Week 6 Assignment

Neimhin Robinson Gunning, 16321701

28th March 2024

(a) (i) Implementing mini-batch Stochastic Gradient Descent

Our global loss function is

$$f_T(x) = \sum_{w \in T} \frac{loss(x, w)}{\#W}$$

which is just the average of $loss(w, x)$ ranging over the entire dataset, T. We can also calculate an approximation of the loss using a subset (minibatch) of T.

$$f_N(x) = \sum_{w \in N} \frac{loss(x, w)}{\#N}$$

This is implemented on line 17 of Algorithm 1. We can also approximate the gradient w.r.t. to the minibatch rather than the full training dataset. In these experiments we use the finite difference methods to estimate the mini-batch gradient according to

$$\frac{df_N}{dx_i} \approx \frac{f_N([x_1, \ldots, x_i + \epsilon, \ldots, x_d]) - f_N(x)}{\epsilon}$$

where we set $\epsilon = 10^{-15}$ for the remainder of this discussion. We also look at only at an example with $d = 2$ so the finite difference gradient is:

$$\nabla f_N = [\frac{f_N([x_1 + \epsilon, x_2]) - f_N(x)}{\epsilon}, \frac{f_N([x_1, x_2 + \epsilon]) - f_N(x)}{\epsilon}]$$

The code implementation of this is on line 4 in Algorithm 1.

To generate mini-batches we first shuffle the rows data set and then take successive slices with n rows, where n is the mini-batch size. The first mini-batch consists of the 1st to the nth data items, the second consists of the $(n+1)th$ to the $(n+n)th$, etc. If we reach the end of the dataset before filling the minibatch we shuffle the dataset and start again from index 1. This is implemented on line 10 of Algorithm 1.

The implementation of mini-batch SGD here relies on generating successive f_{N_t} and ∇f_{N_t}, where N_t is the mini-batch for iteration t. This is implemented on line 40 of Algorithm 1.

At each iteration the step size can be calculated with respect to f_{N_t} and ∇f_{N_t} using of the Polyak, RMSProp, Heavy Ball, and Adam methods. Each of the step types are implemented in `src/sgd.py` which is included in the appendix.

Algorithm 1 Generating mini-batches, N, and associated f_N and ∇f_N.

