Module 5.6 : Stochastic And Mini-Batch Gradient Descent

Let's digress a bit and talk about the stochastic version of these algorithms...

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why?

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation.

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside?

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside? Imagine we have a million points in the training data.

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside? Imagine we have a million points in the training data. To make 1 update to w, b the algorithm makes a million calculations.

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside? Imagine we have a million points in the training data. To make 1 update to w, b the algorithm makes a million calculations. Obviously very slow!!

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside? Imagine we have a million points in the training data. To make 1 update to w, b the algorithm makes a million calculations. Obviously very slow!!
- Can we do something better ?

```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside? Imagine we have a million points in the training data. To make 1 update to w, b the algorithm makes a million calculations. Obviously very slow!!
- Can we do something better ? Yes, let's look at stochastic gradient descent

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

```python
def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

```python
def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm updates the parameters for every single data point

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

```python
def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ?

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ? It is an approximate (rather stochastic) gradient

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- Stochastic because we are estimating the total gradient based on a single data point.

- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ? It is an approximate (rather stochastic) gradient

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- Stochastic because we are estimating the total gradient based on a single data point. Almost like tossing a coin only once and estimating P(heads).

- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ? It is an approximate (rather stochastic) gradient

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

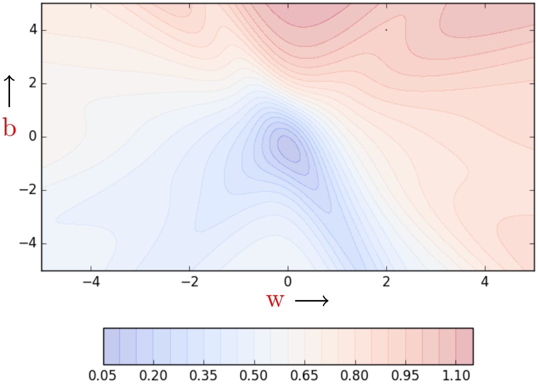
- Stochastic because we are estimating the total gradient based on a single data point. Almost like tossing a coin only once and estimating P(heads).

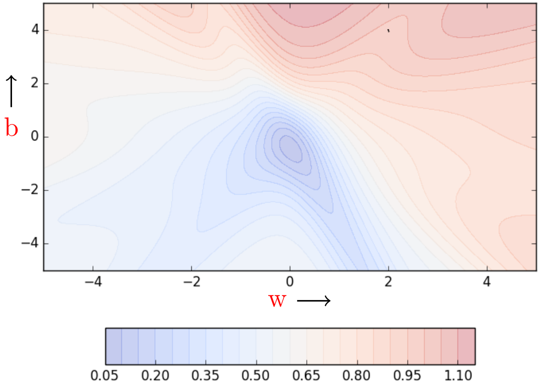
- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ? It is an approximate (rather stochastic) gradient
- No guarantee that each step will decrease the loss

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

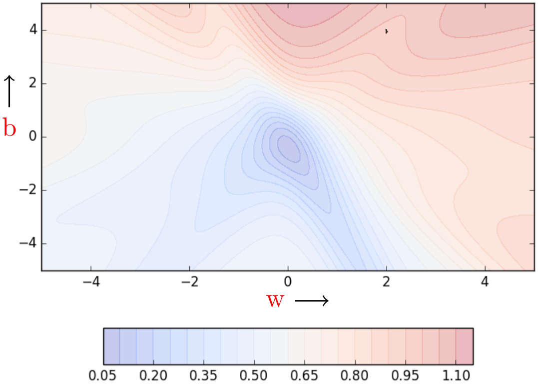
- Stochastic because we are estimating the total gradient based on a single data point. Almost like tossing a coin only once and estimating P(heads).

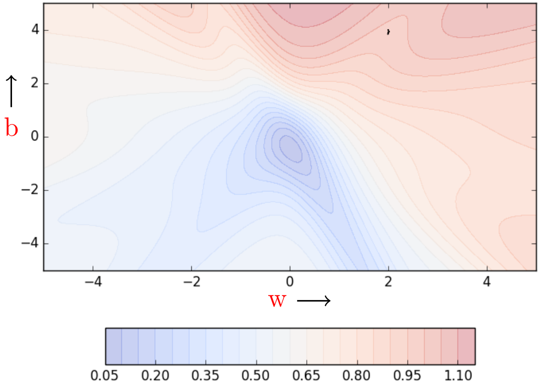
- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ? It is an approximate (rather stochastic) gradient
- No guarantee that each step will decrease the loss
- Let's see this algorithm in action when we have a few data points

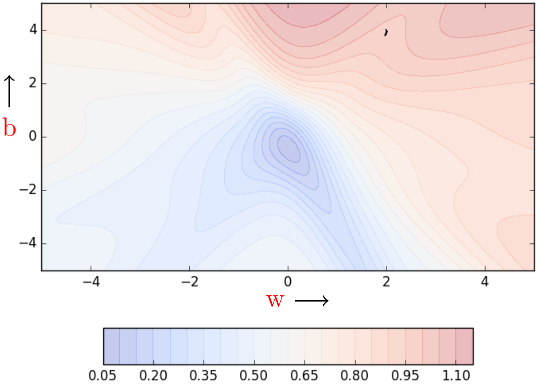

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



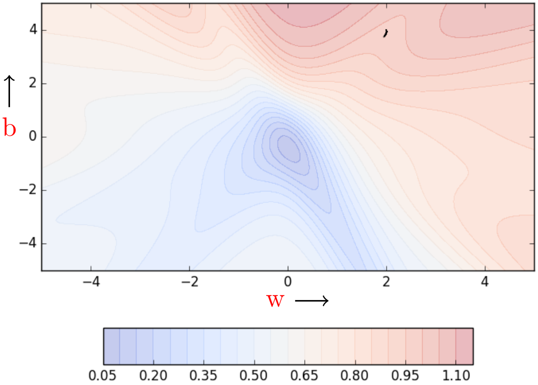
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


b
w
4
2
0
−2
−4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

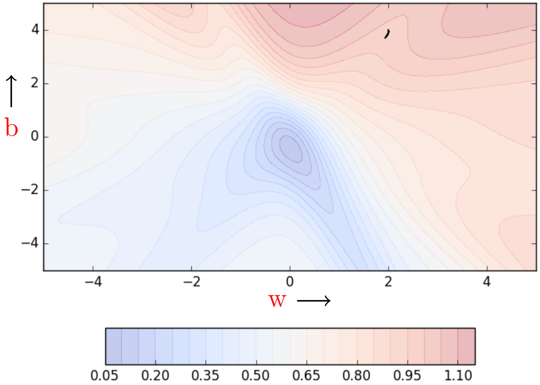

b
w
4
2
0
−2
−4
−4
−2
0
2
4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

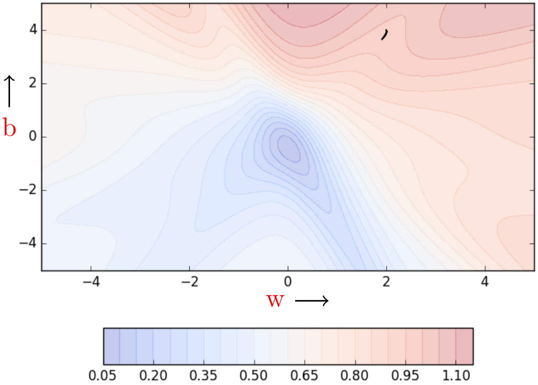


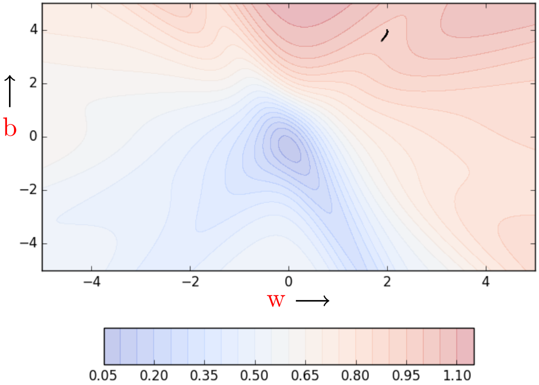
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



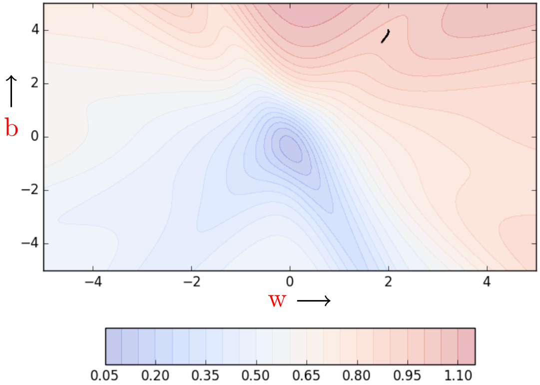
b
w
4
2
0
−2
−4
−4
−2
0
2
4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

