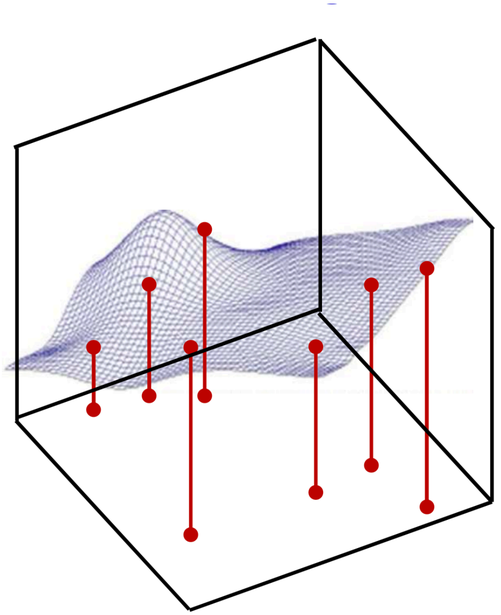
Neural Networks T&P

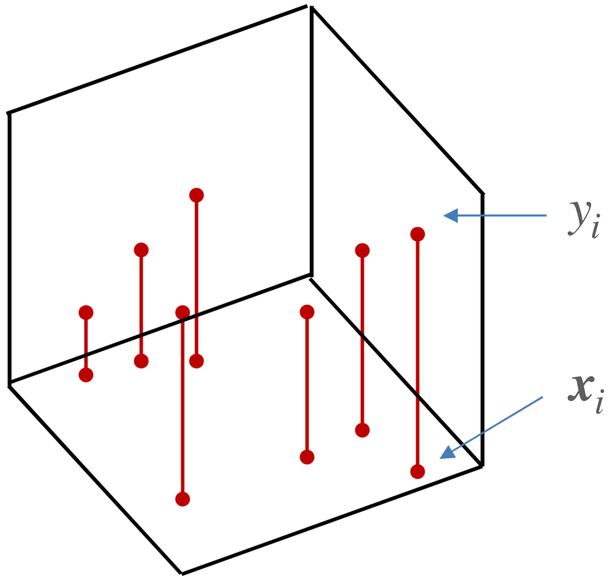
Training NN / Binary + Multiclass / Categorical CE
Stochastic Gradient Descent

Rafał Nowak March 18, 2025

Learning by sampling (recap)

- training set $X_{\text{train}} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N]^T \in \mathbb{R}^{N \times n}$, $y_{\text{train}} = [\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_N]^T \in \mathbb{R}^{N \times k}$
- $f(X, W) \approx g(X)$
- Questions:
  - good sampling?
- Examples:
  - images + labels
  - speech recordings and their transcription



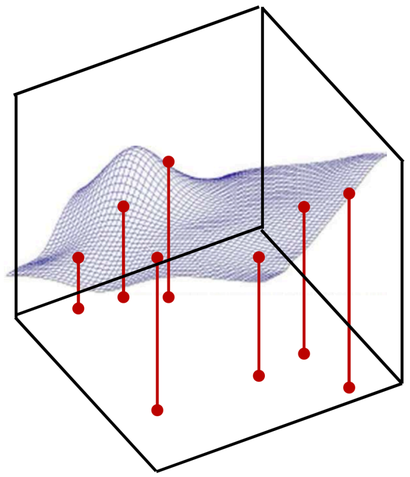


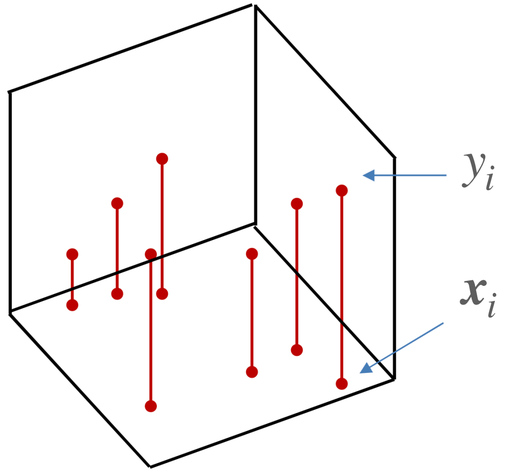
Learning the function (recap)

Error function (loss function)

- Training set $X_{\text{train}} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N]^T \in \mathbb{R}^{N \times n}$, $y_{\text{train}} = [\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_N]^T \in \mathbb{R}^{N \times k}$

$$Err(W) = \frac{1}{N} \sum_{i=1}^{N} div(f(\boldsymbol{x}_i; W), y_i)$$





Multiclass classification

Cross entropy loss

- Label (of class c) $\boldsymbol{y} = [0,0,\ldots,0,1,0,0,\ldots,0]$ is one hot vector with value 1 at c-th position
- Network output $f(\boldsymbol{x}) = \hat{\boldsymbol{y}} \in \mathbb{R}^c$ is probability distribution over c classes
  $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{z})$

$$\hat{y}_i = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)}$$

- **Cross Entropy loss (CE)**

$$div(\hat{y}, y) = -\log \hat{y}_c$$

Cross-Entropy Loss (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k \mid X = x_i) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

Softmax function

$$L_i = -\log P(Y = y_i \mid X = x_i)$$

	Unnormalized log-probabilities / logits	exp	Probabilities must be >= 0	normalize	Probabilities must sum to 1
cat	**3.2**	→	**24.5**	→	**0.13**
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00
	Unnormalized log-probabilities / logits		unnormalized probabilities		probabilities

$L_i = -\log(0.13)$
$= \mathbf{2.04}$

Maximum Likelihood Estimation
Choose weights to maximize the likelihood of the observed data
(See EECS 445 or EECS 545)

Source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Cross-Entropy Loss (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k \mid X = x_i) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

Softmax function

$$L_i = -\log P(Y = y_i \mid X = x_i)$$

	Unnormalized log-probabilities / logits	exp	Probabilities must be >= 0	normalize	Probabilities must sum to 1		Correct probs
cat	**3.2**	→	**24.5**	→	**0.13**	Compare	**1.00**
car	5.1	→	164.0	→	0.87		0.00
frog	-1.7	→	0.18	→	0.00		0.00

Kullback–Leibler divergence

$$D_{KL}(\boldsymbol{P} \,||\, \boldsymbol{Q}) = \sum_{\boldsymbol{y}} \boldsymbol{P}(\boldsymbol{y}) \log \frac{\boldsymbol{P}(\boldsymbol{y})}{\boldsymbol{Q}(\boldsymbol{y})}$$

Source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Cross-Entropy Loss (Multinomial Logistic Regression)

cat	**3.2**
car	5.1
frog	-1.7

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k \mid X = x_i) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

Softmax function

Maximize probability of correct class

$$L_i = -\log P(Y = y_i \mid X = x_i)$$

Putting it all together:

$$L_i = -\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right)$$

Q: If all scores are small random values, what is the loss?

A: -log(1/C)
log(10) ≈ 2.3

Source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Cross entropy loss

$$X \in \mathbb{R}^{N\times 5}$$

$$Z^{(1)} = X\,W^{(1)^T} + \boldsymbol{b}^{(1)} \in \mathbb{R}^{N\times 7} \qquad A^{(1)} = g_1(Z^{(1)}) \in \mathbb{R}^{N\times 7}$$

$$Z^{(2)} = A^{(1)}\,W^{(2)^T} + \boldsymbol{b}^{(2)} \in \mathbb{R}^{N\times 4} \qquad A^{(2)} = g_2(Z^{(2)}) \in \mathbb{R}^{N\times 4}$$

$$Z^{(3)} = A^{(2)}\,W^{(3)^T} + \boldsymbol{b}^{(3)} \in \mathbb{R}^{N\times 3} \qquad A^{(3)} = \text{softmax}(Z^{(3)}) \in \mathbb{R}^{N\times 3}$$

- We have **3 classes (0, 1, 2)** and assume we have 4 samples ($N = 4$) of the following classes: 2, 0, 1, 2.

$$Y = \begin{bmatrix} 0,0,1 \\ 1,0,0 \\ 0,1,0 \\ 0,0,1 \end{bmatrix} \in \mathbb{R}^{N\times 3}$$

(on-hot encoding)

- **Cross entropy loss**

$$L = -\frac{1}{N}\,\text{sum}\left(Y \otimes \log A^{(3)}\right) = -\frac{1}{N}\,\text{sum}\begin{bmatrix} Y[1,:] \otimes A^{(3)}[1,:] \\ Y[2,:] \otimes A^{(3)}[2,:] \\ \vdots \\ Y[N,:] \otimes A^{(3)}[N,:] \end{bmatrix}$$

Cross entropy loss

Demo with PyTorch

- Modular API
- Functional API

- Examples:
  Binary_and_multiclass_classification.ipynb

using `BCELoss` or `BCEWithLogitsLoss`

see https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html#torch.nn.BCEWithLogitsLoss