src/ai.py   Sun Mar 24 17:42:18 2024   1

```python
import numpy as np
import sympy as sp

def gradient_function_fd(minibatch, epsilon=10**(-15)):
    def gradient_fd(x):
        dydx1 = (f(x + np.array([epsilon, 0]), minibatch) - f(x, minibatch)) / epsilon
        dydx2 = (f(x + np.array([0, epsilon]), minibatch) - f(x, minibatch)) / epsilon
        return np.array([dydx1, dydx2])
    return gradient_fd

def loss(x, w):
    z = x - w - 1
    left = 10 * (z[0]**2+z[1]**2)
    right = (z[0]+2)**2+(z[1]+4)**2
    return min(left, right)

def f_clear(x, minibatch):
    return sum(loss(x, w) for w in minibatch) / len(minibatch)

def generate_minibatches(T, n=5, seed=42, shuffle=True):
    if shuffle:
        T = T.copy()
        np.random.seed(seed)
        np.random.shuffle(T)
    num_rows = T.shape[0]
    i = 0

    minibatch = np.zeros((n, T.shape[1]), T.dtype)
    while True:
        for j in range(n):
            minibatch[j] = T[i % num_rows]
            i += 1
            if shuffle and i >= num_rows:
                # begin next epoch
                np.random.shuffle(T)
                i = 0
        current_minibatch = minibatch
        yield minibatch

def generate_optimisation_functions(batch, minibatch_size=5, finite_difference=True, **kwargs):
    minibatch_generator = generate_minibatches(
        batch, n=minibatch_size, **kwargs)
    for minibatch in minibatch_generator:
        def optim_func(x):
            return f_clear(x, minibatch)
        gradf = gradient_function_fd(minibatch)
        yield (optim_func, gradf)
    yield "finished"
```

```python
import lib
import numpy as np
import json


def iterate(self):
    self._x_value = self._start
    self._old_x_value = None
    self._iteration = 0
    self._m = np.zeros(self._x_value.shape, dtype=np.float64)
    self._v = np.zeros(self._x_value.shape, dtype=np.float64)
    self._converged_value = False
    self._grad_value = self._gradient(self._x_value)

    yield self.state_dict()

    while not self._converged_value:
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        self._grad_value = self._gradient(self._x_value)
        self._m = self._beta * self._m + (1-self._beta)*self._grad_value
        # grad_value * grad_value gives element-wise product of np array
        self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
        self._old_x_value = self._x_value
        self._iteration += 1
        m_hat = self._m / (1-(self._beta ** self._iteration))
        v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
        v_hat_aug = v_hat**(0.5) + self._epsilon
        self._adam_grad = m_hat / v_hat_aug
        self._x_value = self._x_value - self._step_size * self._adam_grad
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        yield self.state_dict()
```

```python
import numpy as np
import sympy as sp

def gradient_function_fd(minibatch, epsilon=10**(-15)):
    def gradient_fd(x):
        dydx1 = (f(x + np.array([epsilon, 0]), minibatch) - f(x, minibatch)) / epsilon
        dydx2 = (f(x + np.array([0, epsilon]), minibatch) - f(x, minibatch)) / epsilon
        return np.array([dydx1, dydx2])
    return gradient_fd

def loss(x, w):
    z = x - w - 1
    left = 10 * (z[0]**2+z[1]**2)
    right = (z[0]+2)**2+(z[1]+4)**2
    return min(left, right)

def f_clear(x, minibatch):
    return sum(loss(x, w) for w in minibatch) / len(minibatch)

def generate_minibatches(T, n=5, seed=42, shuffle=True):
    if shuffle:
        T = T.copy()
        np.random.seed(seed)
        np.random.shuffle(T)
    num_rows = T.shape[0]
    i = 0

    minibatch = np.zeros((n, T.shape[1]), T.dtype)
    while True:
        for j in range(n):
            minibatch[j] = T[i % num_rows]
            i += 1
            if shuffle and i >= num_rows:
                # begin next epoch
                np.random.shuffle(T)
                i = 0
        current_minibatch = minibatch
        yield minibatch