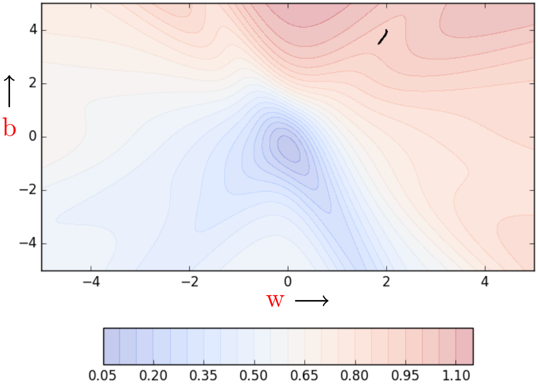

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

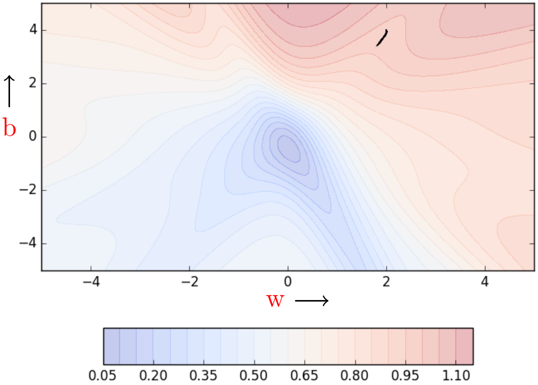

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

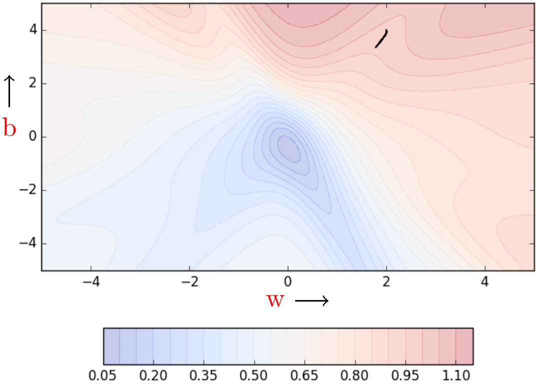


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

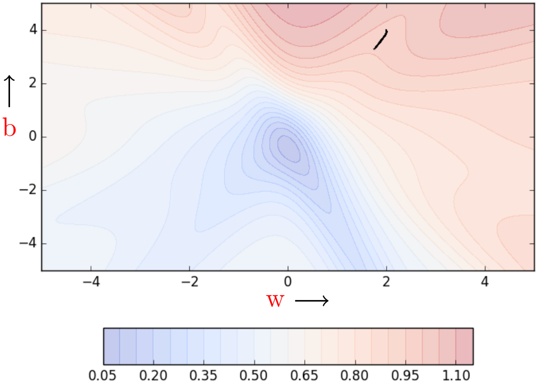

4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

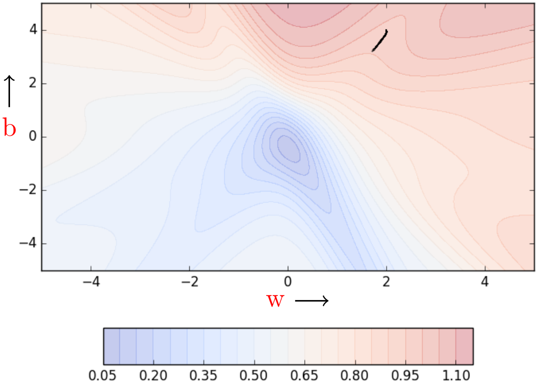

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

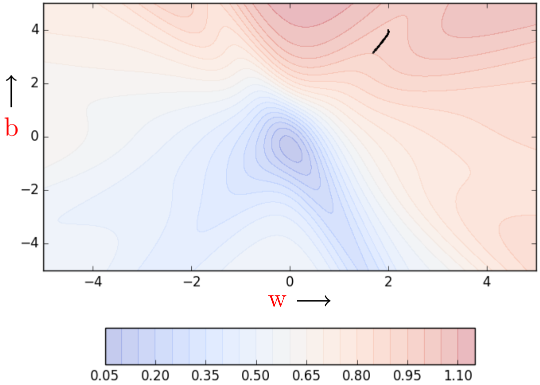

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

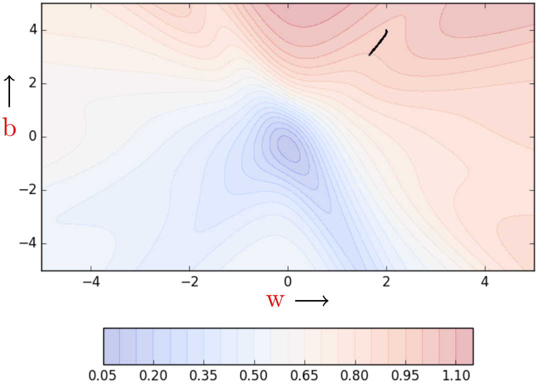


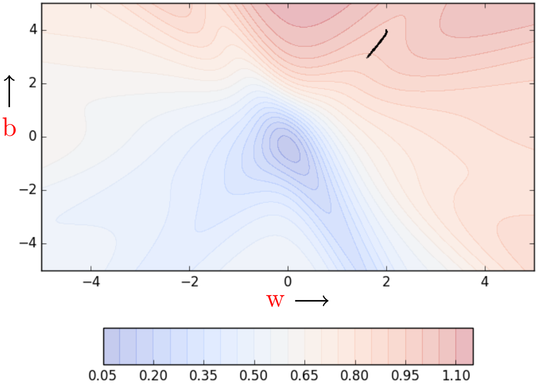
b
w
4
2
0
−2
−4
−4
−2
0
2
4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

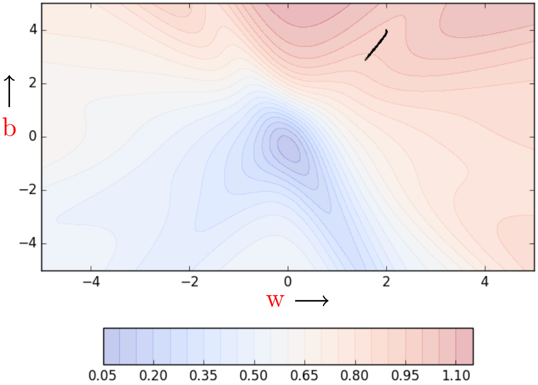

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

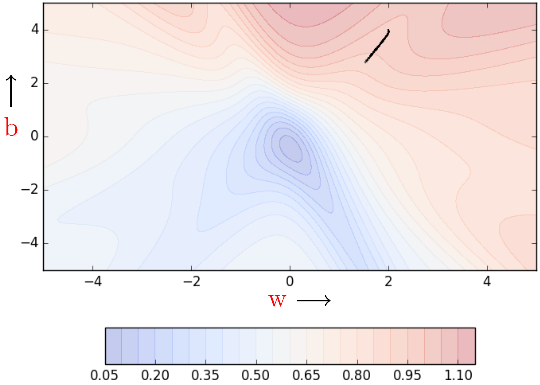

b
w
4
2
0
−2
−4
−4
−2
0
2
4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

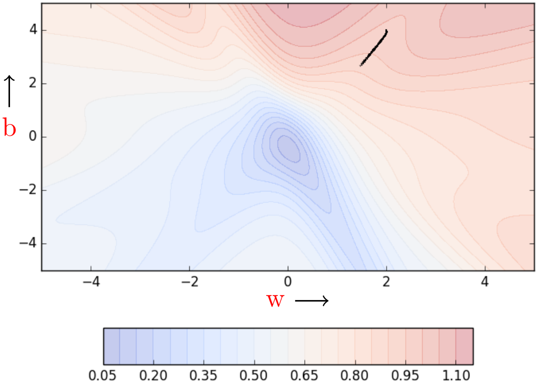

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

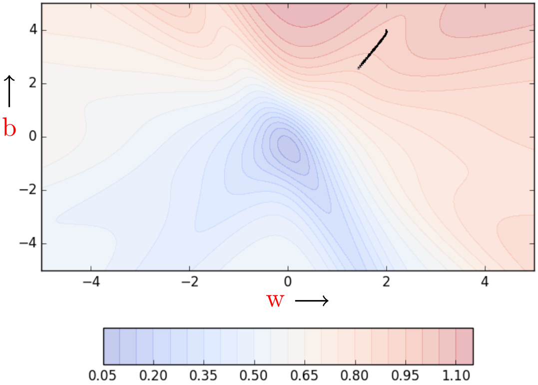

4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



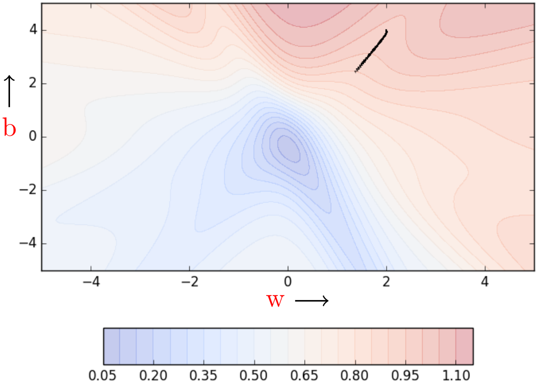
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



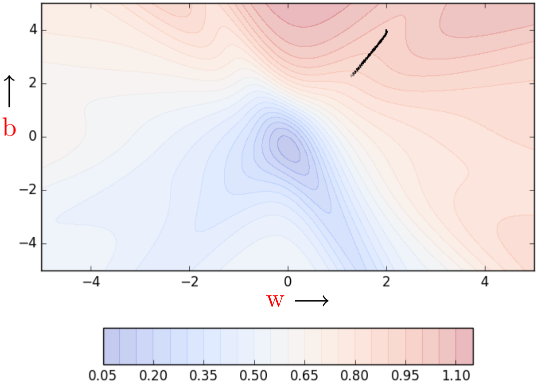
4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

