problem. Do you notice the observations made in the text above?

```
[329]: frames = 20
       epochs = 20

       '''
       TODO

       Make a plot to show what happens if stochastic gradient descent is used on a
       function like this.
```
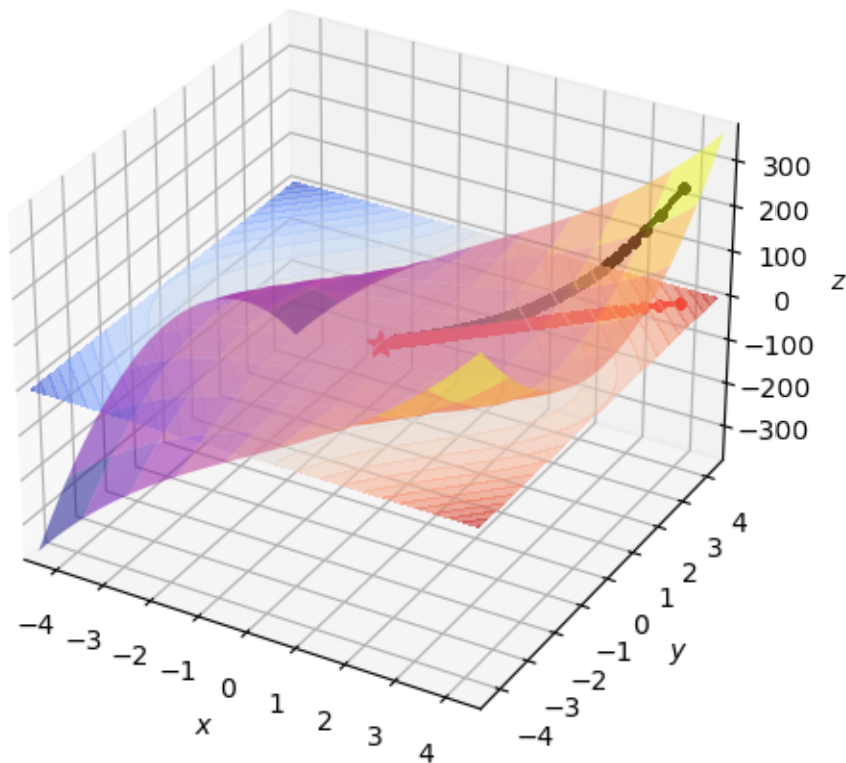
```
Recommended hparams:

initialization: (4., 4.)
alpha: 0.01
'''

func = monkey
#CODE HERE
init_x, init_y = 4., 4.

hparams = {'alpha': 0.01}
optim = Optimizer(init_x, init_y, 'sgrad', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```


[330]:
```
frames = 20
epochs = 5

'''
```

```
TODO

Make a plot to show what happens if stochastic gradient descent with momentum
is used on a function like this.

Recommended hparams:

initialization: (4., 4.)
alpha: 0.01
gamma: 0.99
'''

func = monkey
#CODE HERE
init_x, init_y = 4., 4.

hparams = {'alpha': 0.01, 'gamma': 0.99}
optim = Optimizer(init_x, init_y, 'grad_momentum', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```

```
frames = 20
epochs = 1000

'''
TODO

Make a plot to show what happens if adagrad
is used on a function like this.

Recommended hparams:

initialization: (4., 4.)
alpha: 0.3
eps: 1e-7
'''

func = monkey
#CODE HERE
init_x, init_y = 4., 4.

hparams = {'alpha': 0.3, 'eps': 1e-7}
optim = Optimizer(init_x, init_y, 'adagrad', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```

7 Escaping Local Minima

Make 3 or more plots (with atleast one plot using ADAM) to show how the different functions escape the local minima on the **mult** function. We will initialize at (0.5, 0.5).

```
[345]:  frames = 20
        epochs = 20

        '''
        TODO

        Make a plot to show what happens if ADAM
        is used on a function like this.

        Recommended hparams:

        initialization: (.8, .8)
        alpha: 0.7
        eps: 1e-7
        beta_1: 0.99
```

```
beta_2: 0.9
'''