def generate_optimisation_functions(batch, minibatch_size=5, finite_difference=True, **kwargs):
    minibatch_generator = generate_minibatches(
        batch, n=minibatch_size, **kwargs)
    for minibatch in minibatch_generator:
        def optim_func(x):
            return f_clear(x, minibatch)
        gradf = gradient_function_fd(minibatch)
        yield (optim_func, gradf)
    yield "finished"
```

```python
import lib


def iterate(self):
    self._x_value = self._start
    self._old_x_value = None
    self._iteration = 0
    self._converged_val = False
    self._grad_value = self._gradient(self._x_value)
    self._z = 0
    yield self.state_dict()  # yield initial values

    while not self._converged_val:
        self._iteration += 1
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._z = self._beta * self._z + self._step_size * self._grad_value
        self._x_value = self._x_value - self._z
        self._converged_val = self._converged(self._x_value, self._old_x_value)
        yield self.state_dict()
```

```python
import sympy as sp
import numpy as np
import functools

x, y = sp.symbols('x y', real=True)
f = 3 * (x - 5)**4 + (10 * ((y - 9)**2))
g = sp.Max(x - 5, 0) + (10 * sp.Abs(y - 9))
relu = sp.Max(x,0)

def f_real(x, y):
    return 3 * (x - 5)**4 + 10 * (y - 9)**2


def g_real(x, y):
    return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)

def relu_real(x):
    return np.maximum(x,0)


def apply_sym(x, f):
    for x_sym, x_val in zip(f.free_symbols, x):
        f = f.subs(x_sym, x_val)
    return f

config = {
    "f": {
        "sym": f,
        "real": f_real,
        "name": "f",
    },
    "g": {
        "sym": g,
        "real": g_real,
        "name": "g",
    },
    "relu": {
        "sym": relu,
        "real": lambda x: max(x, 0),
        "name": "relu",
    }
}

class GradientDescent():
    def __init__(self):
        self._max_iter = 1000
        self._debug = False
        self._converged = lambda x1, x2: False
        self._epsilon = 0.0001
        self._dimension = None
        self._beta = 0
        self._algorithm = None
        self._iteration = None
        self._function = None
        self._sum = None
        self._x_value = None
        self._step_coeff = None
        self._converged_value = None
        self._grad_value = None
        self._m = None
        self._v = None
        self._adam_grad = None
        self._beta = None
        self._beta2 = None
        self._step_size = None
        self._z = None
        self._f_star = None

    def step_size(self, a):
        self._step_size = a
        return self

    def beta(self, b):
        self._beta = b
        return self

    def beta2(self, b):
        self._beta2 = b
        return self

    def epsilon(self, e):
        self._epsilon = e
        return self

    def function(self, f, function_name=None, dimension=None):
        self._function = f
        self.function_name = function_name
        self._dimension = dimension
        return self

    def sym_function(self, function, function_name=None):
        self.function_name = function_name
        self._dimension = len(function.free_symbols)
        def fn(x):
            return apply_sym(x, function)

        diffs = [function.diff(var) for var in function.free_symbols]

        def grad(x):
            return np.array([
```

```
                apply_sym(x, diff) for diff in diffs])

        self._function = fn
        self._gradient = grad
        return self

    def gradient(self, g):
        self._gradient = g
        return self

    def max_iter(self, m):
        self._max_iter = m
        return self

    def start(self, s):
        self._start = s
        self._x_value = s
        return self

    def debug(self, d):
        self._debug = d
        return self

    def converged(self, c):
        self._converged = c
        return self

    def set_iterate(self, f):
        self.iterate = functools.partial(f, self)