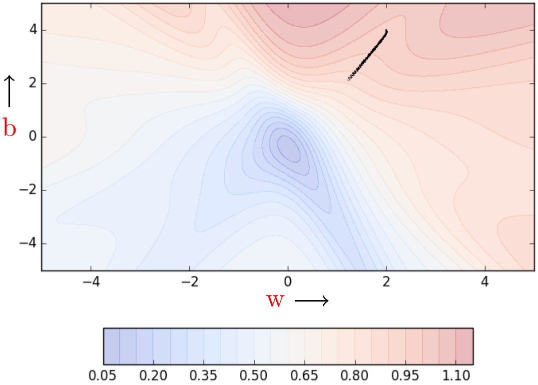


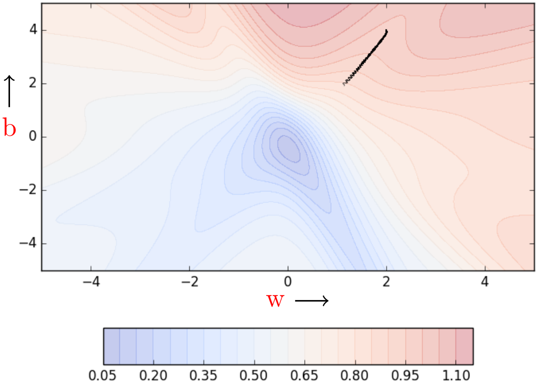
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



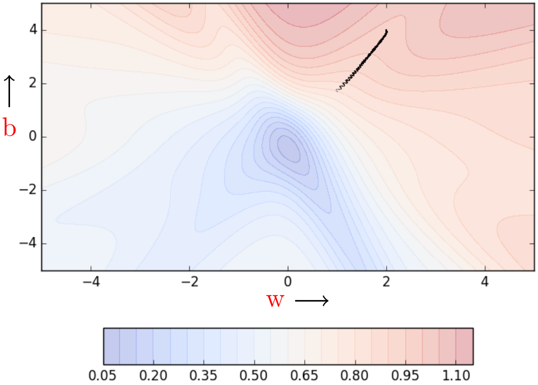
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

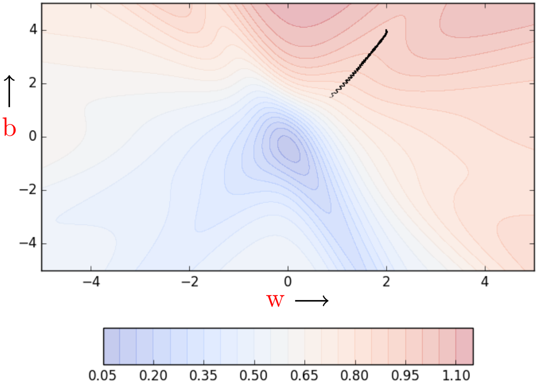

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

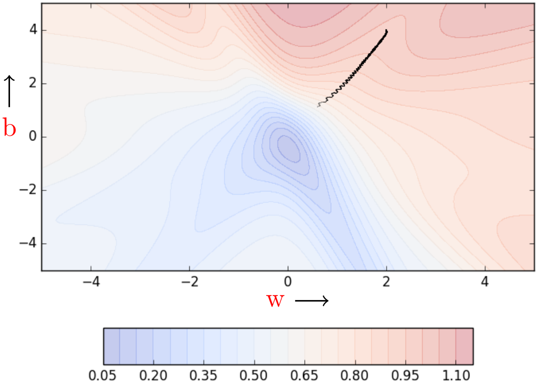

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

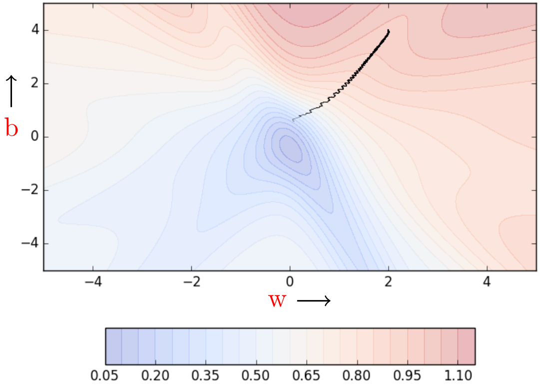


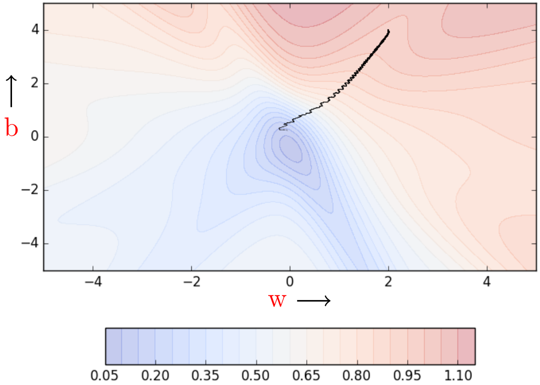
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