One can use the *modular* or *functional* API from PyTorch

Modular API

- torch.nn.BCELoss
- torch.nn.BCEWithLogitsLoss

```
In [9]:
loss_fn = torch.nn.BCELoss()
loss_fn_with_logits = torch.nn.BCEWithLogitsLoss()
```

executed in 6ms, finished 16:48:06 2022-03-29

```
In [10]:
loss_fn(y_pred, y_true)
```

executed in 9ms, finished 16:48:06 2022-03-29

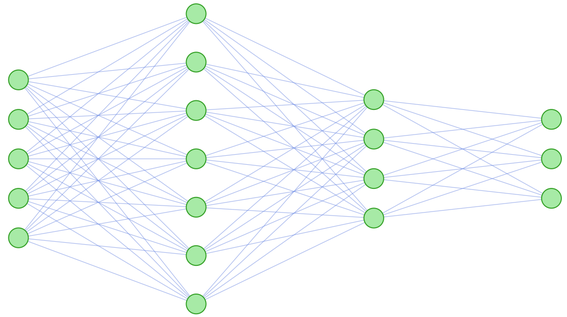
```
Out[10]: tensor(0.4485, dtype=torch.float64)
```

```
In [11]:
loss_fn_with_logits(logits, y_true)
```

executed in 8ms, finished 16:48:07 2022-03-29

```
Out[11]: tensor(0.4485, dtype=torch.float64)
```

Backprop in pytorch



Multilayer perceptron

```
X = torch.tensor(
    [
        [-8.0, 1.0, -3.0, 5.0, 2.0],
        [-2.0, 2.0, -1.0, 5.0, 2.0],
        [7.0, 1.0, 9.0, -5.0, -2.0],
        [3.0, 1.0, 9.0,  5.0,  2.0]
    ])
X
```

executed in 11ms, finished 14:08:51 2023-03-12

```
tensor([[-8.,  1., -3.,  5.,  2.],
        [-2.,  2., -1.,  5.,  2.],
        [ 7.,  1.,  9., -5., -2.],
        [ 3.,  1.,  9.,  5.,  2.]])
```

```
X.shape
```

executed in 9ms, finished 14:08:51 2023-03-12

```
torch.Size([4, 5])
```

```
Y = torch.tensor(
    [
        [0.0, 0.0, 1.0],
        [1.0, 0.0, 0.0],
        [0.0, 1.0, 0.0],
        [0.0, 0.0, 1.0]
    ])
```

executed in 5ms, finished 14:08:51 2023-03-12

```
# Model parameters

W1 = torch.rand(7, 5)  #  R^{7 x 5}
b1 = torch.rand(7)     #  R^7

W2 = torch.rand(4, 7)  #  R^{4 x 7}
b2 = torch.rand(4)     #  R^4

W3 = torch.rand(3, 4)  #  R^{3 x 4}
b3 = torch.rand(3)     #  R^3
```

executed in 4ms, finished 14:08:51 2023-03-12

```
# Forward pass
Z1 = torch.mm(X, W1.T) + b1
A1 = torch.sigmoid(Z1)

Z2 = torch.mm(A1, W2.T) + b2
A2 = torch.sigmoid(Z2)

Z3 = torch.mm(A2, W3.T) + b3
A3 = torch.softmax(Z3, dim=1)   # Note: use dim=1
```

executed in 8ms, finished 14:08:51 2023-03-12

```
A1.shape, A2.shape, A3.shape
```

executed in 4ms, finished 14:08:51 2023-03-12

```
(torch.Size([4, 7]), torch.Size([4, 4]), torch.Size([4, 3]))
```

```
A3
```

executed in 5ms, finished 14:08:51 2023-03-12

```
tensor([[0.4358, 0.2137, 0.3505],
        [0.4785, 0.1890, 0.3325],
        [0.4795, 0.1882, 0.3324],
        [0.4797, 0.1880, 0.3323]])
```

```
# Cross entropy loss

N = len(X)
L = 1/N * torch.sum( Y * A3 )
L
```

executed in 11ms, finished 14:08:51 2023-03-12

```
tensor(0.3374)
```

$$L = -\frac{1}{N}\,\text{sum}\left(Y \otimes \log A^{(3)}\right)$$

Backprop →

```
# W1 = torch.tensor( [ .... ], requires_grad=True )

W1 = W1.clone().requires_grad_(True)
W2 = W2.clone().requires_grad_(True)
W3 = W3.clone().requires_grad_(True)
b1 = b1.clone().requires_grad_(True)
b2 = b2.clone().requires_grad_(True)
b3 = b3.clone().requires_grad_(True)
```

executed in 2ms, finished 14:08:51 2023-03-12

```
# Forward pass
Z1 = torch.mm(X, W1.T) + b1
A1 = torch.sigmoid(Z1)

Z2 = torch.mm(A1, W2.T) + b2
A2 = torch.sigmoid(Z2)

Z3 = torch.mm(A2, W3.T) + b3
A3 = torch.softmax(Z3, dim=1)   # Note: use dim=1
```

executed in 3ms, finished 14:08:51 2023-03-12

```
A3
```

executed in 2ms, finished 14:08:51 2023-03-12

```
tensor([[0.4358, 0.2137, 0.3505],
        [0.4785, 0.1890, 0.3325],
        [0.4795, 0.1882, 0.3324],
        [0.4797, 0.1880, 0.3323]], grad_fn=<SoftmaxBackward0>)
```

Note the result with `grad_fn=<SoftmaxBackward0>`

```
L = - 1/N * torch.sum( Y * torch.log(A3) );
L
```

executed in 2ms, finished 14:08:51 2023-03-12

```
tensor(1.1394, grad_fn=<MulBackward0>)
```

Note the result with `grad_fn=<MulBackward0>`

```
Z1.retain_grad(); A1.retain_grad()
Z2.retain_grad(); A2.retain_grad()
Z3.retain_grad(); A3.retain_grad()

L.backward()
```

executed in 9ms, finished 14:08:51 2023-03-12

$$\frac{\partial L}{\partial A^{(3)}} = -\frac{1}{N}\,Y \div A^{(3)}$$

```
- 1/N * Y / A3
```

executed in 4ms, finished 14:08:51 2023-03-12

```
tensor([[-0.0000, -0.0000, -0.7133],
        [-0.5224, -0.0000, -0.0000],
        [-0.0000, -1.3285, -0.0000],
        [-0.0000, -0.0000, -0.7523]], grad_fn=<DivBackward0>)
```

```
A3.grad
```

executed in 2ms, finished 14:08:51 2023-03-12

```
tensor([[-0.0000, -0.0000, -0.7133],
        [-0.5224, -0.0000, -0.0000],
        [-0.0000, -1.3285, -0.0000],
        [-0.0000, -0.0000, -0.7523]])
```

$$\frac{\partial L}{\partial Z^{(3)}} = \frac{1}{N}(A^{(3)} - Y)$$

```
1/N * (A3-Y)
```

executed in 3ms, finished 14:08:51 2023-03-12

```
tensor([[ 0.1090,  0.0534, -0.1624],
        [-0.1304,  0.0473,  0.0831],
        [ 0.1199, -0.2030,  0.0831],
        [ 0.1199,  0.0470, -0.1669]], grad_fn=<MulBackward0>)
```

```
Z3.grad
```

executed in 3ms, finished 14:08:51 2023-03-12

```
tensor([[ 0.1090,  0.0534, -0.1624],
        [-0.1304,  0.0473,  0.0831],
        [ 0.1199, -0.2030,  0.0831],
        [ 0.1199,  0.0470, -0.1669]])
```

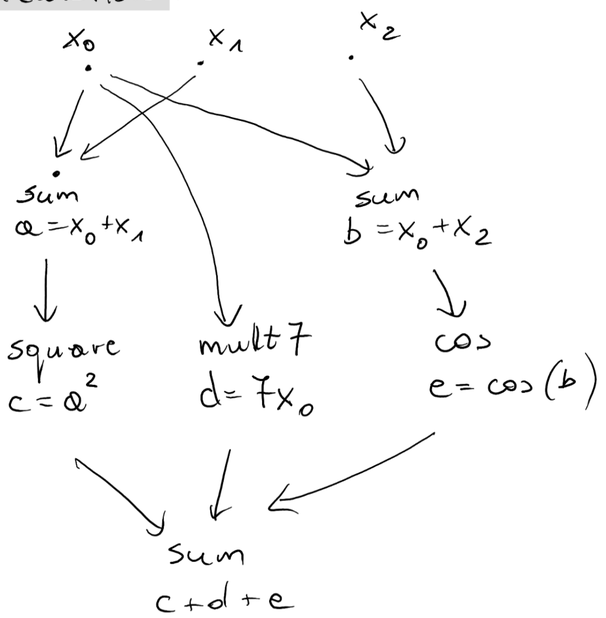
Backpropagation graph

Backpropagation (example)

$$f(x) = (x_0 + x_1)^2 + 7x_0 + \cos(x_0 + x_2)$$

$$x = [2, 3, -2]$$

Calculation:



$$\frac{\partial f}{\partial x_1} = 2(x_0 + x_1) \cdot 1 + 7 - \sin(x_0 + x_2) \cdot 1$$