        return self

    def algorithm(self, alg):
        self._algorithm = alg
        if self._algorithm == "rmsprop":
            import rmsprop
            self.set_iterate(rmsprop.iterate)
        elif self._algorithm == "adam":
            import adam
            self.set_iterate(adam.iterate)
        elif self._algorithm == "heavy_ball":
            import heavy_ball
            self.set_iterate(heavy_ball.iterate)
        else:
            raise Exception("Unknown algorithm:" + alg)
        return self

    def state_dict(self):
        print(self._function(self._x_value))
        return {
            "alg": self._algorithm,
            "function_name": self.function_name,
            "iteration": self._iteration,
            "step_coeff": self._step_coeff,
            "adam_grad": self._adam_grad,
            "f(x)": self._function(self._x_value),
            "epsilon": self._epsilon,
            "converged": self._converged_value,
            "gradient": self._grad_value,
            "m": self._m,
            "v": self._v,
            "beta1": self._beta,
            "beta2": self._beta2,
            "alpha": self._step_size,
            "sum": self._sum,
            "z": self._z,
            **{"x" + str(i): self._x_value[i] for i in range(len(self._x_value))},
        }

    def run2csv(self, fname, summarise=True):
        import pandas as pd
        iterations = list(self.iterate())
        df = pd.DataFrame(iterations)
        df.to_csv(fname)
        if(summarise):
            with open(fname + ".summary", "w") as f:
                print(f"iterations: {len(df)}", file=f)
                print(f"start: {df['x0'][0]} {df['x1'][0]}", file=f)
                print(f"final: {df['x0'][len(df) - 1]} {df['x1'][len(df) - 1]}", file=f)


if __name__ == "__main__":
    print(f.diff(x), f.diff(y))
    print(g.diff(x), g.diff(y))
```

```python
import pandas as pd
import matplotlib.pyplot as plt


df = pd.read_csv("data/T.csv")


plt.scatter(df["0"], df["1"])
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.title("Traning Data")
plt.savefig("fig/T.pdf")
```

```python
import numpy as np
import pandas as pd
import week6
import matplotlib.pyplot as plt
import argparse
from mpl_toolkits.mplot3d import Axes3D

ap = argparse.ArgumentParser()
ap.add_argument("--show",action="store_true")
args = ap.parse_args()

# Global variables for extents
x_min, x_max = -5, 5
y_min, y_max = -5, 5

def plot_wireframe_and_contour(f, T, resolution=100):
    global x_min, x_max, y_min, y_max

    # Generate data for wireframe plot
    x_range = np.linspace(x_min, x_max, resolution)
    y_range = np.linspace(y_min, y_max, resolution)
    X, Y = np.meshgrid(x_range, y_range)
    Z = np.zeros_like(X)
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f([X[i, j], Y[i, j]], T)

    # Plot wireframe
    fig = plt.figure(figsize=(12, 6))

    ax_wireframe = fig.add_subplot(121, projection='3d')
    ax_wireframe.plot_wireframe(X, Y, Z, color='blue')
    ax_wireframe.set_xlabel('X')
    ax_wireframe.set_ylabel('Y')
    ax_wireframe.set_zlabel('f(x, T)')
    ax_wireframe.set_title('Wireframe Plot of f(x, T)')

    # Generate data for contour plot
    Z_contour = np.zeros_like(X)
    for i in range(resolution):
        for j in range(resolution):
            Z_contour[i, j] = f([X[i, j], Y[i, j]], T)

    # Plot contour
    ax_contour = fig.add_subplot(122)
    contour = ax_contour.contourf(X, Y, Z_contour, levels=20, cmap='viridis')
    plt.colorbar(contour, ax=ax_contour, label='f(x, T)')
    ax_contour.set_xlabel('X')
    ax_contour.set_ylabel('Y')
    ax_contour.set_title('Contour Plot of f(x, T)')

    plt.tight_layout()
    if args.show:
        plt.show()
    else:
        plt.savefig("fig/wire-contour.pdf")

if __name__ == "__main__":
    df = pd.read_csv("data/T.csv")
    T = df.values
    plot_wireframe_and_contour(week6.f, T)  # Call the function to plot wireframe and contour

```

```python
import numpy as np

def iterate(self):
    self._x_value = self._start
    self._old_x_value = None
    self._f_star = 0
    self._iteration = 0
    self._converged_value = False
    self._grad_value = self._gradient(self._x_value)

    yield self.state_dict()

    while not self._converged_value:
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        numerator = self._function(self._x_value) - self._f_star
        self._grad_value = self._gradient(self._x_value)
        denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
        self._old_x_value = self._x_value
        step = numerator/denominator
        self._x_value = self._x_value - step * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        yield self.state_dict()
```

```python
import sgd
import week6
import pandas as pd
import numpy as np

if __name__ == "__main__":
    T = pd.read_csv("data/T.csv").values
    o = sgd.StochasticGradientDescent().alg("polyak")
    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
    o.function_generator(fg)
    o.start(np.array([3, 3]))
    for i in range(100):
        o.step()
        print("grad", o._grad_value)
    print("polyak", o._x_value)
```

```python
def iterate(self):
    import numpy as np
    self._x_value = self._start
    old_x_value = None
    self._iteration = 0
    self._sum = np.zeros(self._x_value.shape)
    alpha_n = np.zeros(self._x_value.shape)
    alpha_n.fill(self._step_size)
    self._converged_value = False
    self._grad_value = self._gradient(self._x_value)
    self._step_coeff = self._step_size

    yield self.state_dict()

    while not self._converged_value:
        self._iteration += 1
        if self._max_iter > 0 and self._iteration > self._max_iter:
            break
        self._grad_value = self._gradient(self._x_value)
        old_x_value = self._x_value
        self._x_value = self._x_value - alpha_n * self._grad_value
        self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
        alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
        self._step_coeff = alpha_n
        self._converged_value = self._converged(self._x_value, old_x_value)
        yield self.state_dict()
```

```python
import numpy as np
import functools
import lib
import week6

class StochasticGradientDescent(lib.GradientDescent):
    def __init__(self):
        self._iteration = 0
        self._max_iter = 1000
        self._converged = lambda x1, x2: False
        self._epsilon = 0.0001
        self._f_star = 0
        self._debug = False
        self._beta = 0
        self._function_generator = None
        self._dimension = None
        self._algorithm = None
        self._function = None
        self._sum = None
        self._x_value = None
        self._old_x_value = None
        self._step_coeff = None
        self._converged_value = None
        self._grad_value = None
        self._m = None
        self._v = None
        self._adam_grad = None
        self._beta = None
        self._beta2 = None
        self._step_size = None
        self._z = None

    def adam_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complet step
        self._grad_value = self._gradient(self._x_value)
        self._m = self._beta * self._m + (1-self._beta)*self._grad_value
        # grad_value * grad_value gives element-wise product of np array
        self._v = self._beta2 * self._v + (1-self._beta2) * (self._grad_value*self._grad_value)
        self._old_x_value = self._x_value
        self._iteration += 1
        m_hat = self._m / (1-(self._beta ** self._iteration))
        v_hat = np.array(self._v / (1-(self._beta2 ** self._iteration)))
        v_hat_aug = v_hat**(0.5) + self._epsilon
        self._adam_grad = m_hat / v_hat_aug
        self._x_value = self._x_value - self._step_size * self._adam_grad
        return True

    def polyak_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complet step
        self._iteration += 1
        numerator = self._function(self._x_value) - self._f_star
        self._grad_value = self._gradient(self._x_value)
        denominator = np.dot(self._grad_value, self._grad_value) # sum of element-wise products
        if denominator == 0.0:
            # do nothing this step (hope for non-zero on next mini-batch)
            return False
        self._old_x_value = self._x_value
        step = numerator/denominator
        self._x_value = self._x_value - step * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        return True  # completed step

    def constant_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False  # did not complete step
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._x_value = self._x_value - self._step_size * self._grad_value
        self._converged_value = self._converged(self._x_value, self._old_x_value)
        return True  # completed step

    def rmsprop_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._x_value = self._x_value - self._alpha_n * self._grad_value
        self._sum = self._beta * self._sum + (1-self._beta) * (self._grad_value**2)
        self._alpha_n = self._step_size / (self._sum**0.5+self._epsilon)
        self._step_coeff = self._alpha_n
        return True


    def heavy_ball_step(self):
        self._function, self._gradient = next(self._function_generator)
        if self._function == "finished":
            return False
        self._iteration += 1
        self._grad_value = self._gradient(self._x_value)
        self._old_x_value = self._x_value
        self._z = self._beta * self._z + self._step_size * self._grad_value
        self._x_value = self._x_value - self._z
```

```python
        return True

    # pass a function which generates the function to be evaluated,
    # e.g. with different minibatches at each iteration
    def function_generator(self, fg):
        self._function_generator = fg
        return self

    def alg(self, a):
        if a == "constant":
            self.step = self.constant_step
        elif a == "polyak":
            self.step = self.polyak_step
        elif a == "rmsprop":
            self.step = self.rmsprop_step
            if self._step_size is None:
                raise Exception("Need step_size to initialize rmsprop")
            if self._x_value is None:
                raise Exception("Need start/x_value to initialize rmsprop")
            self._sum = np.zeros(self._x_value.shape)
            self._alpha_n = np.zeros(self._x_value.shape)
            self._alpha_n.fill(self._step_size)
        elif a == "adam":
            self.step = self.adam_step
            if self._x_value is None:
                raise Exception("Need start/x_value to initialize rmsprop")
            self._m = np.zeros(self._x_value.shape, dtype=np.float64)
            self._v = np.zeros(self._x_value.shape, dtype=np.float64)
        elif a == "heavy_ball":
            self.step = self.heavy_ball_step
            self._z = 0
        else:
            raise Exception(f"Alg {a} NYI")
        self.function_name = a
        return self

    def polyak_init(self):
        self._x_value = self._start
        self._old_x_value = None
        self._f_star = 0
        self._iteration = 0
        self._converged_value = False
        self._grad_value = self._gradient(self._x_value)