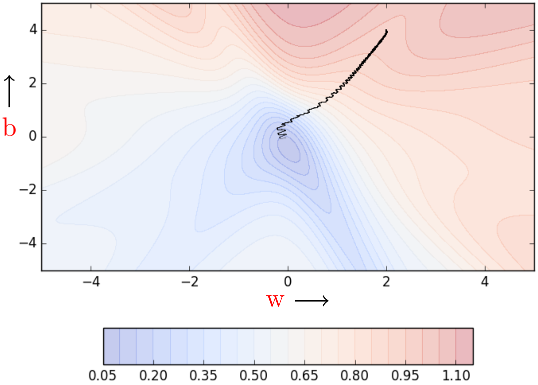

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

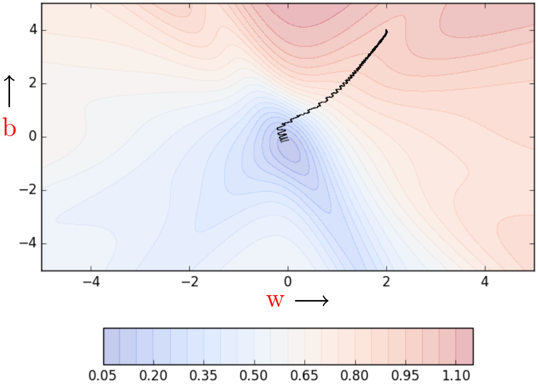

4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

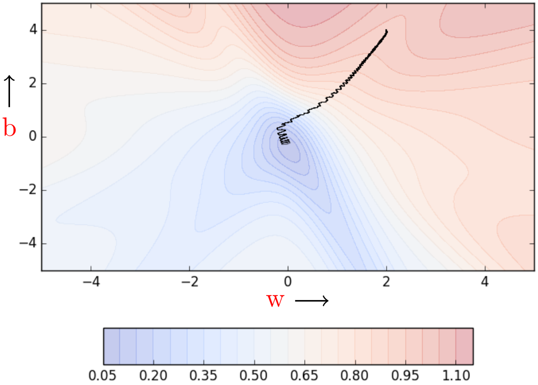

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

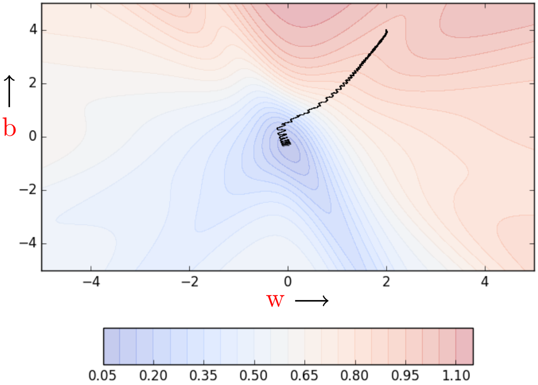

b
w
4
2
0
−2
−4
−4
−2
0
2
4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

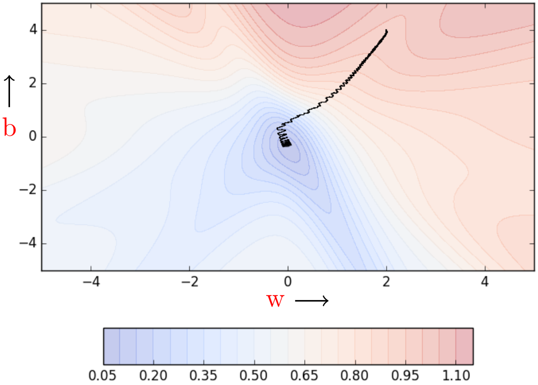

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

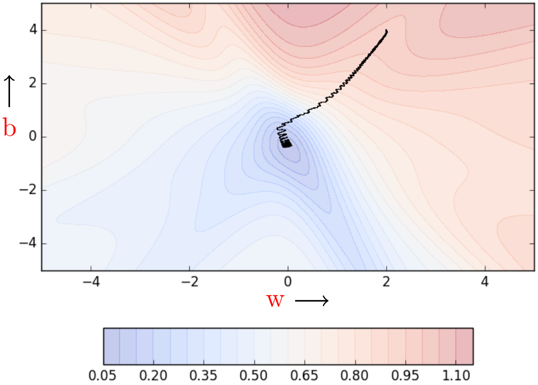

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

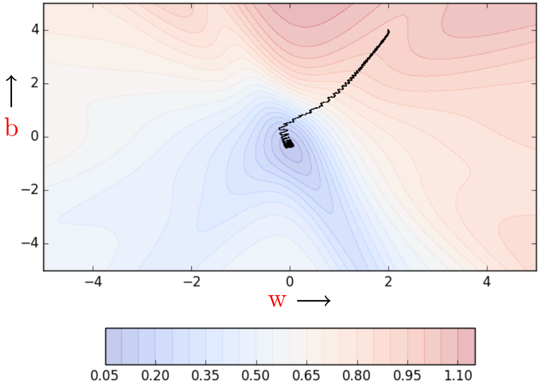

4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

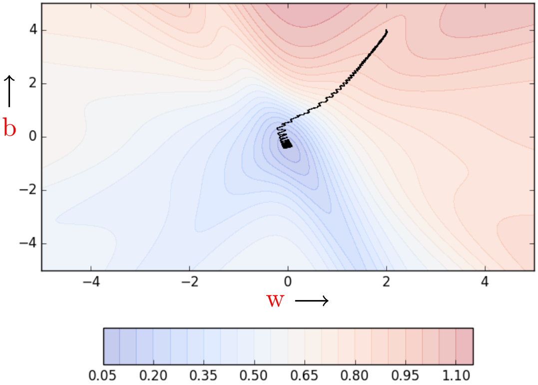
- We see many oscillations. Why ?



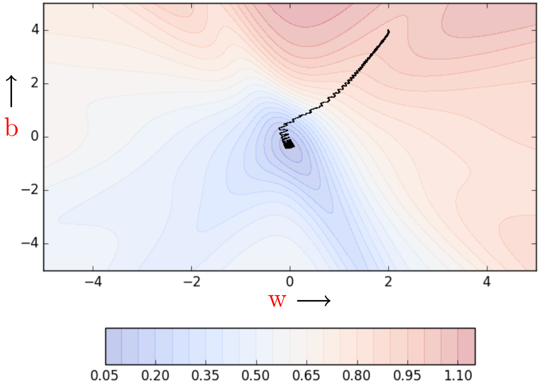
- We see many oscillations. Why ? Because we are making greedy decisions.



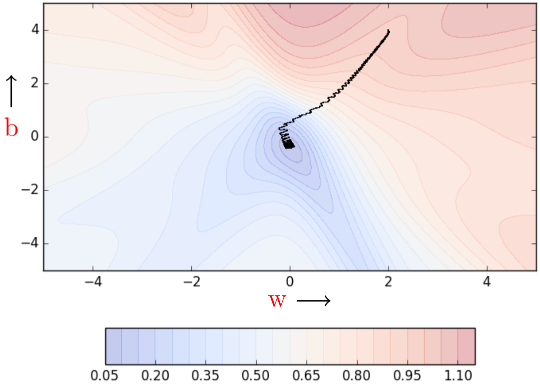
- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it



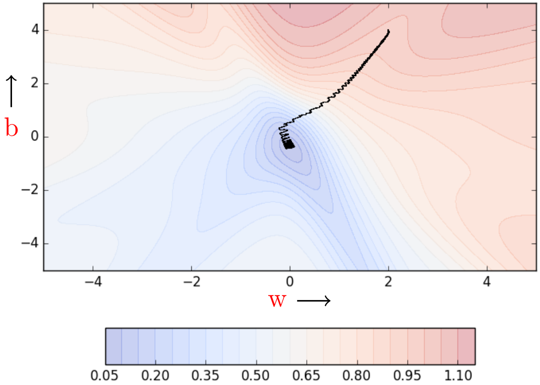
- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)



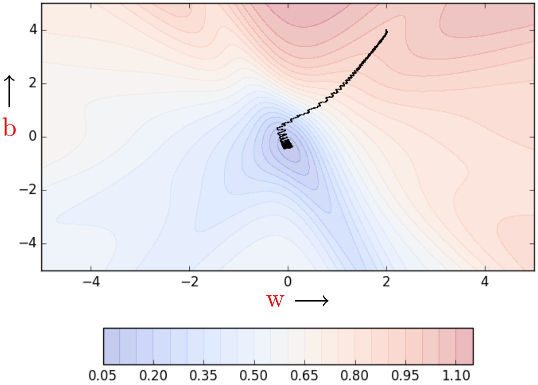
- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)
- A parameter update which is locally favorable to one point may harm other points



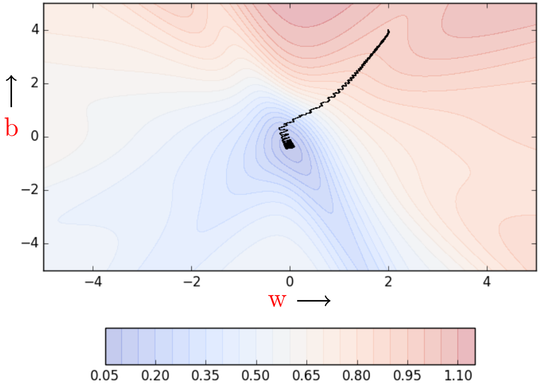
- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)



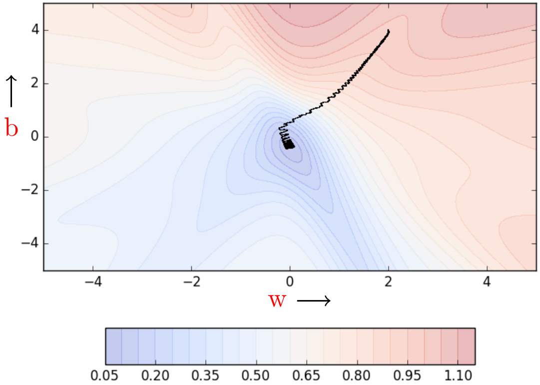
- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)
- Can we reduce the oscillations by improving our stochastic estimates of the gradient



- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)
- Can we reduce the oscillations by improving our stochastic estimates of the gradient (currently estimated from just 1 data point at a time)



- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)
- Can we reduce the oscillations by improving our stochastic estimates of the gradient (currently estimated from just 1 data point at a time)
- Yes, let's look at mini-batch gradient descent



```python
def do_mini_batch_gradient_descent() :
    w, b, eta =-2, -2, 1.0
    mini_batch_size, num_points_seen = 2, 0
    for i in range(max_epochs) :
        dw, db, num_points = 0, 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen +=1

            if num_points_seen % mini_batch_size == 0 :
                # seen one mini_batch
                w = w - eta * dw
                b = b - eta * db
                dw, db = 0, 0 #reset gradients
```

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- Notice that the algorithm updates the parameters after it sees *mini_batch_size* number of data points

```python
def do_mini_batch_gradient_descent() :
    w, b, eta =-2, -2, 1.0
    mini_batch_size, num_points_seen = 2, 0
    for i in range(max_epochs) :
        dw, db, num_points = 0, 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen +=1

            if num_points_seen % mini_batch_size == 0 :
                # seen one mini_batch
                w = w - eta * dw
                b = b - eta * db
                dw, db = 0, 0 #reset gradients
```

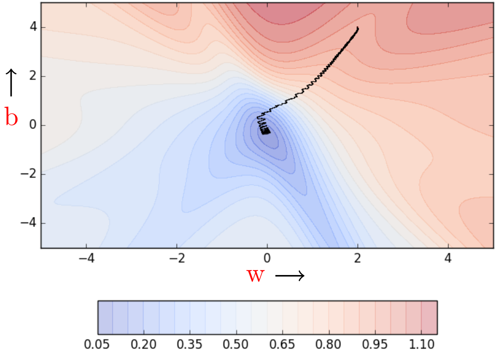
```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

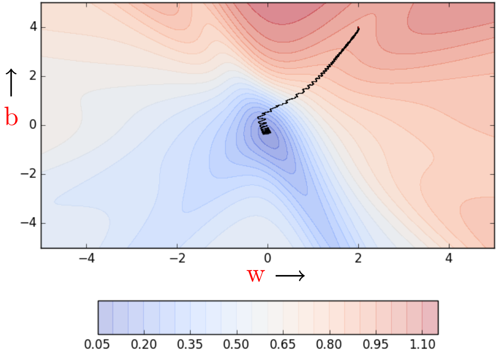
- Notice that the algorithm updates the parameters after it sees *mini_batch_size* number of data points
- The stochastic estimates are now slightly better

```python
def do_mini_batch_gradient_descent() :
    w, b, eta =-2, -2, 1.0
    mini_batch_size, num_points_seen = 2, 0
    for i in range(max_epochs) :
        dw, db, num_points = 0, 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen +=1

            if num_points_seen % mini_batch_size == 0 :
                # seen one mini_batch
                w = w - eta * dw
                b = b - eta * db
                dw, db = 0, 0 #reset gradients
```

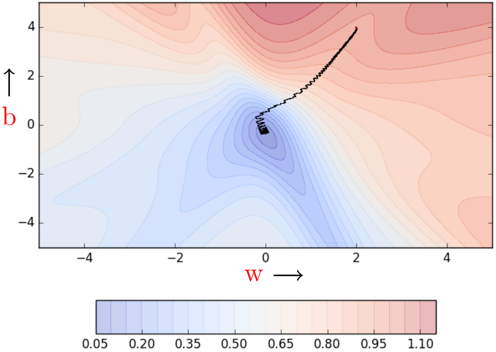
```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

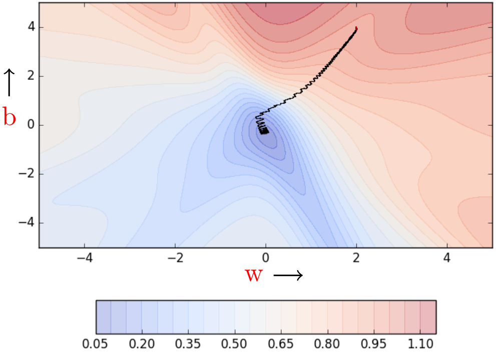
- Notice that the algorithm updates the parameters after it sees $mini\_batch\_size$ number of data points
- The stochastic estimates are now slightly better
- Let's see this algorithm in action when we have k = 2


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

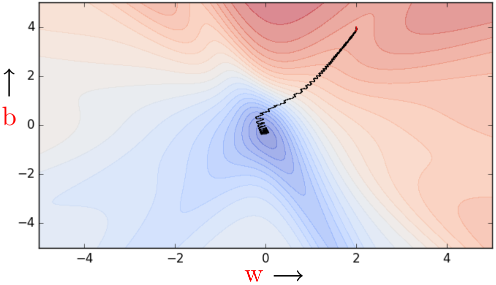


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

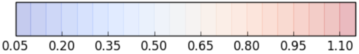

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



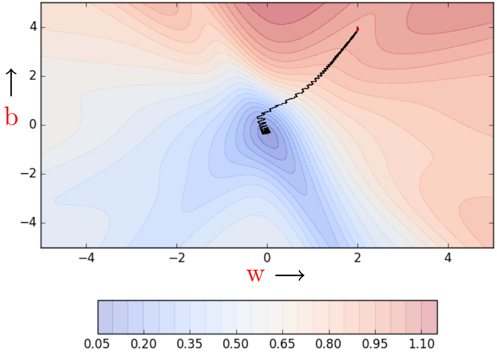
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

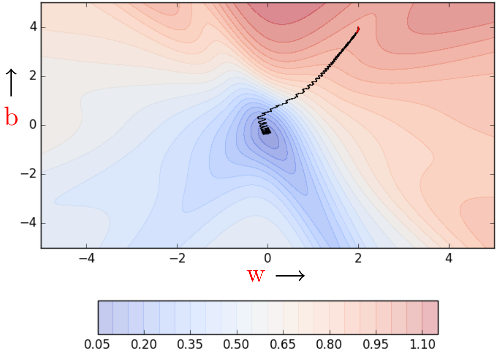


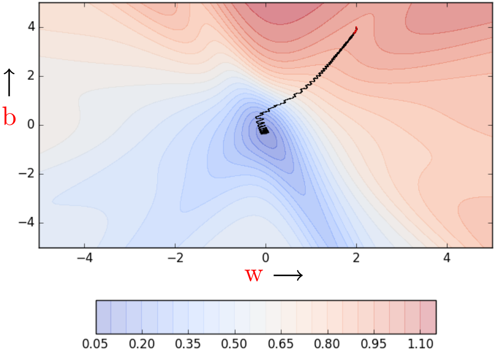
4
2
0
−2
−4
b
−4
−2
0
2
4
w

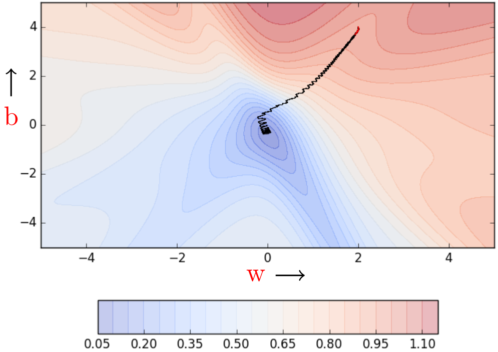


0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

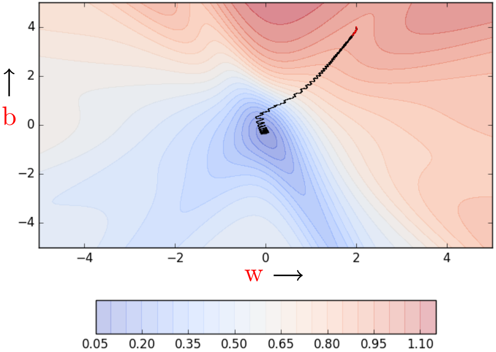

4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

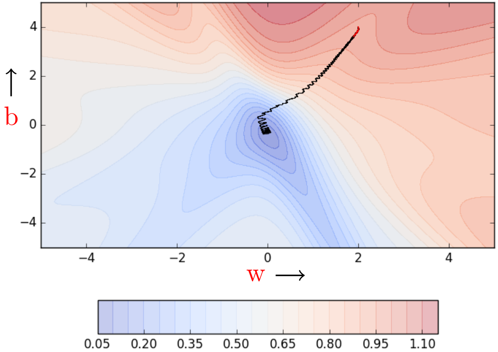

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

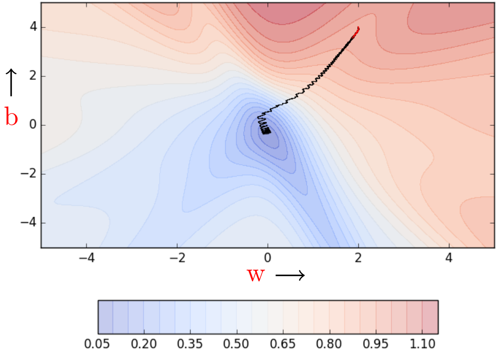


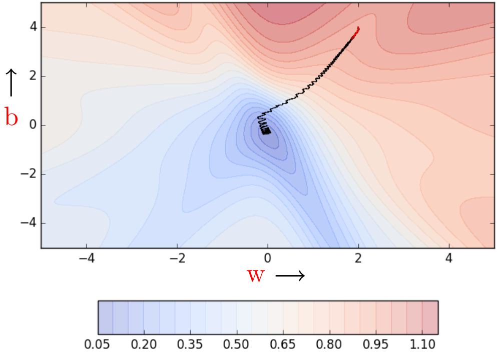
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

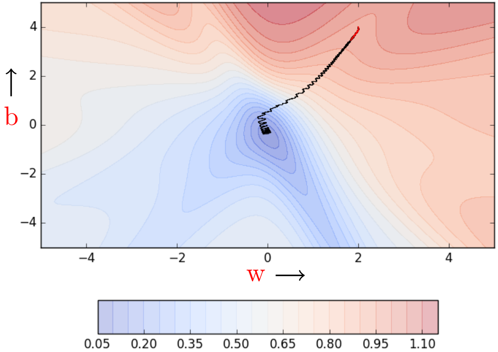


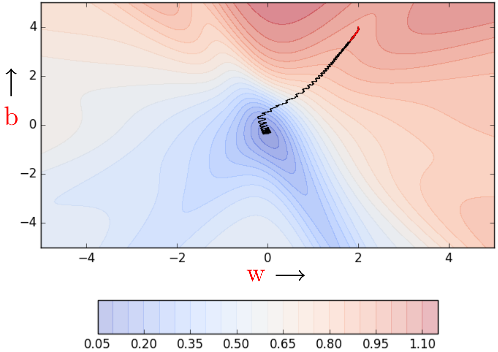
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

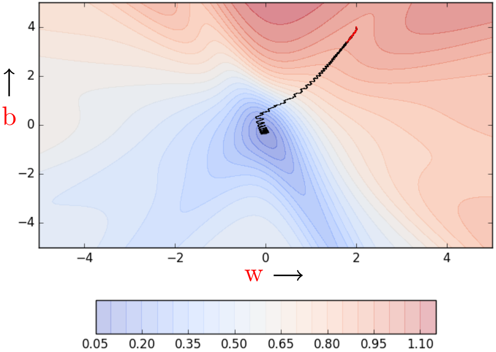

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



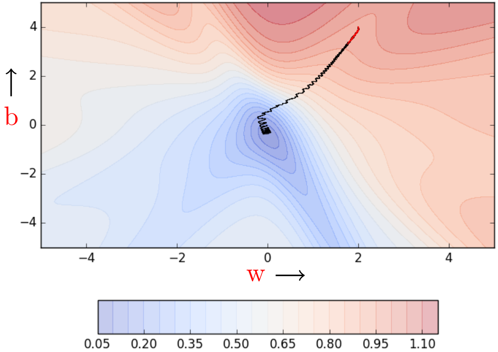
b
w
4
2
0
−2
−4
−4
−2
0
2
4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