func = mult
#CODE HERE
init_x, init_y = .8, .8

hparams = {'alpha': 0.7, 'eps': 1e-7, 'beta_1': 0.99, 'beta_2': 0.9}
optim = Optimizer(init_x, init_y, 'adam', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```
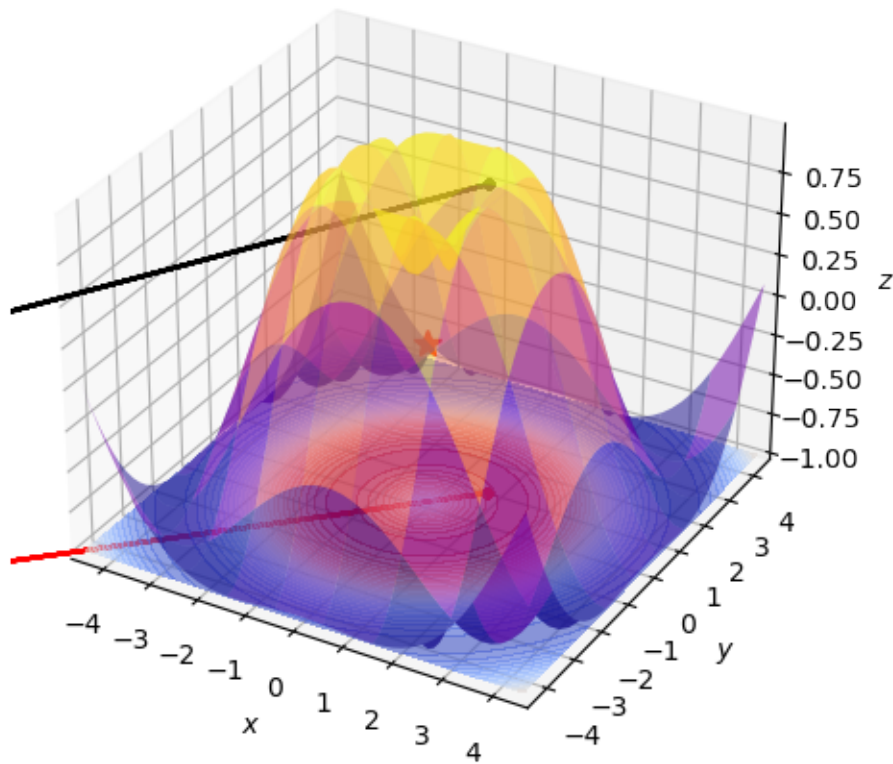

[334]:
```
frames = 20
epochs = 20

'''
TODO

Make a plot to show what happens if adagrad
```

```
is used on a function like this.

Recommended hparams:

initialization: (.8, .8)
alpha: 0.7
eps: 1e-7
'''

func = mult
#CODE HERE
init_x, init_y = .8, .8

hparams = {'alpha': 0.7, 'eps': 1e-7}
optim = Optimizer(init_x, init_y, 'adagrad', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```
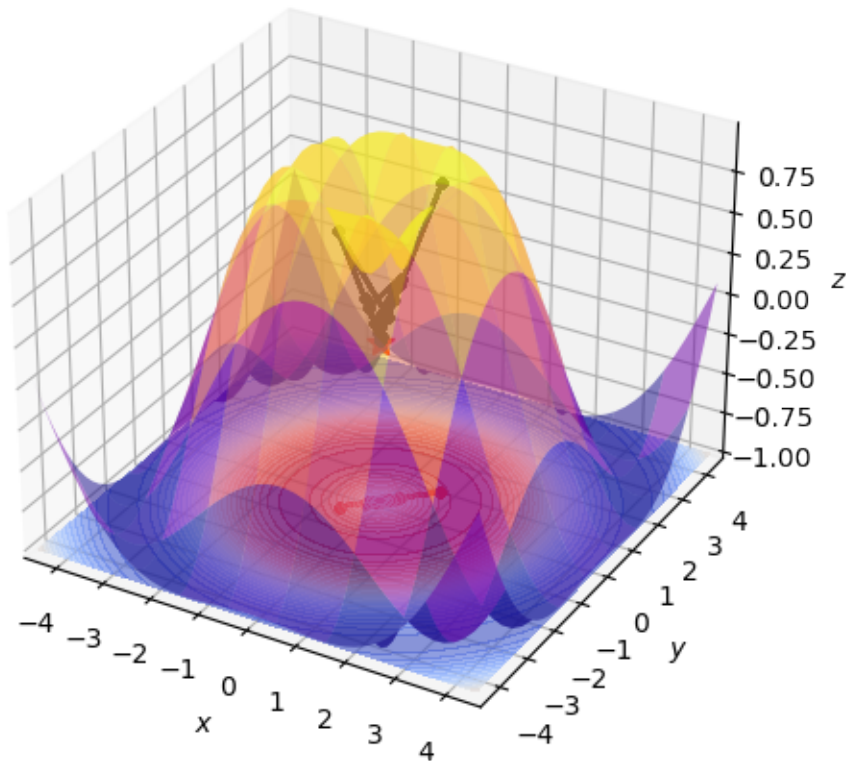

[335]:
```
frames = 20
epochs = 20
```

```python
'''
TODO

Make a plot to show what happens if stochastic gradient descent with momentum
is used on a function like this.