```

```python
import sgd
import week6
import pandas as pd
import numpy as np

if __name__ == "__main__":
    T = pd.read_csv("data/T.csv").values

    o = sgd.StochasticGradientDescent().alg("constant")
    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
    o.function_generator(fg)
    o.step_size(0.01)
    o.start(np.array([3, 3]))
    for i in range(100):
        o.step()
    print("constant", o._x_value)

    o = sgd.StochasticGradientDescent().alg("polyak")
    fg = week6.generate_optimisation_functions(T, minibatch_size=10, shuffle=False)
    o.function_generator(fg)
    o.start(np.array([0.9, 0.9]))
    for i in range(100):
        o.step()
    print("polyak", o._x_value)

    o = sgd.StochasticGradientDescent().alg("polyak")
    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
    o.function_generator(fg)
    o.start(np.array([3, 3]))
    for i in range(100):
        o.step()
    print("polyak", o._x_value)

    o = sgd.StochasticGradientDescent()
    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
    o.function_generator(fg)
    o.start(np.array([3, 3]))
    o.step_size(0.00001)
    o.beta(0.99)
    o.alg("rmsprop")
    for i in range(100):
        o.step()
    print("rmsprop", o._x_value)

    o = sgd.StochasticGradientDescent()
    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
    o.function_generator(fg)
    o.start(np.array([3, 3]))
    o.step_size(0.00001)
    o.beta(0.99)
    o.alg("heavy_ball")
    for i in range(100):
        o.step()
    print("heavy_ball", o._x_value)

    o = sgd.StochasticGradientDescent()
    fg = week6.generate_optimisation_functions(T, minibatch_size=5)
    o.function_generator(fg)
    o.start(np.array([3, 3]))
    o.step_size(0.00001)
    o.beta(0.99)
    o.beta2(0.25)
    o.alg("adam")
    for i in range(100):
        o.step()
    print("adam", o._x_value)
```

```python
import week6
import pandas as pd
import sympy as sp

T = pd.read_csv("data/T.csv").values

sympy_loss = week6.sympy_loss(T)

x1, x2 = sp.symbols('x1 x2', real=True)

dydx1 = sp.diff(sympy_loss, x1)
dydx2 = sp.diff(sympy_loss, x2)

solutions = sp.solve([dydx1, dydx2], (x1, x2), simplify=False, rational=False)

print(solutions)
```

```python
import numpy as np
import pandas as pd
import week6
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm

# Global variables for extents
x_min, x_max = -5, 5
y_min, y_max = -5, 5

def plot_wireframe_and_contour(f, T, resolution=100):
    global x_min, x_max, y_min, y_max

    # Generate data for wireframe plot
    x_range = np.linspace(x_min, x_max, resolution)
    y_range = np.linspace(y_min, y_max, resolution)
    X, Y = np.meshgrid(x_range, y_range)
    Z = np.zeros_like(X)
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f([X[i, j], Y[i, j]], T)

    # Plot wireframe
    fig = plt.figure(figsize=(12, 6))

    ax_wireframe = fig.add_subplot(121, projection='3d')
    ax_wireframe.plot_wireframe(X, Y, Z, color='blue')
    ax_wireframe.set_xlabel('X')
    ax_wireframe.set_ylabel('Y')
    ax_wireframe.set_zlabel('f(x, T)')
    ax_wireframe.set_title('Wireframe Plot of f(x, T)')

    # Generate data for contour plot
    Z_contour = np.zeros_like(X)
    for i in range(resolution):
        for j in range(resolution):
            Z_contour[i, j] = f([X[i, j], Y[i, j]], T)

    # Plot contour with log scale color
    ax_contour = fig.add_subplot(122)
    contour = ax_contour.contourf(X, Y, Z_contour, levels=20, norm=LogNorm(), cmap='viridis')
    plt.colorbar(contour, ax=ax_contour, label='f(x, T)')
    ax_contour.set_xlabel('X')
    ax_contour.set_ylabel('Y')
    ax_contour.set_title('Contour Plot of f(x, T)')

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    df = pd.read_csv("data/T.csv")
    T = df.values
    plot_wireframe_and_contour(week6.f, T)  # Call the function to plot wireframe and contour
```

```python
import numpy as np
import pandas as pd
import week6
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Global variables for extents
x_min, x_max = -5, 5
y_min, y_max = -5, 5

def plot_wireframe_and_contour(f, T, resolution=100):
    global x_min, x_max, y_min, y_max

    # Generate data for wireframe plot
    x_range = np.linspace(x_min, x_max, resolution)
    y_range = np.linspace(y_min, y_max, resolution)
    X, Y = np.meshgrid(x_range, y_range)
    Z = np.zeros_like(X)
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f([X[i, j], Y[i, j]], T)

    # Plot wireframe
    fig = plt.figure(figsize=(12, 6))

    ax_wireframe = fig.add_subplot(121, projection='3d')
    ax_wireframe.plot_wireframe(X, Y, Z, color='blue')
    ax_wireframe.set_xlabel('X')
    ax_wireframe.set_ylabel('Y')
    ax_wireframe.set_zlabel('f(x, T)')