Example 2. Gradient of simple vector function

$$f(x) = (x_0 + x_1)^2 + 7x_0 + \cos(x_0 + x_2)$$

```
[ ]  1  def f(x):
     2      return (x[0]+x[1])**2 + 7*x[0] + torch.cos(x[0]+x[2])
```

```
[ ]  1  x = torch.tensor([2.0, 3.0, -2.0], requires_grad=True, device='cpu')
     2  y = f(x)
     3  y
```

```
tensor(40., grad_fn=<AddBackward0>)
```

$$\partial f / \partial x = \begin{bmatrix} 2(x_0 + x_1) + 7 - \sin(x_0 + x_2) \\ 2(x_0 + x_1) \\ -\sin(x_0 + x_2) \end{bmatrix}$$

```
[ ]  1  print(x.grad)
```

```
None
```

```
[ ]  1  y.backward()
```

```
[ ]  1  x.grad
```

```
tensor([17., 10.,  0.])
```

```
[ ]  1  # Intermediate variables
     2
     3  x = torch.tensor([2.0, 3.0, -2.0], requires_grad=True, device='cpu')
     4  a = x[0] + x[1]
     5  b = x[0] + x[2]
     6  c = a**2
     7  d = 7*x[0]
     8  e = torch.cos(b)
     9  y = c+d+e
    10  y
```

```
tensor(40., grad_fn=<AddBackward0>)
```

```
[ ]  1  y.backward()
```

see example code: Backprop - autograd (pytorch)

Model building (in pytorch)

MLP building

torch.nn.Linear

`torch.nn.Linear`

In [3]:
```
from torch.nn import Linear
```

In [4]:
```
model_1 = Linear(4, 5)
model_2 = Linear(5, 7)
model_3 = Linear(7, 3)

def model(x):
    x = model_1(x)
    x = torch.tanh(x)
    x = model_2(x)
    x = torch.tanh(x)
    logits = model_3(x)
    return logits
```

In [5]:
```
model(X_input)
```

Out[5]:
```
tensor([[-0.2232,  0.0404, -0.0912],
        [ 0.2082,  0.3690, -0.0370],
        [ 0.1019,  0.2788, -0.0246],
        [ 0.2656,  0.4020, -0.0071],
        [ 0.1335,  0.1268,  0.1896],
        [-0.3925, -0.2287,  0.1415],
        [ 0.0502,  0.3030, -0.1084],
        [-0.0103,  0.0464,  0.1387],
        [ 0.1726,  0.3016, -0.0038],
        [-0.2446,  0.0146, -0.0266]], grad_fn=<AddmmBackward0>)
```

In [6]:
```
torch.softmax( model(X_input), dim=1 )[:5]
```

Out[6]:
```
tensor([[0.2905, 0.3781, 0.3315],
        [0.3382, 0.3972, 0.2647],
        [0.3252, 0.3882, 0.2866],
        [0.3439, 0.3942, 0.2618],
        [0.3278, 0.3256, 0.3467]], grad_fn=<SliceBackward0>)
```

torch.nn.Sequential

Sequencial model building

```
from torch.nn import Sequential, Linear, Tanh, Flatten
```

```
model = Sequential(
    Flatten(),
    Linear(28*28, 100),
    Tanh(),
    Linear(100, 10)
)
```

```
for p in model.parameters():
    print(p.shape)
```

```
torch.Size([100, 784])
torch.Size([100])
torch.Size([10, 100])
torch.Size([10])
```

torch.nn.Module

Base class for all neural network modules

- https://pytorch.org/docs/stable/generated/torch.nn.Module.html
- model.training
- model.train() / model.eval()
  - Set the module in training / evaluation mode
- model.cpu() / model.cuda()
  - Move all model parameters and buffers to the CPU / GPU

In [17]:
```
class MyModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.model_1 = Linear(4, 5)
        self.model_2 = Linear(5, 7)
        self.model_3 = Linear(7, 3)

    def forward(self, x):
        x = self.model_1(x)
        x = torch.tanh(x)
        x = self.model_2(x)
        x = torch.tanh(x)
        x = self.model_3(x) # logits
        return x
```

In [18]:
```
model = MyModel()
```

In [19]:
```
model(X_input)
```

Out[19]:
```
tensor([[-0.1078, -0.2213,  0.3109],
        [-0.2863, -0.1555,  0.3096],
        [ 0.0231, -0.5331,  0.1133],
        [-0.0299, -0.4571,  0.1467],
        [ 0.1102, -0.4696,  0.2299],
        [ 0.2339, -0.5825,  0.1450],
        [-0.1765, -0.2206,  0.2782],
        [ 0.0843, -0.5875,  0.1477],
        [ 0.1531, -0.7359,  0.0648],
        [ 0.0805, -0.4202,  0.1711]], grad_fn=<AddmmBackward0>)
```

torch.nn.ModuleList

Holds submodules in a list.

CLASS torch.nn.ModuleList(*modules=None*) [SOURCE]

Holds submodules in a list.

`ModuleList` can be indexed like a regular Python list, but modules it contains are properly registered, and will be visible by all `Module` methods.

Parameters

modules (*iterable, optional*) – an iterable of modules to add

Example:

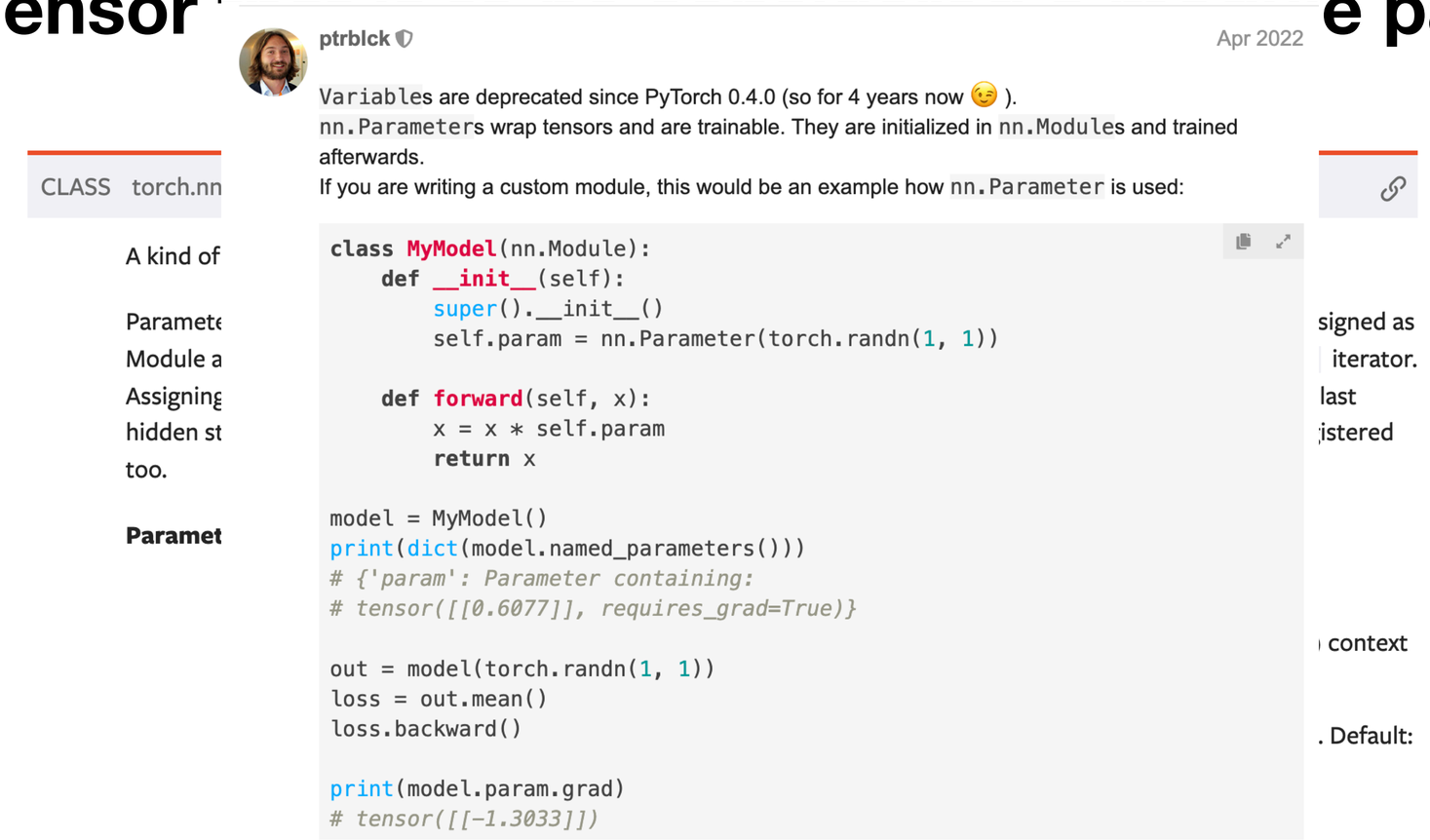
```
class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

- https://pytorch.org/docs/stable/generated/torch.nn.ModuleList.html

torch.nn.Parameter

A kind of Tensor that is to be considered a module parameter



CLASS torch.nn

A kind of

Paramete
Module a
Assigning
hidden st
too.

Paramet

signed as
iterator.
last
istered

context

. Default:

ptrblck Apr 2022

`Variables` are deprecated since PyTorch 0.4.0 (so for 4 years now 😉).
`nn.Parameters` wrap tensors and are trainable. They are initialized in `nn.Modules` and trained afterwards.
If you are writing a custom module, this would be an example how `nn.Parameter` is used:

```
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.param = nn.Parameter(torch.randn(1, 1))

    def forward(self, x):
        x = x * self.param
        return x

model = MyModel()
print(dict(model.named_parameters()))
# {'param': Parameter containing:
# tensor([[0.6077]], requires_grad=True)}

out = model(torch.randn(1, 1))
loss = out.mean()
loss.backward()

print(model.param.grad)
# tensor([[-1.3033]])
```

- https://pytorch.org/docs/stable/generated/torch.nn.parameter.Parameter.html

Demo code

LinearLayer

- see also https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

```python
class LinearLayer(nn.Module):
    def __init__(self, in_features: int, out_features: int, init="xavier"):
        """
        Initialize a linear layer with a given initialization method
        :param in_features: number of input features
        :param out_features: number of output features
        :param init: initialization method (xavier, he, normal, uniform, zeros, ones)

        :returns: Linear layer with given initialization method
        """
        super(LinearLayer, self).__init__()
        self.in_features = torch.tensor(in_features)
        self.out_features = torch.tensor(out_features)

        if init == "xavier":
            self.weight = nn.Parameter(
                torch.randn(out_features, in_features)
                * torch.sqrt(2 / (self.in_features + self.out_features))
            )
        elif init == "he":
            self.weight = nn.Parameter(
                torch.randn(out_features, in_features)
                * torch.sqrt(2 / self.in_features)
            )
        elif init == "normal":
            self.weight = nn.Parameter(torch.randn(out_features, in_features))
        elif init == "uniform":
            self.weight = nn.Parameter(torch.rand(out_features, in_features) * 2 - 1)
        elif init == "zeros":
            self.weight = nn.Parameter(torch.zeros(out_features, in_features))
        elif init == "ones":
            self.weight = nn.Parameter(torch.ones(out_features, in_features))
        else:
            raise ValueError("Unknown initialization method")

        self.bias = nn.Parameter(torch.zeros(out_features))

    def forward(self, x):
        return torch.nn.functional.linear(x, self.weight, self.bias)
```

Training the neural network

- We must pay attention to the **magnitude of weights** and the magnitude of **gradients** in each layer, we want that the gradient updates are not too lare with comparison to weight magnitude but that they are not negligible either.
- We must ensure that the scale of values that flow through the network is controlled. We do it using **input normalization** and careful **weigth initialization**.

Input normalization

- Rarely, neural networks are applied directly to the raw data of a dataset.
- We need a data preparation
- Normalization
  - $x' = \frac{x - x_{min}}{x_{max} - x_{min}}(u - l) + l$
- Standarization
  - $x' = \frac{x - \mu}{\sigma}$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Datasets

… from torchvision …

```
!pip install torch
!pip install torchvision
```

In [1]:
```
import torch
import torchvision
```

In [3]:
```
from torchvision.datasets import MNIST
```

In [4]:
```
dataset_train = MNIST(root='datasets/', download=True, train=True)
dataset_test  = MNIST(root='datasets/', download=False, train=False)
```

In [6]:
```
len(dataset_train), len(dataset_test)
```

Out[6]:
```
(60000, 10000)
```

In [12]:
```
X_train = dataset_train.data / 255
X_test = dataset_test.data / 255

X_train = torch.flatten( X_train, start_dim=1 )
X_test = torch.flatten( X_test, start_dim=1 )
```

In [13]:
```
X_train.shape, X_test.shape
```

Out[13]:
```
(torch.Size([60000, 784]), torch.Size([10000, 784]))
```

Training loop (pytorch live demo)

SGD

```
# Training loop (without using batches)
n_epochs = 3
for epoch in range(n_epochs):

    images, labels = dataset

    # Forward propagation
    logits = model( images )

    # Cross entropy loss
    loss = criterion( logits, labels )

    # Backward propagation
    loss.backward()

    # Single step of Gradient Descent
    # TODO: Update model parameters

    # Zeroing gradients (for next iteration)
    # TODO: Zero gradients


    with torch.no_grad():
        pred_class = torch.argmax(logits, dim=1)
        accuracy = torch.sum(pred_class == labels)/len(labels)

    print(f"epoch={epoch:3d}, loss={loss.item():.3f}, accuracy={accuracy.item():.3f}")
```

Overfitting

Overfitting

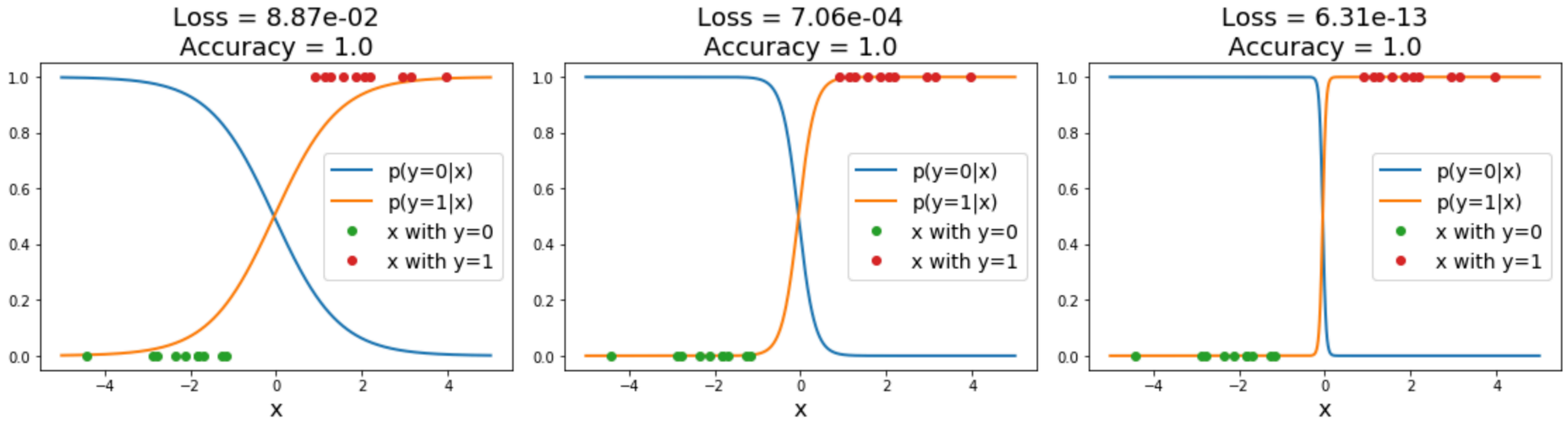
A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i$$

$$p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y)$$



Both models have perfect accuracy on train data!

Low loss, but unnatural "cliff" between training points

source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

λ is a hyperparameter giving regularization strength

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_{k,l} W_{k.l}^2$

L1 regularization: $R(W) = \sum_{k,l} |W_{k,l}|$

More complex:

Dropout

Batch normalization

Cutout, Mixup, Stochastic depth, etc…

source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

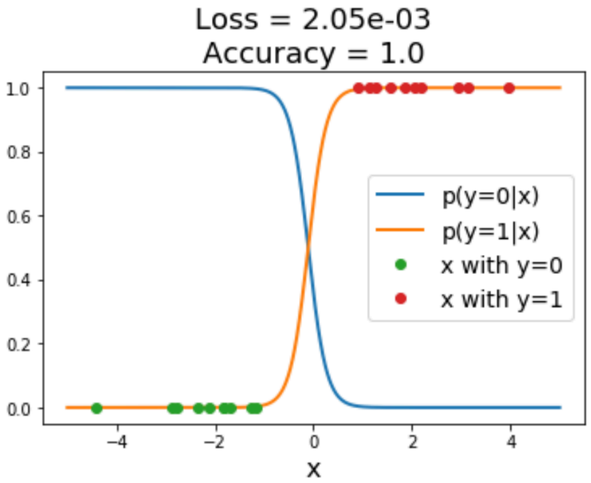
Regularization: Prefer Simpler Models

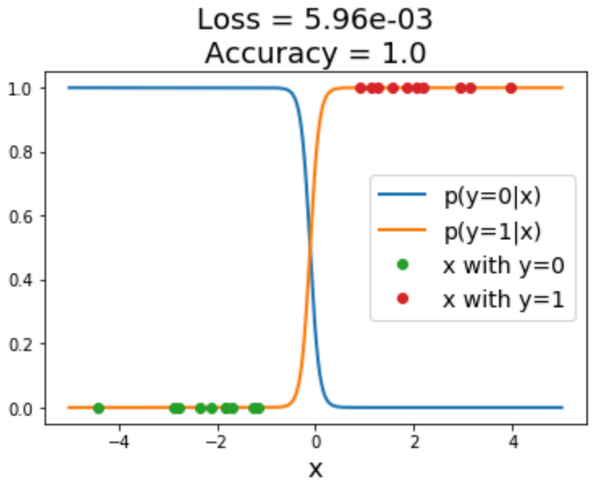
Example: Linear classifier with 1D inputs, 2 classes, softmax loss

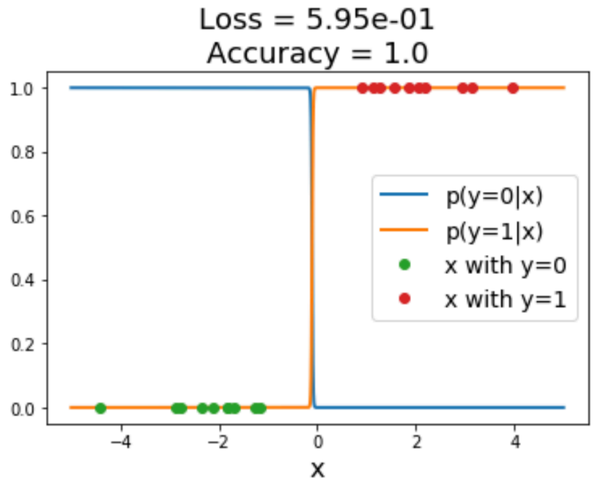
$$s_i = w_i x + b_i \qquad p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y) + \lambda \sum_i w_i^2$$

Regularization term causes loss to **increase** for model with sharp cliff







source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Regularization: Expressing Preferences

L2 Regularization

$$R(W) = \sum_{k,l} W_{k,l}^2$$

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

L2 regularization prefers weights to be "spread out"

$$w_1^T x = w_2^T x = 1$$

Same predictions, so data loss will always be the same

source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Finding a good W

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

Loss function consists of **data loss** to fit the training data and **regularization** to prevent overfitting

source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

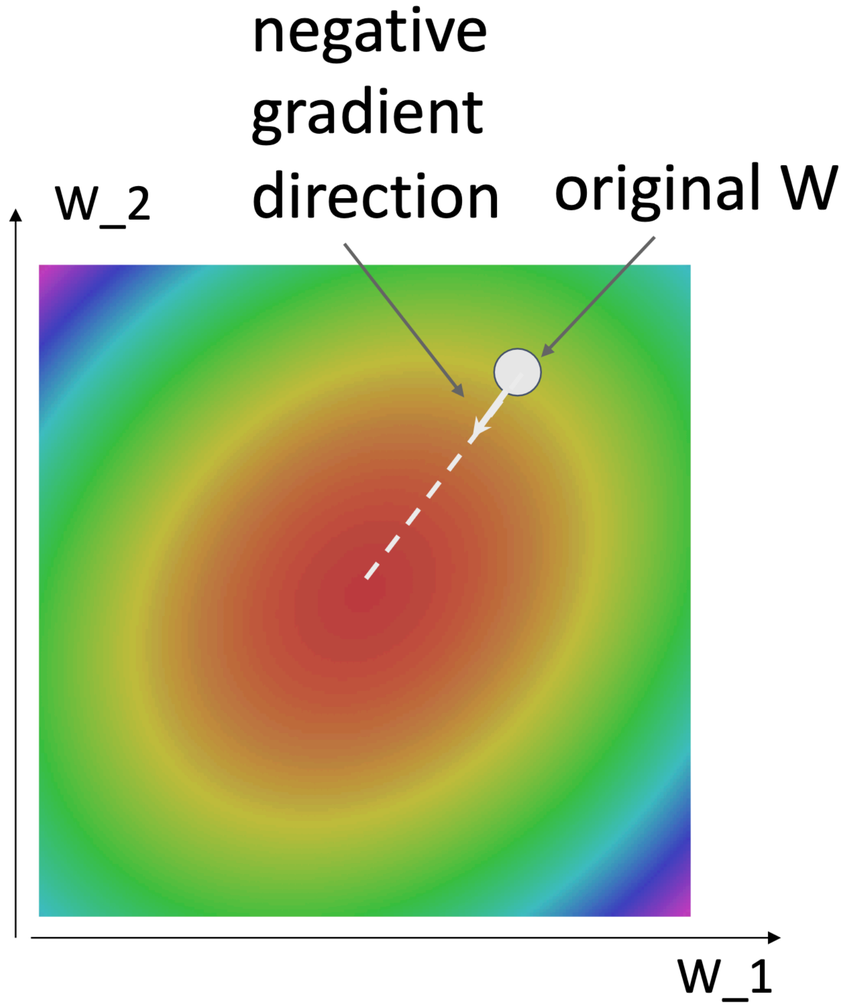
Gradient Descent

Iteratively step in the direction of the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate



source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Batch Gradient Descent

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples 32 / 64 / 128 common

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

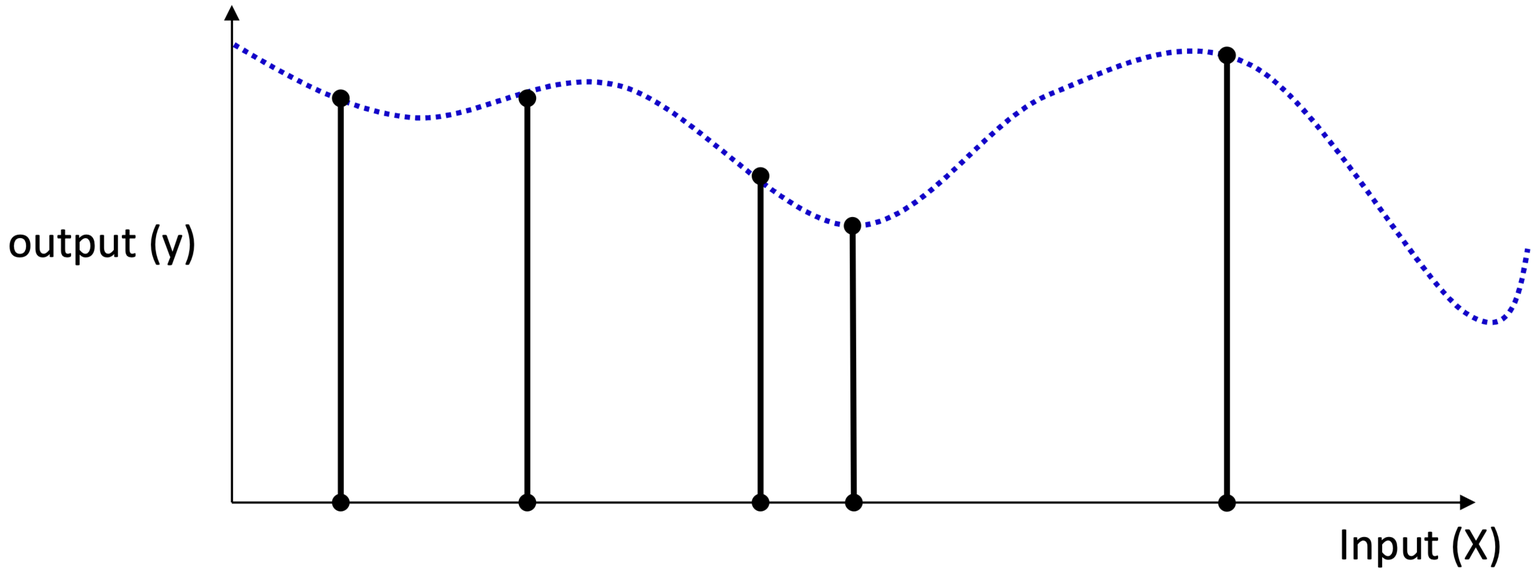
Incremental updates

- Gradient Descent doesn't exploit the structure of the loss function, and every step it takes requires computing the gradient on the **full data set**.

  This is different than e.g. the behavior of the perceptron learning machine, which updated its parameters after seeing each training example.

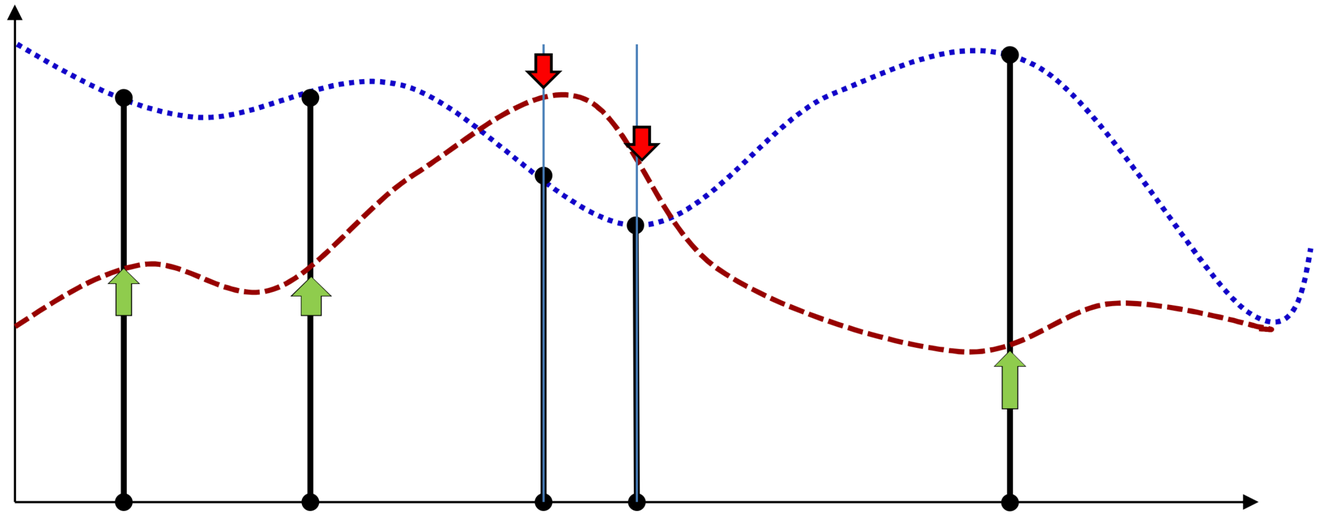
- Such behavior is implemented in the **Stochastic Gradient Descent** algorithm which updates the parameters based on the gradient evaluated on a small subset of the training data (perhaps on one sample!)

The training formulation



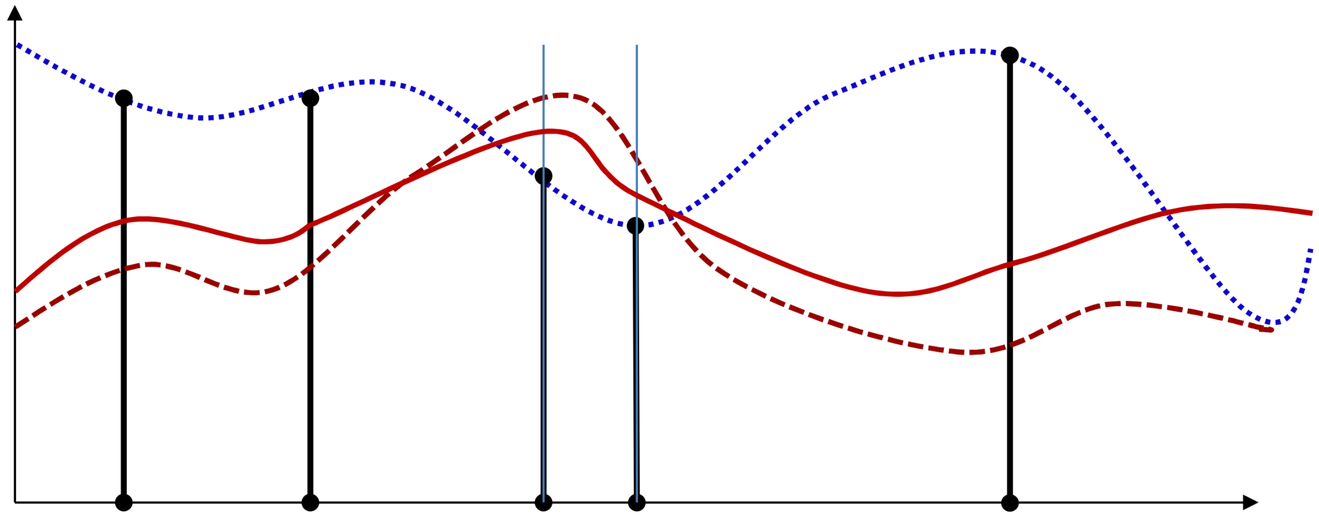
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



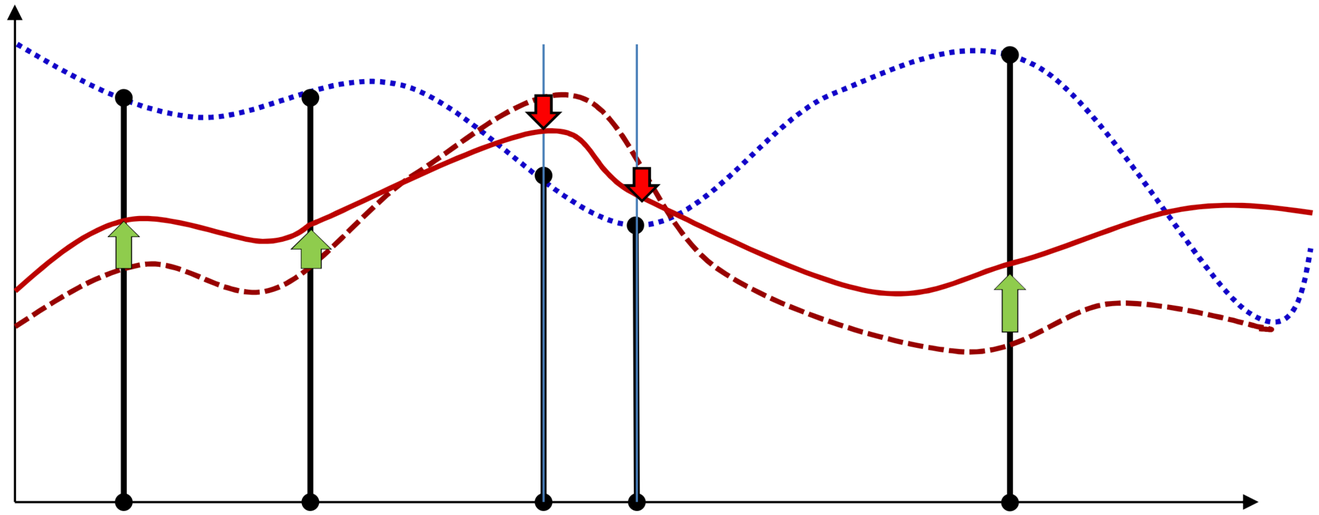
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



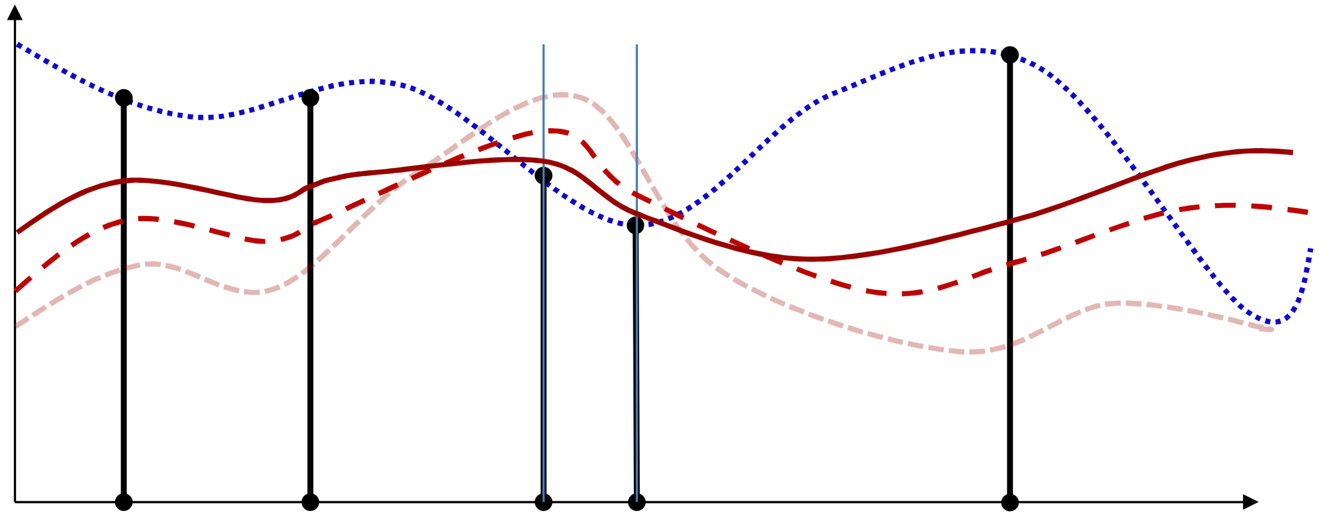
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



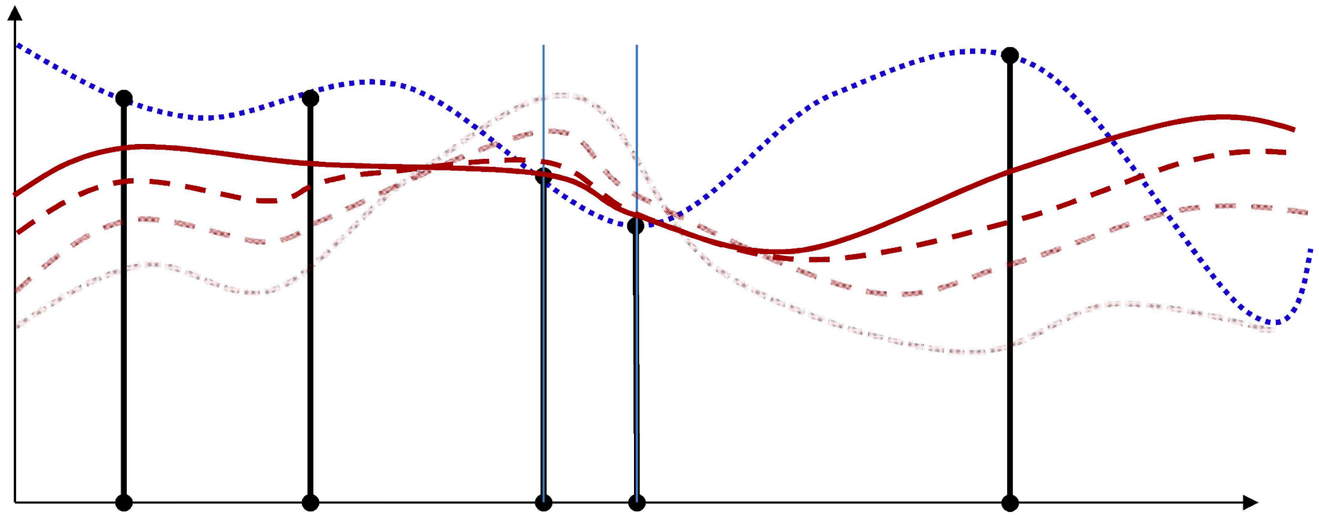
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



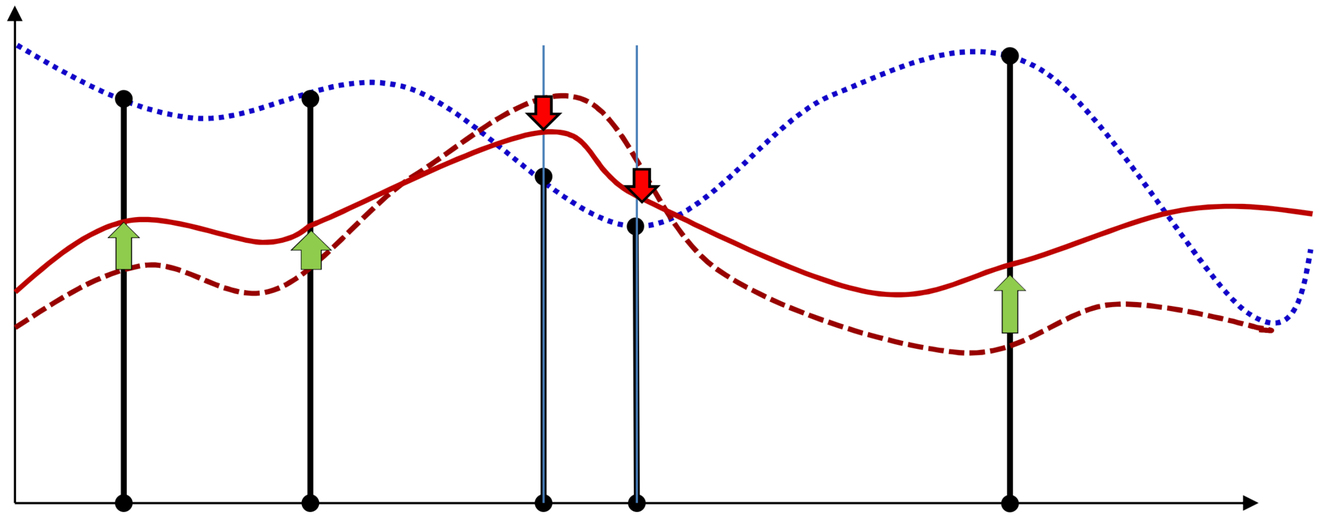
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



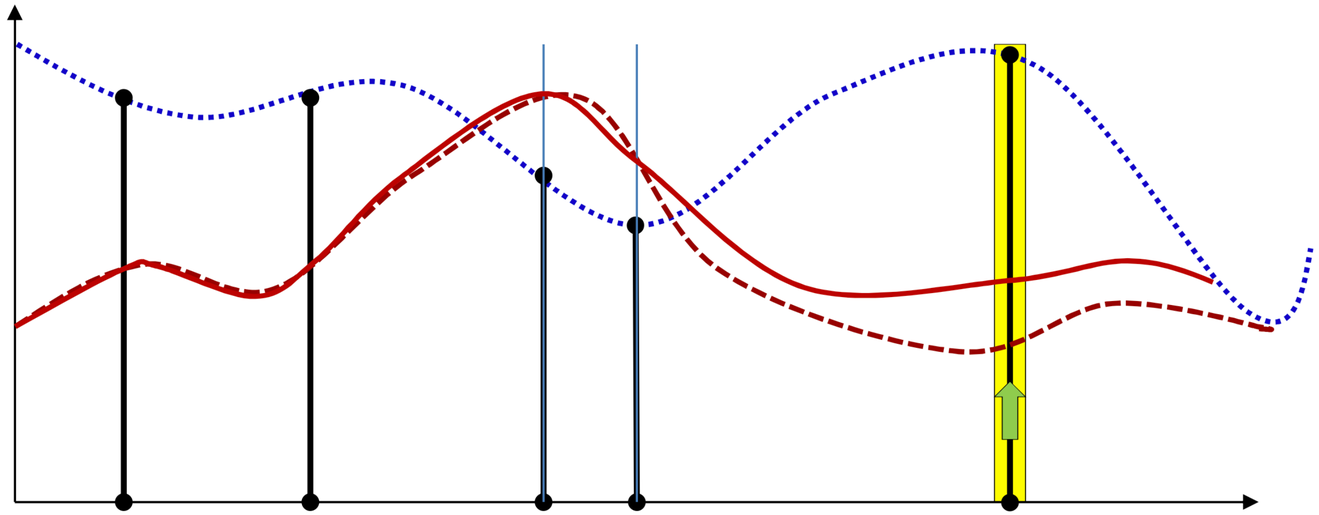
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Effect of number of samples



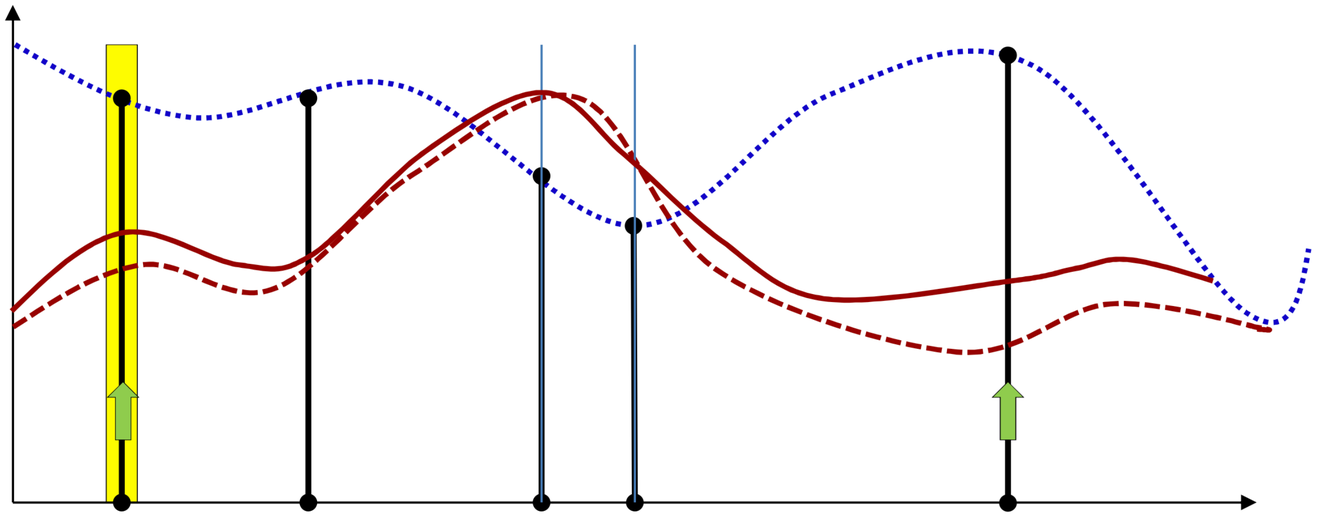
- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
  - We must process *all* training points before making a single adjustment
  - “Batch” update

Alternative: Incremental update



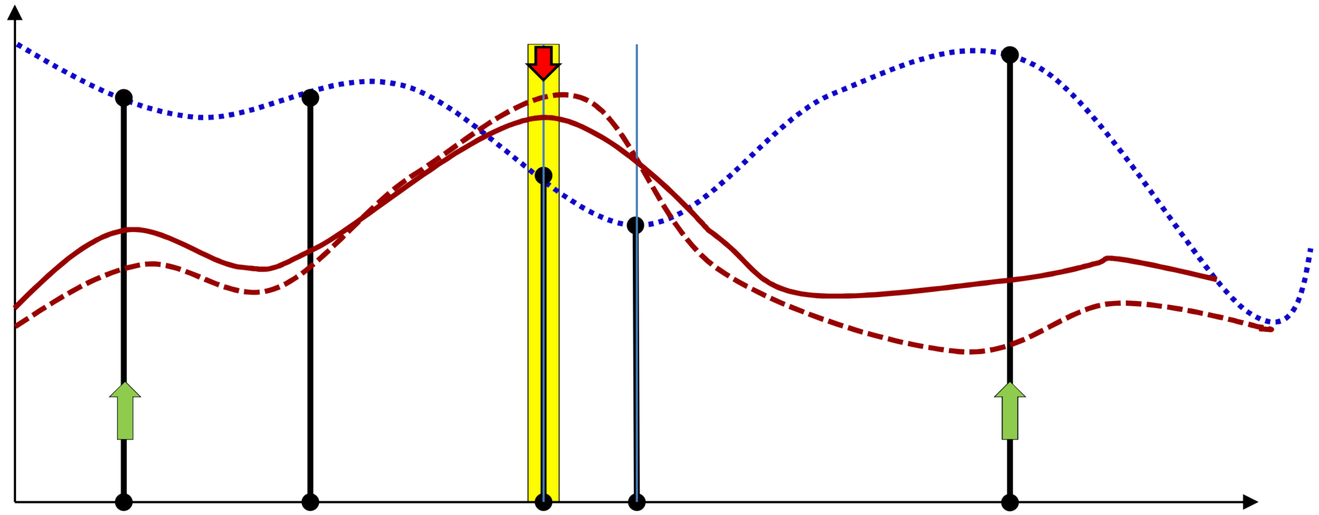
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



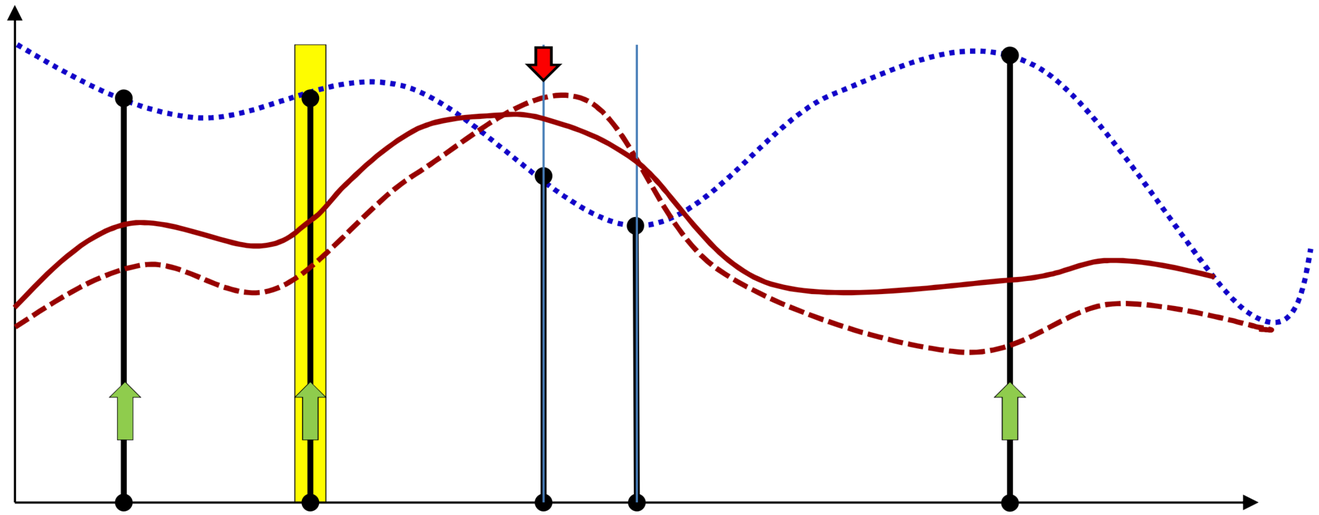
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



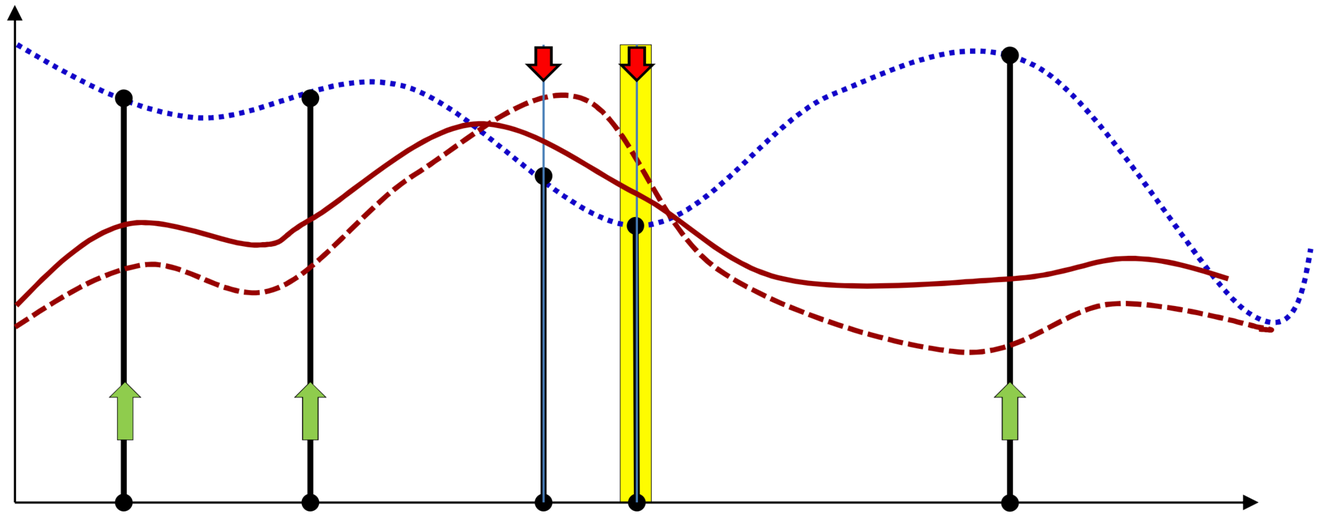
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
  - Keep adjustments small
  - Eventually, when we have processed all the training points, we will have adjusted the entire function
    - With *greater* overall adjustment than we would if we made a single "Batch" update

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples 32 / 64 / 128 common

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

source: https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

SGD terminology

SGD training uses some new vocabulary

- **iteration** means one **training step**
- **minibatch** is the data used for one iteration
- **epoch** is a full pass over training data.
- Typically during an epoch:
  - data is **shuffled**
  - minibatches are generated from the shuffled data (in this way each training sample is used once during an epoch)
  - training iterations are performed on each **minibatch**
  - learning rate, or step size denotes the **hyperparameter** α
  - It is often set according to a **schedule**, or based on **validation** results

DataLoader

… from torch.utils.data …

Dataloader

```
from torch.utils.data import DataLoader

BATCH_SIZE = 32

dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

```
for batch_nr, batch in enumerate(dataloader):
    images, labels = batch
```

Training loop (pytorch)
SGD

```
n_epochs = 3
for epoch in range(n_epochs):

    for batch_nr, batch in enumerate(dataloader):
        images, labels = batch

        # Forward propagation
        logits = model( images )

        # Cross entropy loss
        loss = criterion( logits, labels )

        # Backward propagation
        loss.backward()

        # Single step of SGD
        # TODO: Update model parameters

        # Zeroing gradients (for next iteration)
        # TODO: Zero gradients


        if batch_nr%10 == 0:
            with torch.no_grad():
                pred_class = torch.argmax(logits, dim=1)
                acc = torch.sum(pred_class == labels)/len(labels)

            print(f"epoch={epoch:3d}, i={batch_nr:4d}, loss={loss.item():.3f}, accuracy={acc.item():.3f}")
```