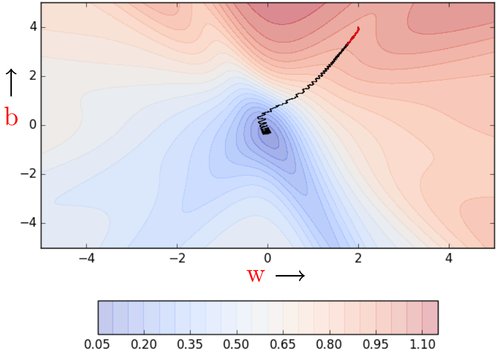

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



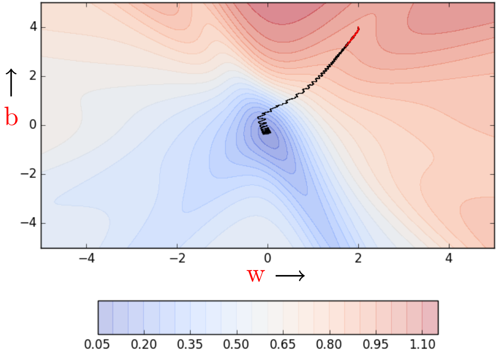
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

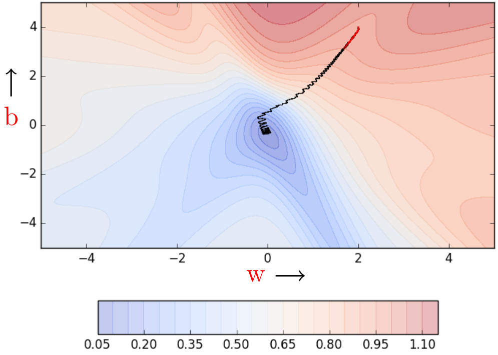


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

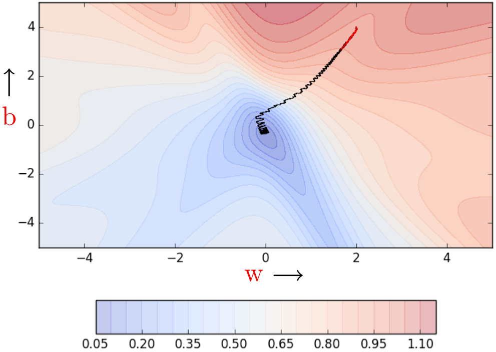


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

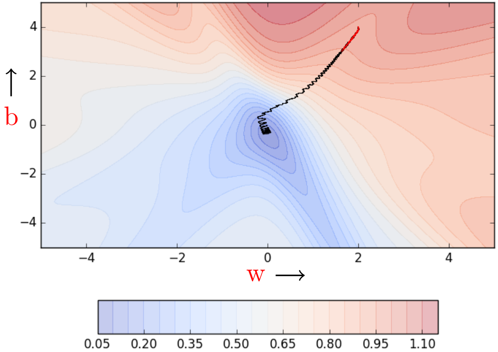

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



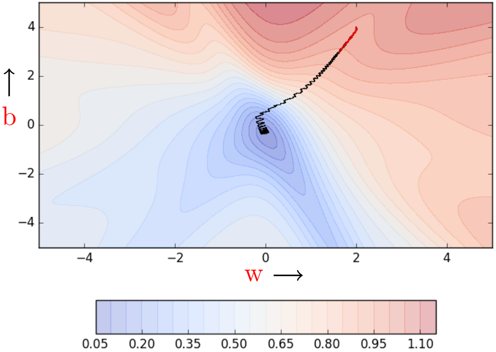
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



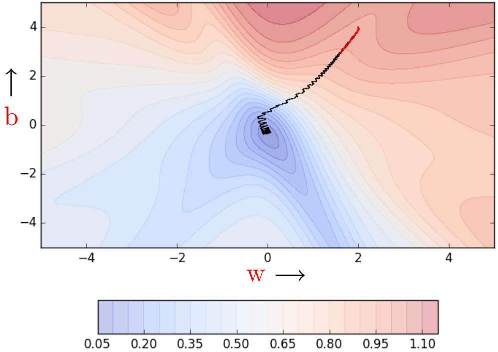
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



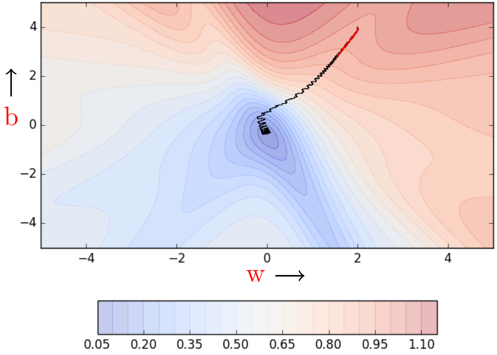
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



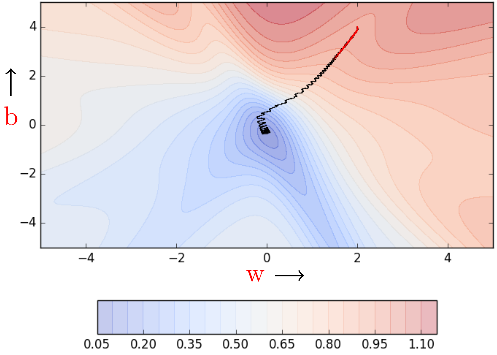
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



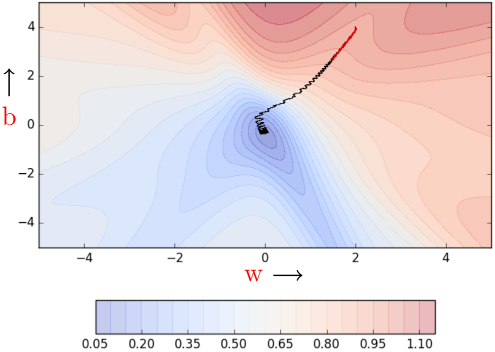
4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



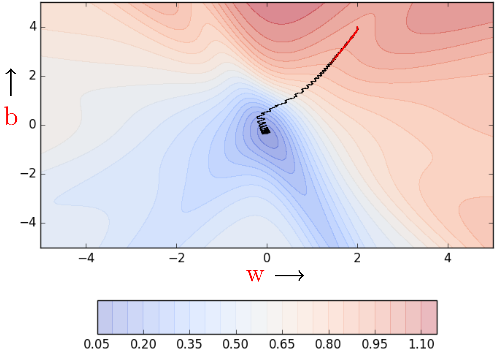
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



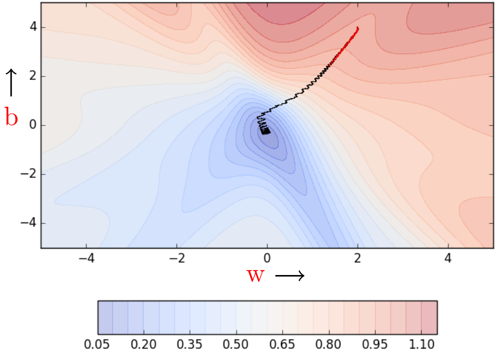
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

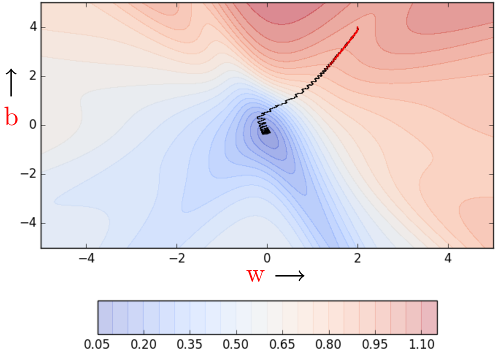


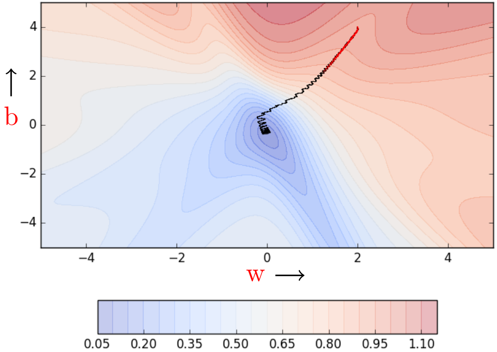
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



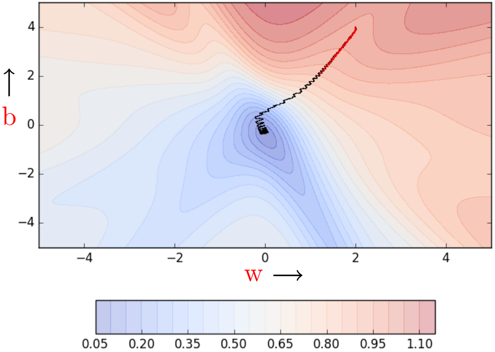
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

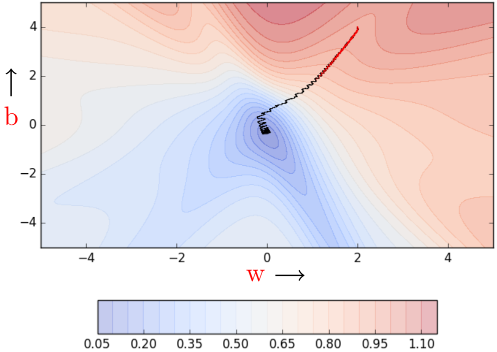

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

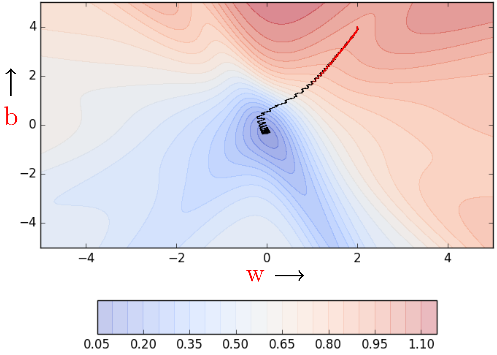


b
w
4
2
0
−2
−4
−4
−2
0
2
4
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

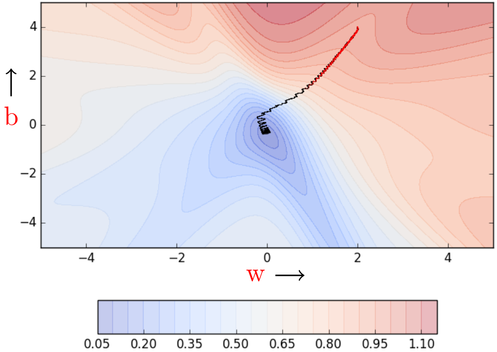

4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

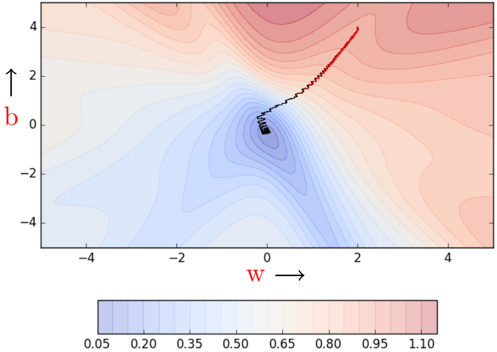


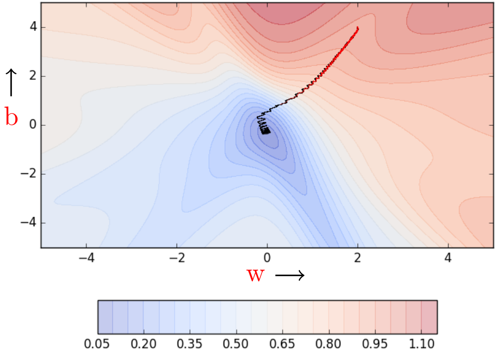
4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10



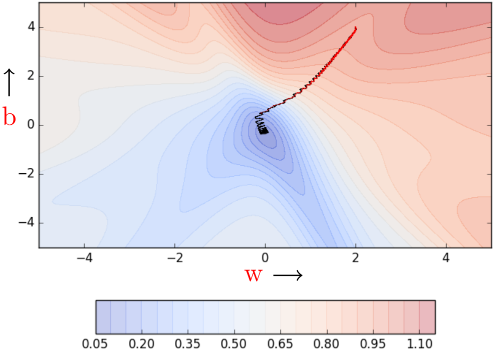
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

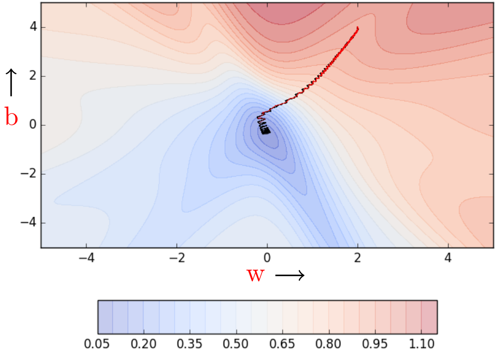

4
2
0
−2
−4
−4
−2
0
2
4
b
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

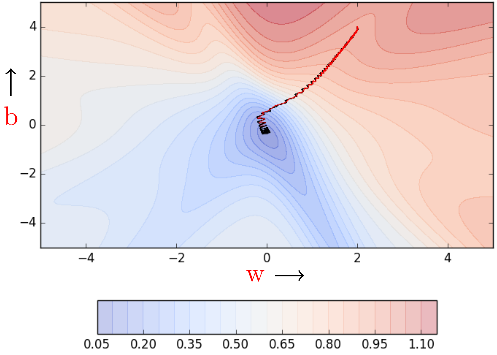


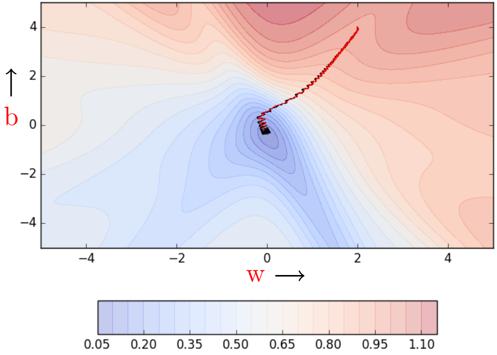
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

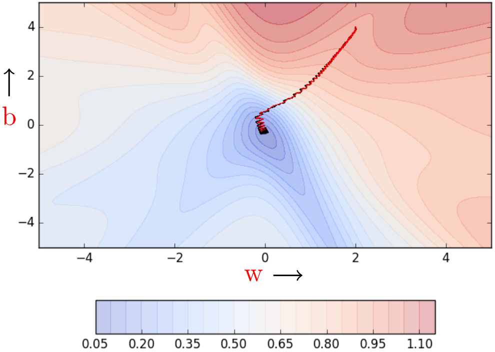


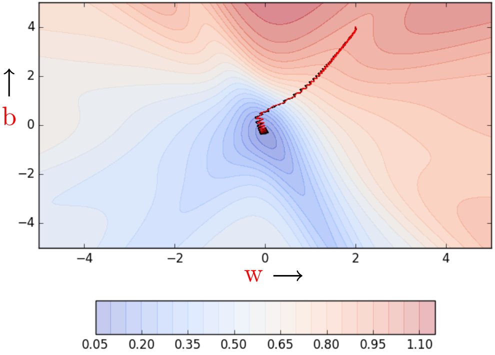
4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

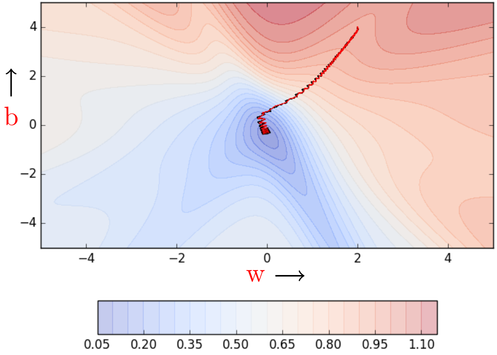

4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

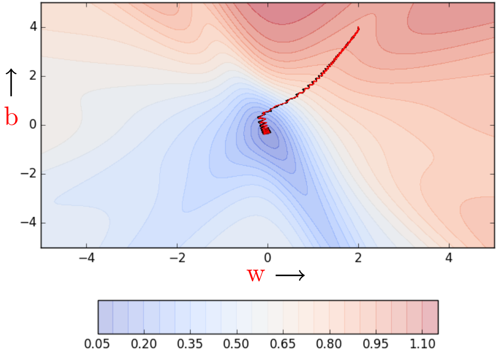


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

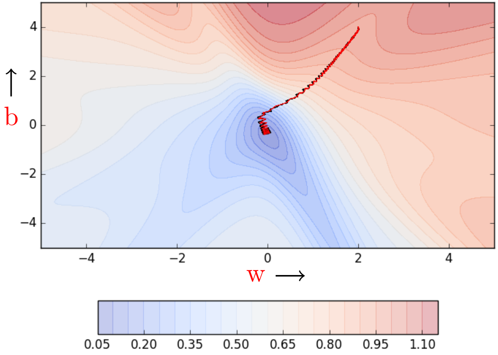


4
2
0
−2
−4
b
−4
−2
0
2
4
w
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

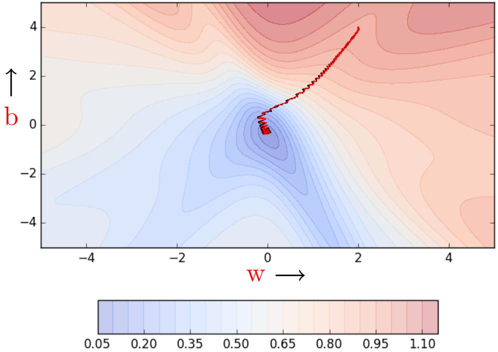
- Even with a batch size of k=2 the oscillations have reduced slightly.



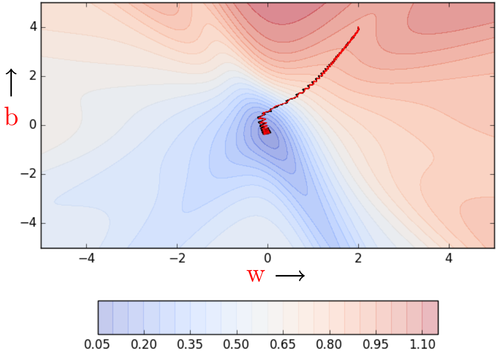
- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?



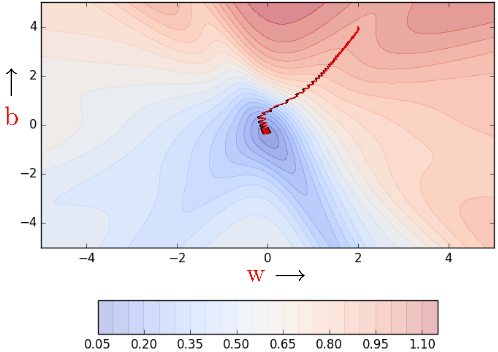
- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?
- Because we now have slightly better estimates of the gradient



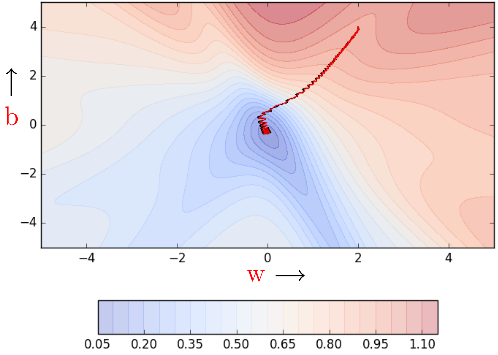
- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?
- Because we now have slightly better estimates of the gradient [analogy: we are now tossing the coin k=2 times to estimate P(heads)]



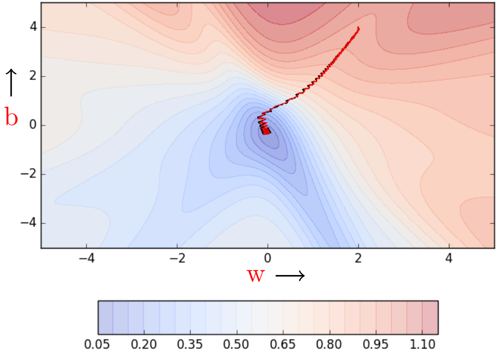
- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?
- Because we now have slightly better estimates of the gradient [analogy: we are now tossing the coin k=2 times to estimate P(heads)]
- The higher the value of k the more accurate are the estimates



- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?
- Because we now have slightly better estimates of the gradient [analogy: we are now tossing the coin k=2 times to estimate P(heads)]
- The higher the value of k the more accurate are the estimates
- In practice, typical values of k are 16, 32, 64



- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?
- Because we now have slightly better estimates of the gradient [analogy: we are now tossing the coin k=2 times to estimate P(heads)]
- The higher the value of k the more accurate are the estimates
- In practice, typical values of k are 16, 32, 64
- Of course, there are still oscillations and they will always be there as long as we are using an approximate gradient as opposed to the true gradient



Some things to remember ....

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	
Stochastic Gradient Descent	
Mini-Batch Gradient Descent	

Some things to remember ....

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	
Mini-Batch Gradient Descent	

Some things to remember ....

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	

Some things to remember ....

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	$\frac{N}{B}$

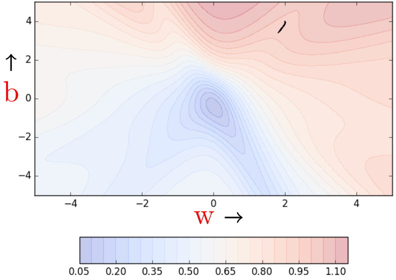
Similarly, we can have stochastic versions of Momentum based gradient descent and Nesterov accelerated based gradient descent

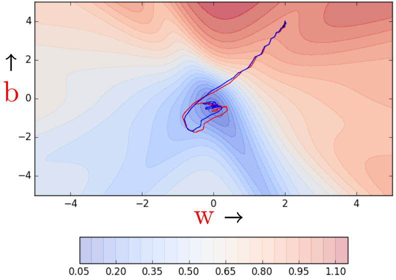
```python
def do_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        v_w = gamma * prev_v_w + eta* dw
        v_b = gamma * prev_v_b + eta* db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```

```python
def do_stochastic_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)

            v_w = gamma * prev_v_w + eta* dw
            v_b = gamma * prev_v_b + eta* db
            w = w - v_w
            b = b - v_b
            prev_v_w = v_w
            prev_v_b = v_b
```

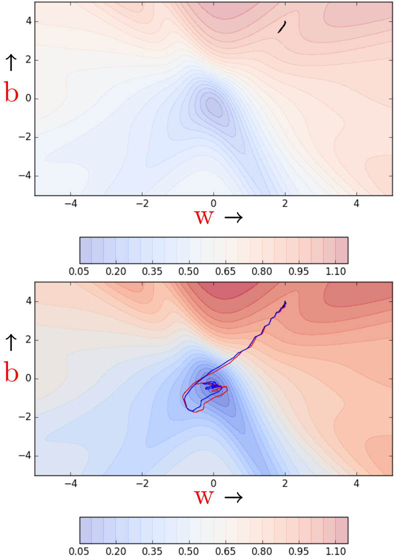
```python
def do_nesterov_accelerated_gradient_descent() :

    w, b, eta = init_w, init_b  , 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        #do partial updates
        v_w = gamma * prev_v_w
        v_b = gamma * prev_v_b
        for x,y in zip(X, Y) :
            #calculate gradients after partial update
            dw += grad_w(w - v_w, b - v_b, x, y)
            db += grad_b(w - v_w, b - v_b, x, y)

        #now do the full update
        v_w = gamma * prev_v_w + eta * dw
        v_b = gamma * prev_v_b + eta * db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```

```python
def do_nesterov_accelerated_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            #do partial updates
            v_w = gamma * prev_v_w
            v_b = gamma * prev_v_b
            #calculate gradients after partial update
            dw = grad_w(w - v_w, b - v_b, x, y)
            db = grad_b(w - v_w, b - v_b, x, y)

            v_w = gamma * prev_v_w + eta * dw
            v_b = gamma * prev_v_b + eta * db
            w = w - v_w
            b = b - v_b
            prev_v_w = v_w
            prev_v_b = v_b
```

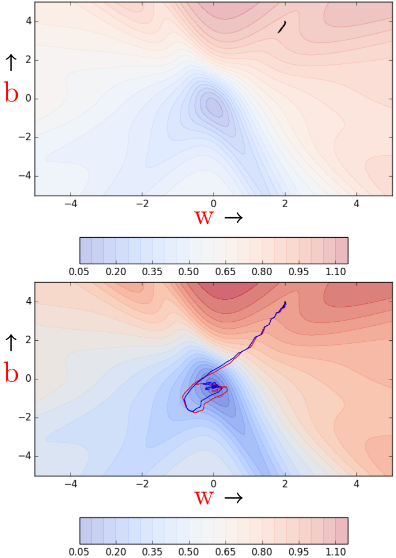

4
2
0
−2
−4
−4
−2
0
2
4
↑ b
w →
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10


4
2
0
−2
−4
−4
−2
0
2
4
↑ b
w →
0.05
0.20
0.35
0.50
0.65
0.80
0.95
1.10

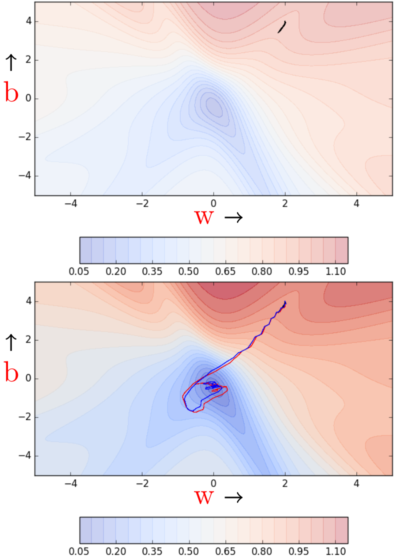
- While the stochastic versions of both Momentum [red] and NAG [blue] exhibit oscillations the relative advantage of NAG over Momentum still holds



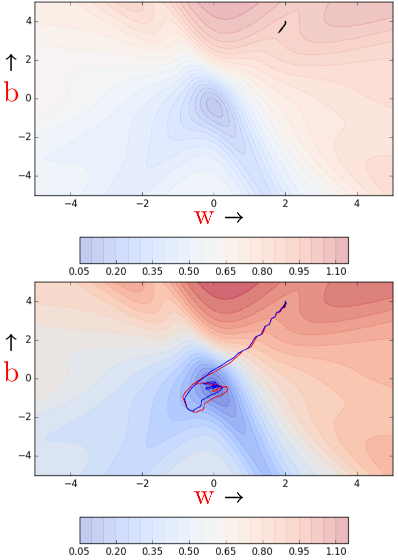
- While the stochastic versions of both Momentum [red] and NAG [blue] exhibit oscillations the relative advantage of NAG over Momentum still holds (i.e., NAG takes relatively shorter u-turns)



- While the stochastic versions of both Momentum [red] and NAG [blue] exhibit oscillations the relative advantage of NAG over Momentum still holds (i.e., NAG takes relatively shorter u-turns)
- Further both of them are faster than stochastic gradient descent



- While the stochastic versions of both Momentum [red] and NAG [blue] exhibit oscillations the relative advantage of NAG over Momentum still holds (i.e., NAG takes relatively shorter u-turns)
- Further both of them are faster than stochastic gradient descent (after 60 steps, stochastic gradient descent [black - top figure] still exhibits a very high error whereas NAG and Momentum are close to convergence)



And, of course, you can also have the mini batch version of Momentum and NAG...

And, of course, you can also have the mini batch version of Momentum and NAG...I leave that as an exercise :-)