    ax_wireframe.set_title('Wireframe Plot of f(x, T)')

    # Generate data for contour plot
    Z_contour = np.zeros_like(X)
    for i in range(resolution):
        for j in range(resolution):
            Z_contour[i, j] = f([X[i, j], Y[i, j]], T)

    # Plot contour
    ax_contour = fig.add_subplot(122)
    contour = ax_contour.contourf(X, Y, Z_contour, levels=20, cmap='viridis')
    plt.colorbar(contour, ax=ax_contour, label='f(x, T)')
    ax_contour.set_xlabel('X')
    ax_contour.set_ylabel('Y')
    ax_contour.set_title('Contour Plot of f(x, T)')

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    df = pd.read_csv("data/T.csv")
    T = df.values
    plot_wireframe_and_contour(week6.f, T)  # Call the function to plot wireframe and contour
```

```python
import numpy as np
import sympy as sp

current_minibatch = None

def generate_trainingdata(m=25):
    return np.array([0,0]) + 0.25 * np.random.randn(m,2)


def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y = 0
    count = 0
    for w in minibatch:
        z = x - w - 1
        left = 10 * (z[0]**2+z[1]**2)
        right = (z[0]+2)**2+(z[1]+4)**2
        y = y + min(left, right)
        count = count + 1
    return y/count


def gradient_function_fd(minibatch, epsilon=10**(-15)):
    def gradient_fd(x):
        dydx1 = (f(x + np.array([epsilon, 0]), minibatch) - f(x, minibatch)) / epsilon
        dydx2 = (f(x + np.array([0, epsilon]), minibatch) - f(x, minibatch)) / epsilon
        return np.array([dydx1, dydx2])
    return gradient_fd

def sympy_loss(minibatch):
    x1, x2 = sp.symbols('x1 x2', real=True)
    function = 0
    for w in minibatch:
        z1 = x1 - w[0] - 1
        z2 = x2 - w[1] - 1
        left = 10 * (z1**2 + z2**2)
        right = (z1 + 2)**2 + (z2 + 4)**2
        function = sp.Min(left, right) + function
    function = function / len(minibatch)
    return function

def gradient_function(minibatch):
    function = sympy_loss(minibatch)
    def gradient(x):
        dydx1 = function.diff(x1)
        dydx2 = function.diff(x2)
        return np.array([
            dydx1.subs(x1, x[0]).subs(x2, x[1]),
            dydx2.subs(x1, x[0]).subs(x2, x[1]),
        ])

    return gradient


def loss(x, w):
    z = x - w - 1
    left = 10 * (z[0]**2+z[1]**2)
    right = (z[0]+2)**2+(z[1]+4)**2
    return min(left, right)


def f_clear(x, minibatch):
    return sum(loss(x, w) for w in minibatch) / len(minibatch)


def generate_minibatches(T, N=5, seed=42, shuffle=True):
    global current_minibatch
    if shuffle:
        T = T.copy()
        np.random.seed(seed)
        np.random.shuffle(T)
    num_rows = T.shape[0]
    i = 0

    minibatch = np.zeros((N, T.shape[1]), T.dtype)
    while True:
        for j in range(N):
            minibatch[j] = T[i % num_rows]
            i += 1
            if shuffle and i >= num_rows:
                # begin next epoch
                np.random.shuffle(T)
                i = 0
        current_minibatch = minibatch
        yield minibatch


def generate_optimisation_functions(batch, minibatch_size=5, finite_difference=True, **kwargs):
    minibatch_generator = generate_minibatches(
        batch, N=minibatch_size, **kwargs)
    for minibatch in minibatch_generator:
        def optim_func(x):
            return f_clear(x, minibatch)
        gradf = None
        if finite_difference:
            gradf = gradient_function_fd(minibatch)
        else:
            gradf = gradient_function(minibatch)
        yield (optim_func, gradf)
    yield "finished"
```

```python


if __name__ == "__main__":
    import os
    os.makedirs("data", exist_ok=True)
    T = generate_trainingdata()
    import pandas as pd
    df = pd.DataFrame(T)
    df.to_csv("data/T.csv", index=False)

    x = np.array([3, 3])
```

```python
import week6
import numpy as np

if __name__ == "__main__":
    T = week6.generate_trainingdata()
    import pandas as pd
    df = pd.read_csv("data/T.csv")
    T = df.values

    x = np.array([3, 3])
    print(week6.f(x, T) - week6.f_clear(x, T))

    generator = week6.generate_minibatches(T, N=2, shuffle=False)
    for i in range(3):
        n = next(generator)
        print(len(n), n)

    fgen = week6.generate_optimisation_functions(T, minibatch_size=5)
    zipped = zip(range(10), fgen)
    for (i, f) in zipped:
        print(f[0](x), f[1](x))
```

```python
import numpy as np
minibatch = np.array([
    [0.0918635, -0.0468714],
    [-0.66994666, -0.133955],
    [-0.08386569, 0.3052427],
    [-0.00564624, -0.12876412],
    [-0.38826176, 0.23831869]
])

x = [0.80697696, 1.05286489]

import week6
print(minibatch)
print("gradient:", week6.gradient_function_fd(minibatch)(x))
```