Recommended hparams:

initialization: (.8, .8)
alpha: 0.7
gamma: 0.99
'''


func = mult
#CODE HERE
init_x, init_y = .8, .8

hparams = {'alpha': 0.7, 'gamma': 0.99}
optim = Optimizer(init_x, init_y, 'grad_momentum', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```
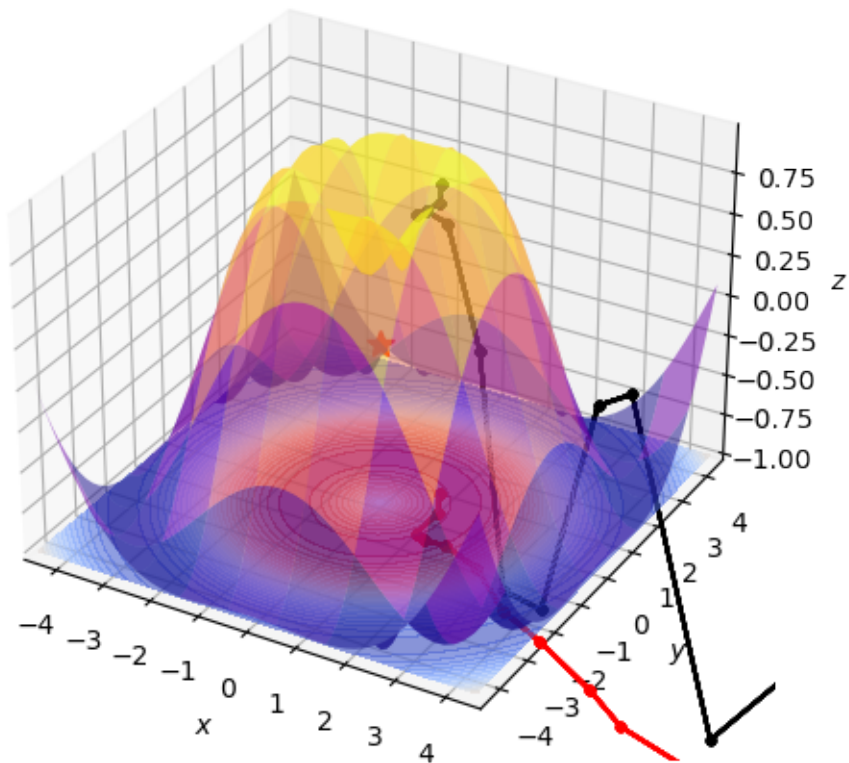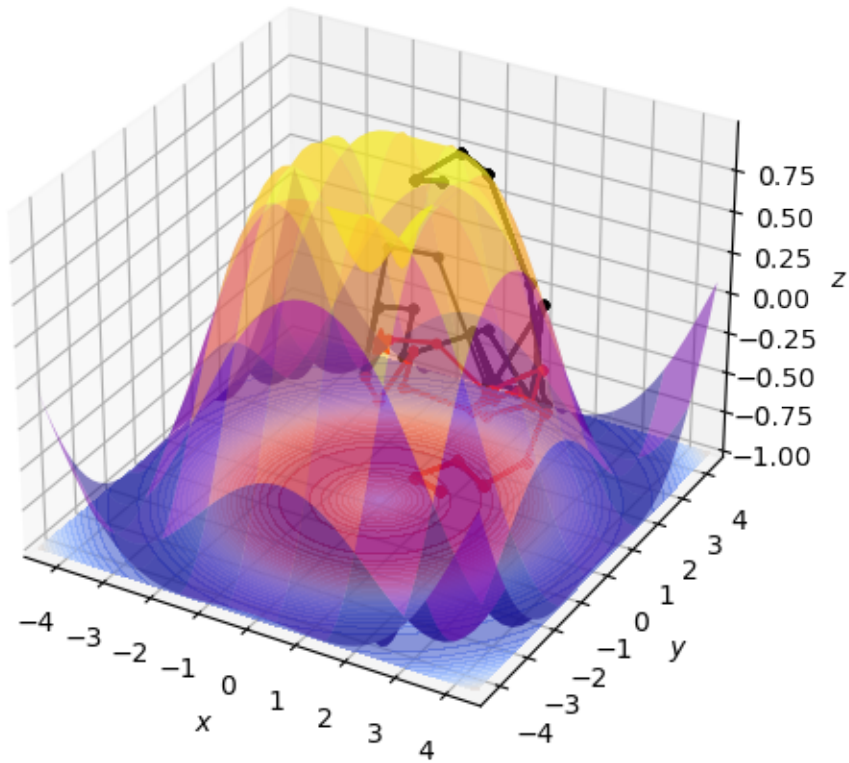
[343]:
```
frames = 20
epochs = 20


'''
TODO

Make a plot to show what happens if stochastic gradient descent with momentum
is used on a function like this.

Recommended hparams:

initialization: (.8, .8)
alpha: 0.7
'''

func = mult
#CODE HERE
init_x, init_y = .8, .8
```

```
hparams = {'alpha': 0.7}
optim = Optimizer(init_x, init_y, 'sgrad', func, hparams)
optim.fit(epochs)
plot_function(epochs, frames, func)
```


[ ]: