Neural Networks

PyTorch + some training techniques

Rafał Nowak March 25, 2025

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

MODULE

CLASS torch.nn.Module(*args, **kwargs) [SOURCE]

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

• NOTE

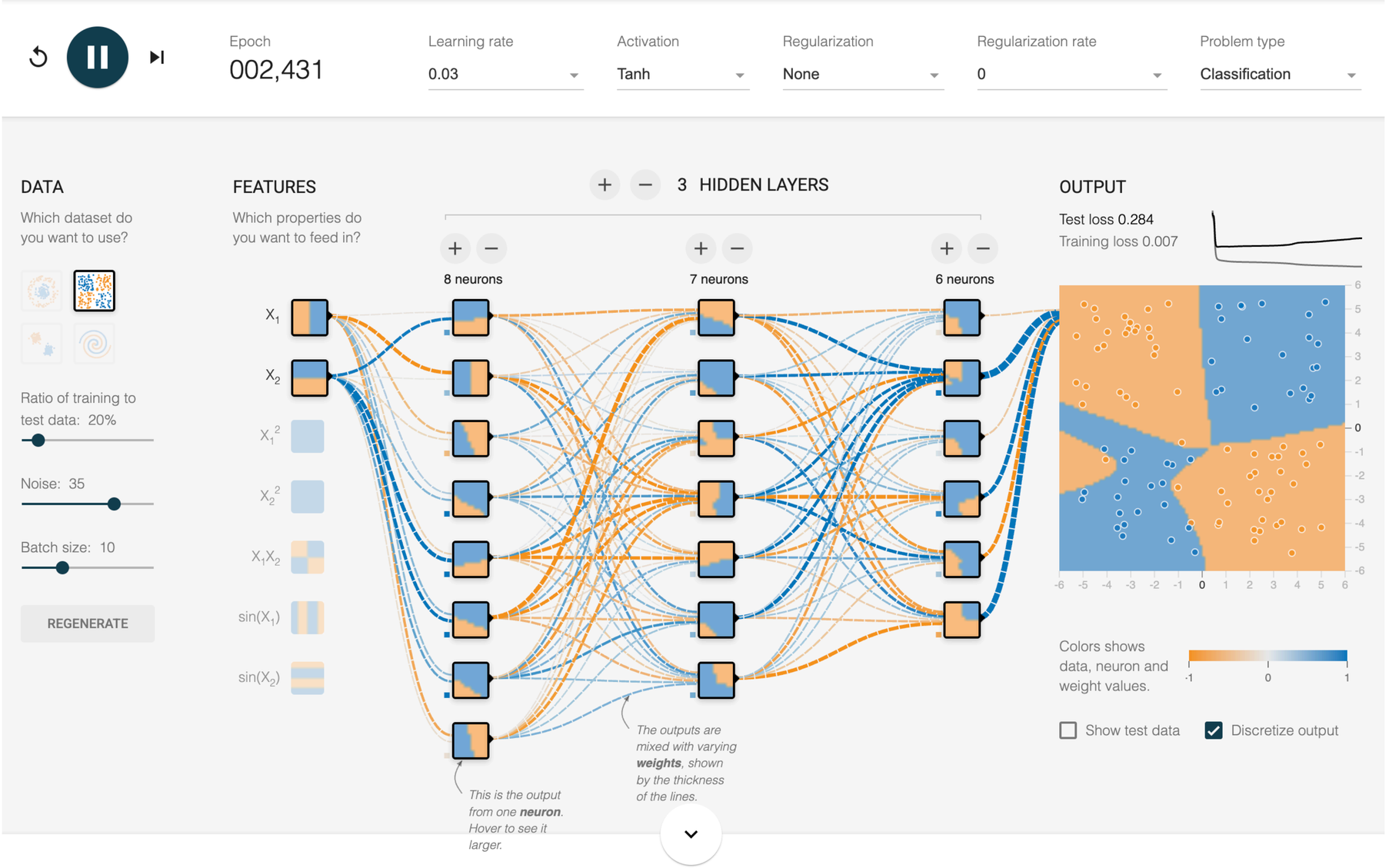
As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

Training loop (pytorch)

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

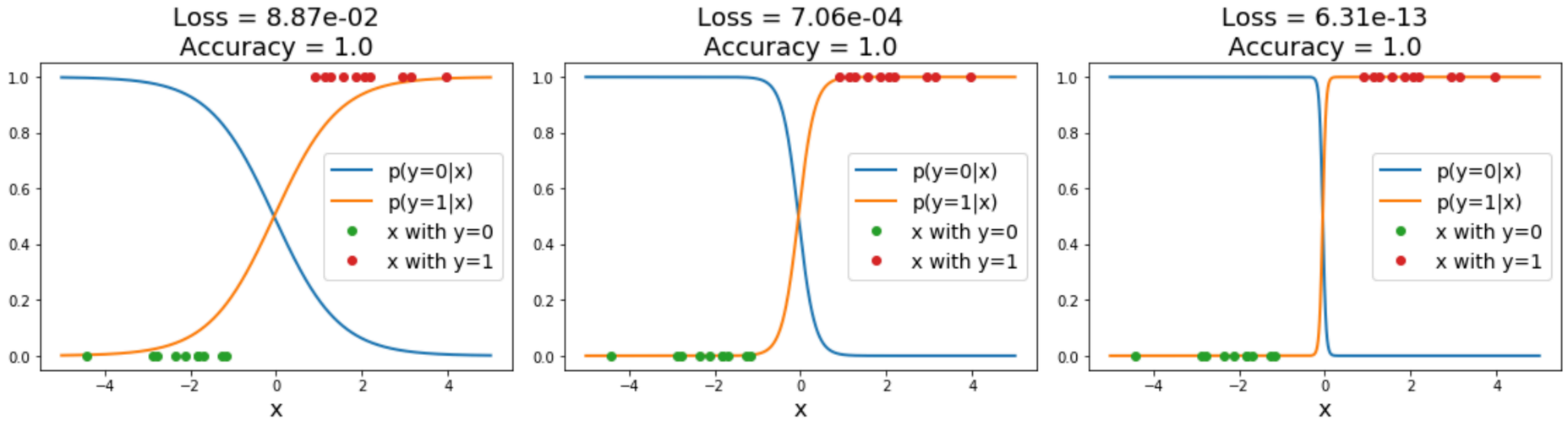
A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

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/

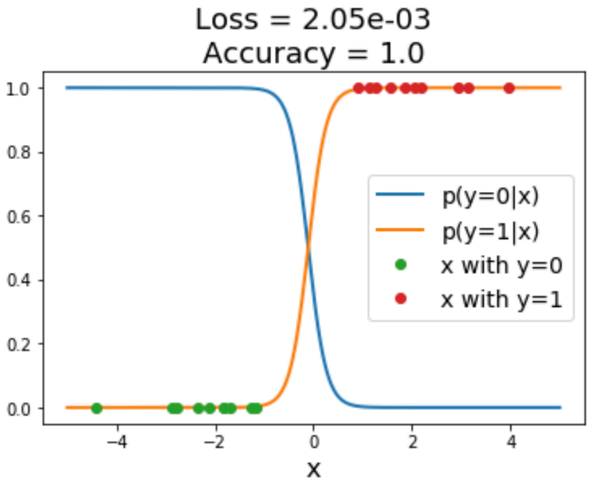
Regularization: Prefer Simpler Models

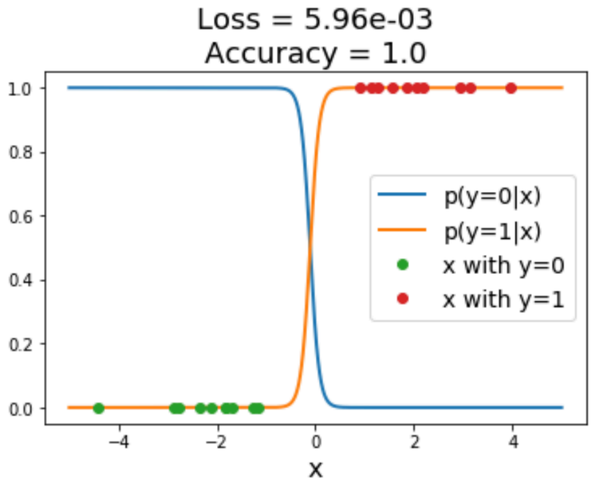
Example: Linear classifier with 1D inputs, 2 classes, softmax loss

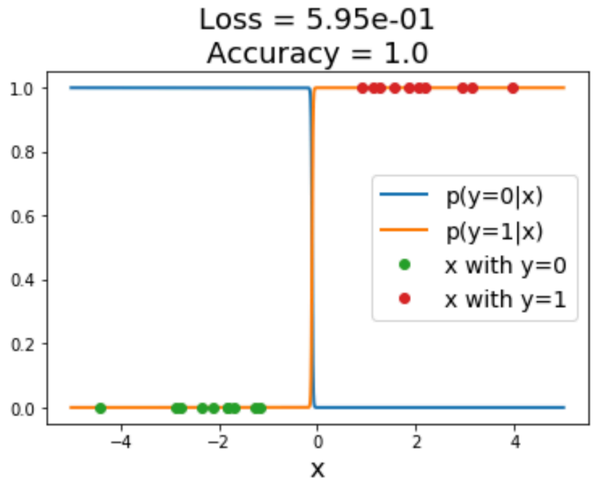
$$s_i = w_i x + b_i \qquad p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

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

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/

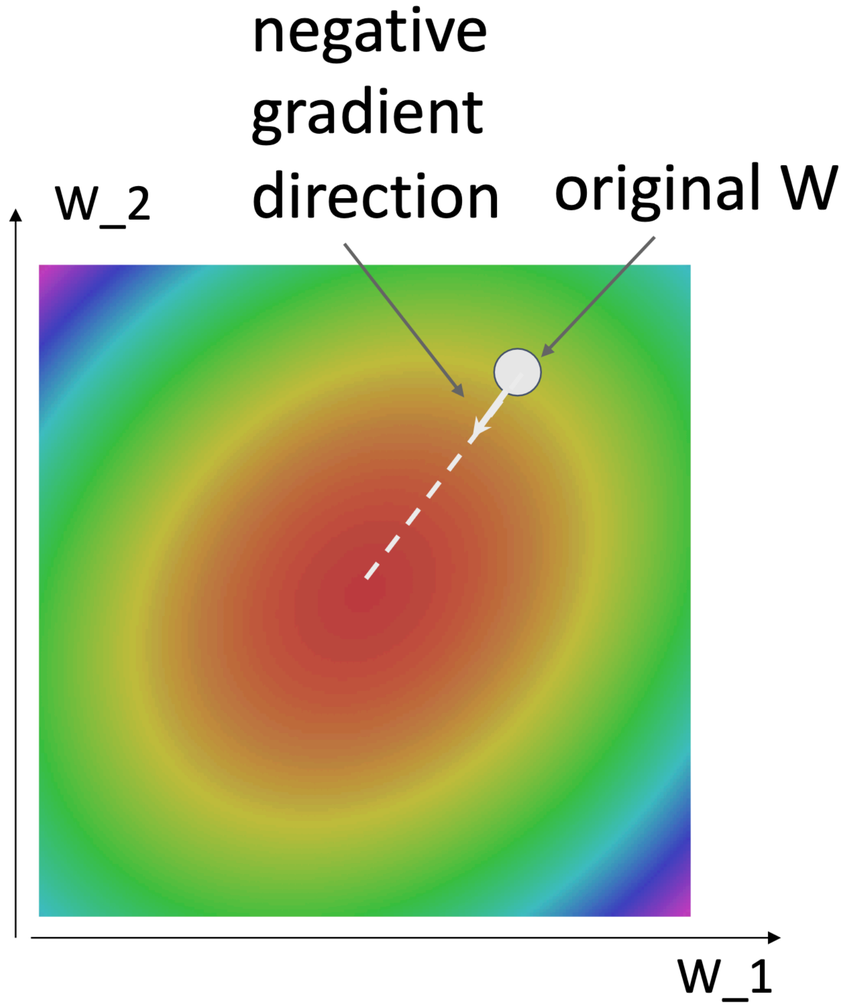
Gradient Descent

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/

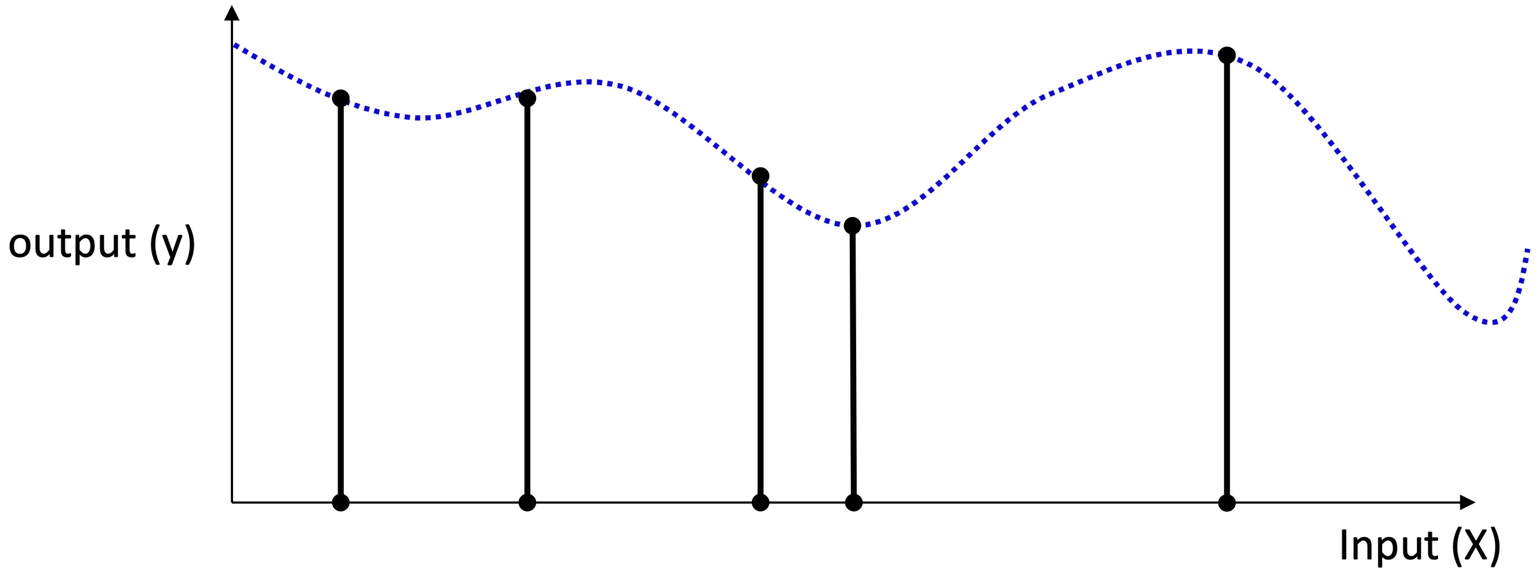
Incremental updates

- 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.

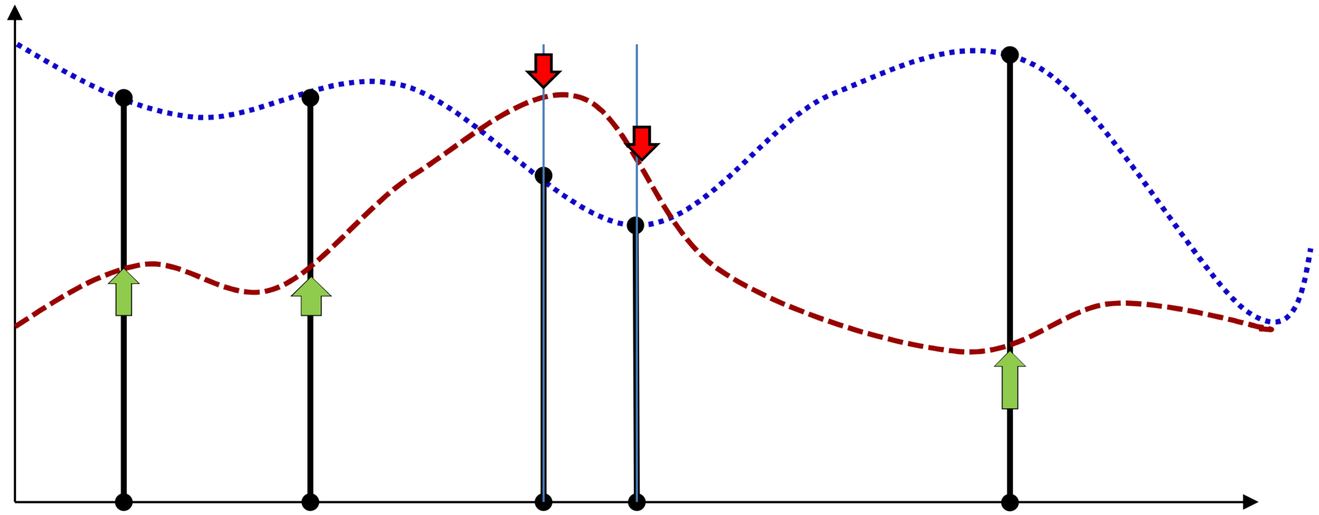
- Such behavior is implemented in the **Stochastic Gradient Descent** algorithm which updates the parameters based on the gradient evaluated on a small subset of the training data (perhaps on one sample!)

The training formulation



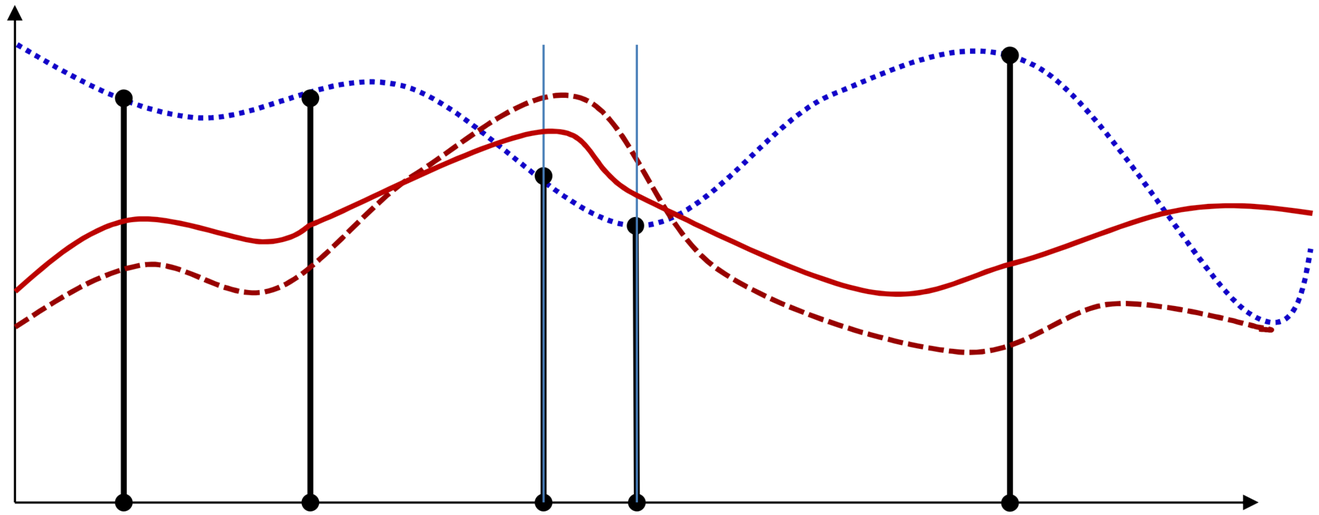
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



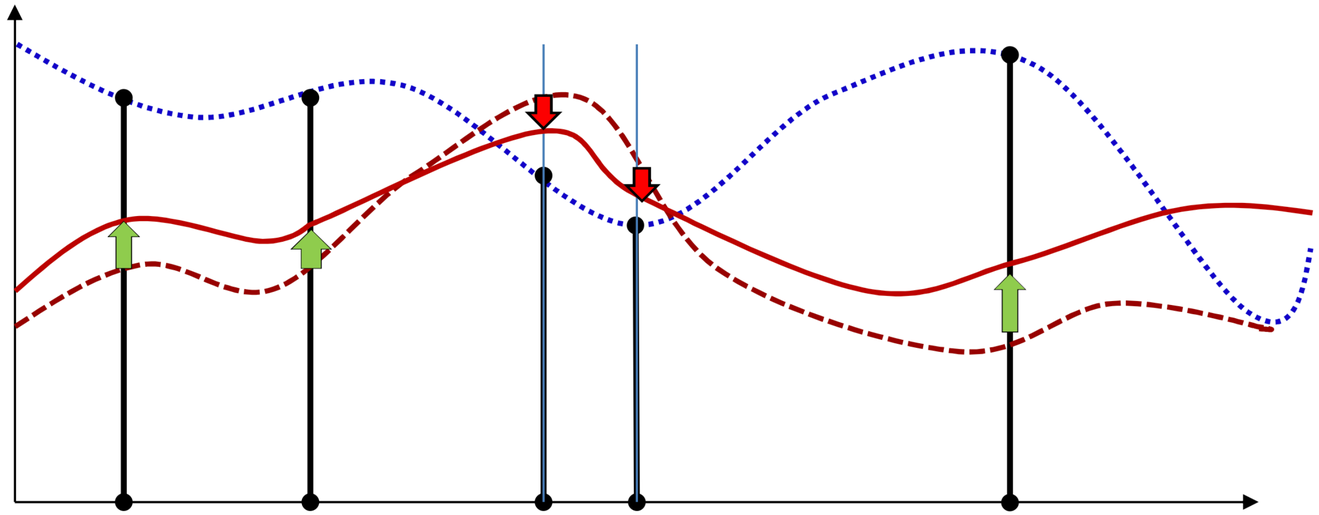
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



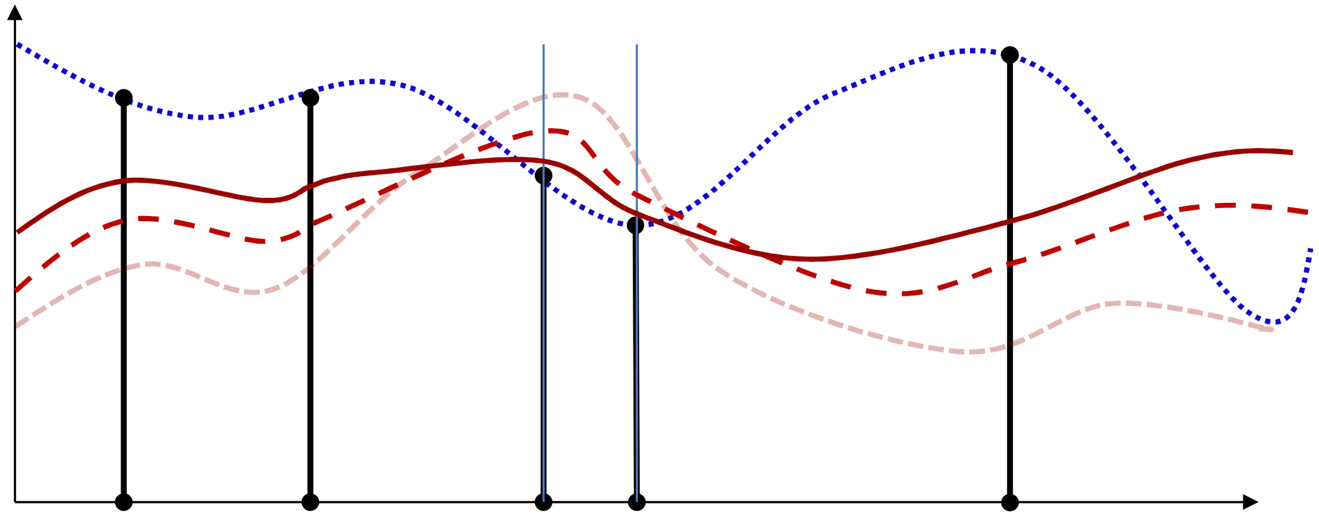
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



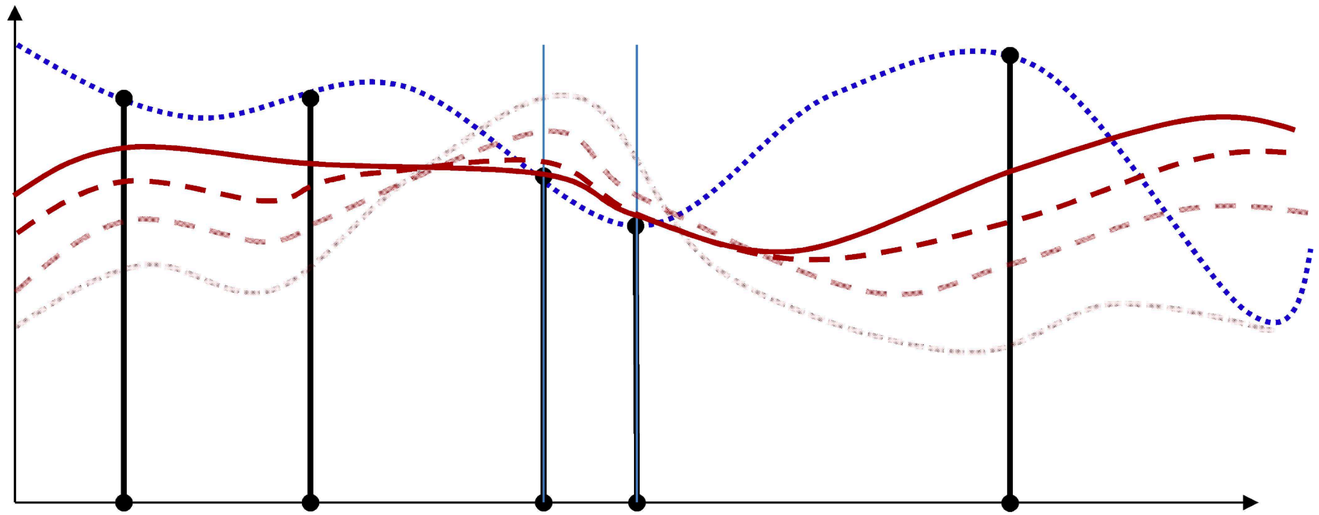
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



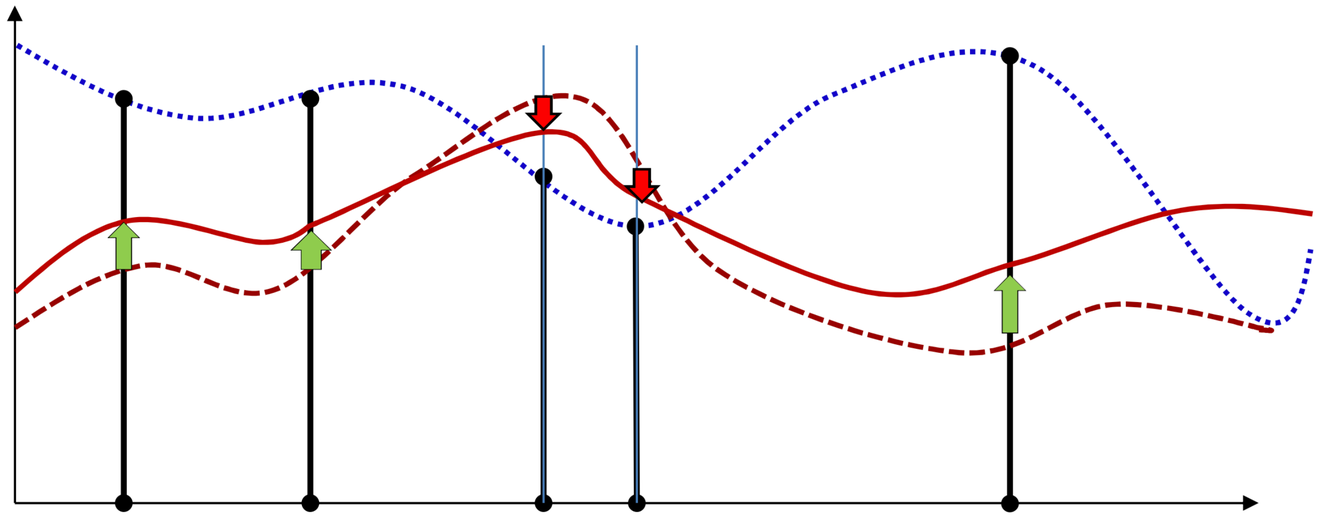
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Gradient descent



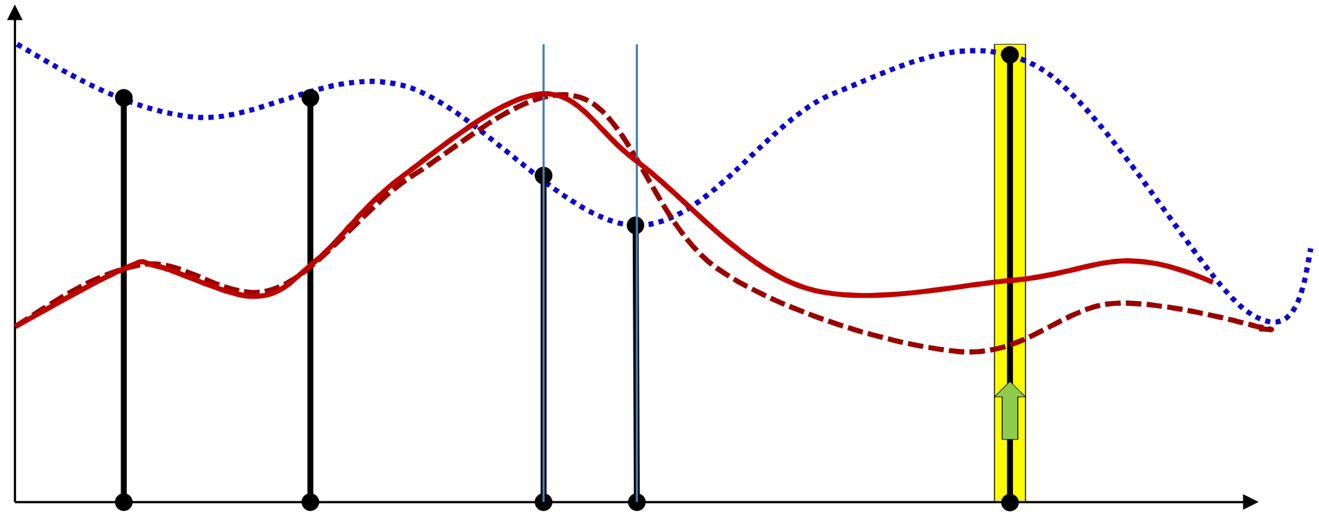
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

Effect of number of samples



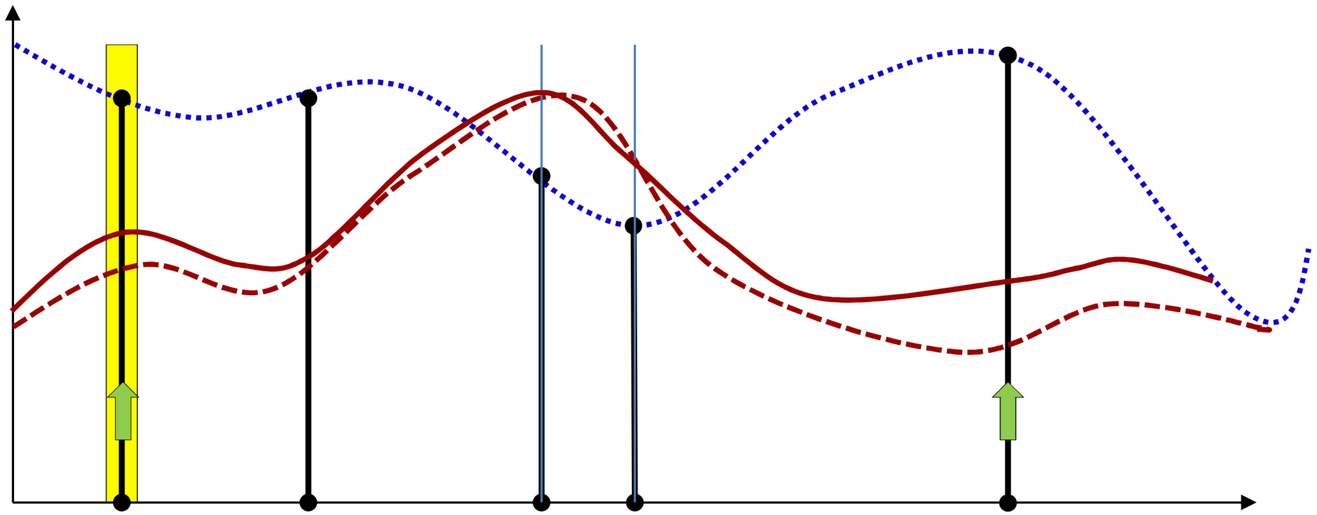
- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
  - We must process *all* training points before making a single adjustment
  - “Batch” update

Alternative: Incremental update



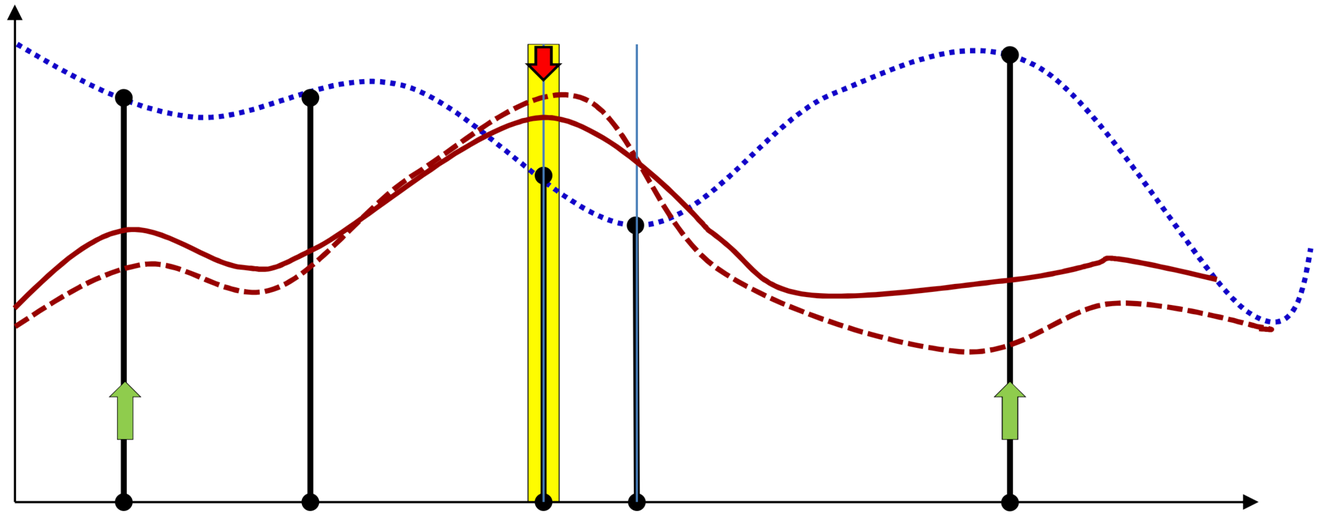
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



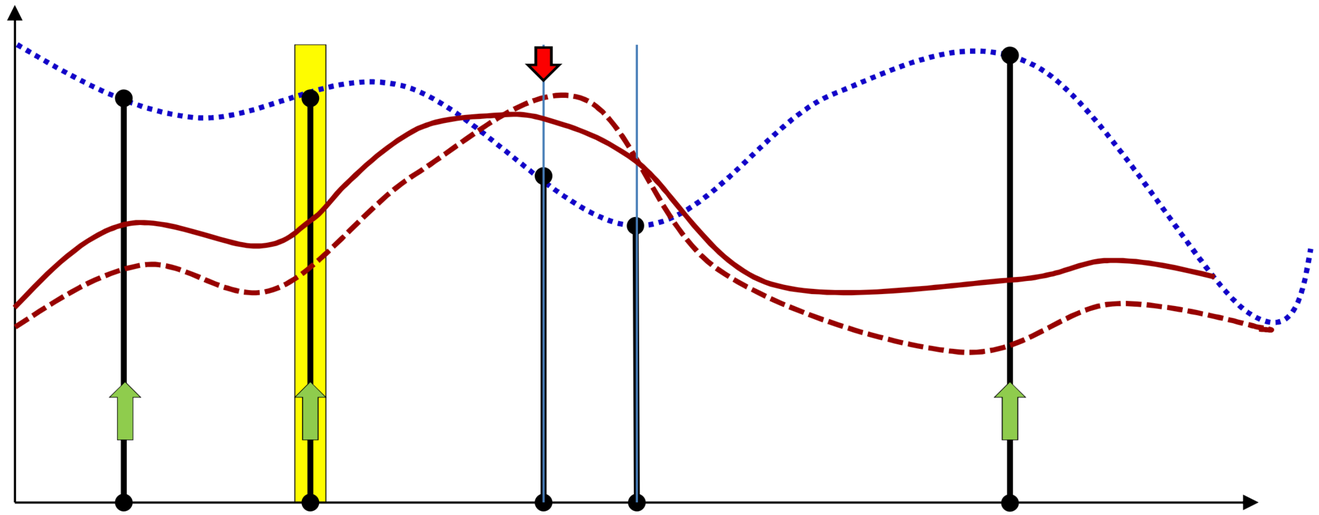
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



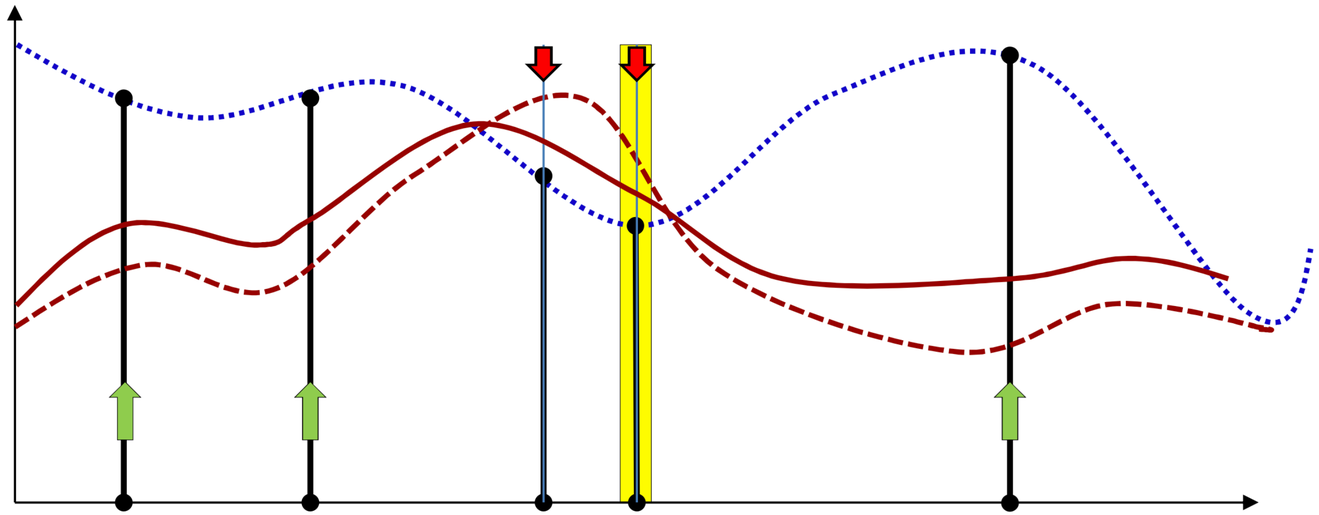
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

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}")
```

Inference loop (pytorch)

How to calucate [validation / testing] [loss / accuracy]

- switch to eval mode
  `model.eval()`
- calculate **mean** loss for all samples
  - no need to shuffle
- do not calculate gradients
  `with torch.no_grad():`
- use `___.item()`

```python
def evaluate(model, loader: torch.utils.data.DataLoader, device: torch.device):
    model.eval()
    loss = 0.0
    correct = 0
    # define loss function with reduction="mean"
    loss_fn = nn.CrossEntropyLoss(reduction="sum")
    with torch.no_grad():  # disable gradient calculation
        for data, target in loader:  # loop over the data
            data, target = data.to(device), target.to(device)  # move data to device
            output = model(data)  # forward pass
            loss += loss_fn(output, target).item()  # compute loss
            pred = output.argmax(
                dim=1, keepdim=True
            )  # get the index of the max log-probability
            correct += (
                pred.eq(target.view_as(pred)).sum().item()
            )  # count correct predictions
    loss /= len(loader.dataset)  # compute average loss
    accuracy = correct / len(loader.dataset)  # compute accuracy
    return loss, accuracy
```

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
  - $x' = \frac{x - \mu}{\sigma}$

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

Weights initialization

- You can build better deep learning models that train much faster by using the correct weight initialization techniques.
- Initial weights have this impact because the loss surface of a deep neural network is a complex, high-dimensional, and non-convex landscape with many local minima.
- The point where the weights start on this loss surface determines the local minimum to which they converge; the better the initialization, the better the model.

https://www.pinecone.io/learn/weight-initialization/

Weights initialization

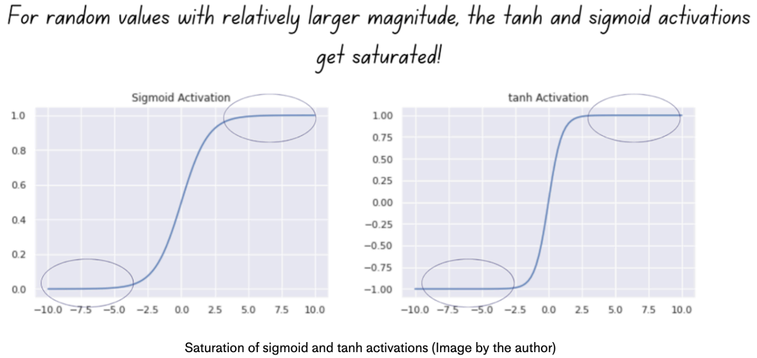
Trivial initialization

- Trivial initializations (all weights = 0 or constant) do not work!
  - Such weights are initially equal and receive the same update at each step. The neurons, therefore, evolve **symmetrically** as the training proceeds.
  - All the neurons in such a layer learn the “same” thing. Such a model performs poorly in practice.

https://www.pinecone.io/learn/weight-initialization/

Weights initialization

Random initialization



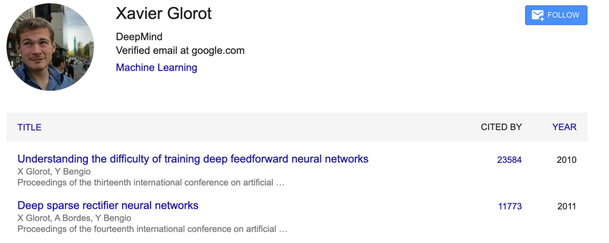
Saturation of sigmoid and tanh activations (Image by the author)

- $N(0,\sigma^2)$ or $U(-a, a)$
- When the magnitude of *activations* is **small**, the **gradients are vanishingly** small, and the neurons do not learn anything!
- When the *weights* have a **large** magnitude, the sigmoid and tanh activation functions take on values very close to saturation.
- When the activations become saturated, the gradients move close to zero during backpropagation.

https://www.pinecone.io/learn/weight-initialization/

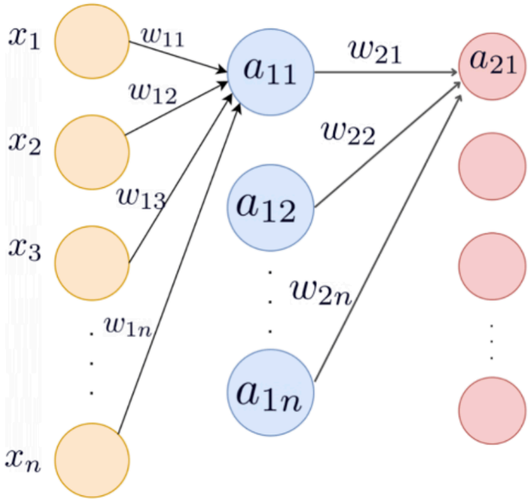
Weights initialization

Xavier Glorot initialization

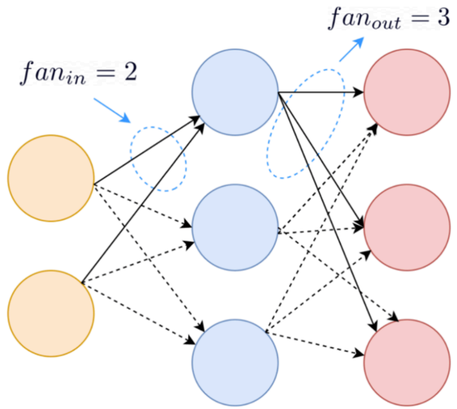


X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks (2010), AISTATS 2010

Normal distribution $N(0,\sigma^2)$



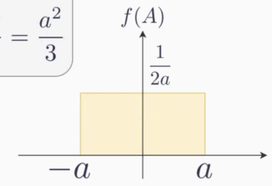
$$a_{21} = \sum_{i=1}^{n} a_{1i} w_{2i}$$



Explaining fan_in and fan_out (Image by the author)

Uniform distribution

$$\mathbb{E}(A) = 0$$
$$Var(A) = \frac{(2a)^2}{12} = \frac{a^2}{3}$$



$$Var(a_{11}) = n.Var(w)Var(x)$$

$$Var(a_{21}) = n^2[Var(w)]^2 Var(x)$$

$$Var(a_{ki}) = [n.Var(w)]^k Var(x)$$

$$n.Var(w) = 1$$

$$Var(w) = \frac{1}{n}$$

$$\frac{fan_{in} + fan_{out}}{2}.Var(w) = 1$$

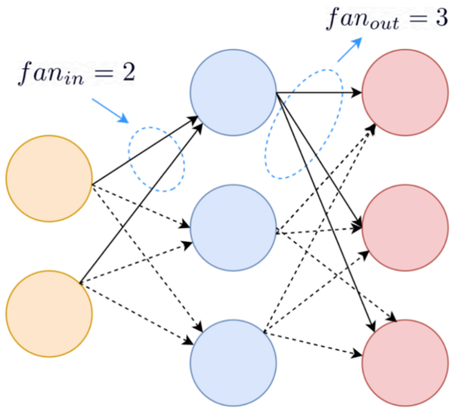
$$Var(w) = \frac{2}{fan_{in} + fan_{out}}$$

$$w \in \mathcal{U}\left[-\sqrt{\frac{6}{fan_{in} + fan_{out}}}, \sqrt{\frac{6}{fan_{in} + fan_{out}}}\right]$$

https://www.pinecone.io/learn/weight-initialization/

Weights initialization

Xavier Glorot initialization



Explaining fan_in and fan_out (Image by the author)

$$\frac{fan_{in} + fan_{out}}{2} . Var(w) = 1$$

$$Var(w) = \frac{2}{fan_{in} + fan_{out}}$$

Forward Pass Equation

$$Var[z_k^{i+1}] = Var[f(z^i W_{.,k}^i + b_k^i)] \Rightarrow Var[W^i] = \frac{1}{n^i}$$

Backward Pass Equation

$$Var\left[\frac{\partial L}{\partial s_k^i}\right] = Var\left[\sum_{j=1}^{n^{i+2}} W_{j,k}^{i+1} \frac{\partial L}{\partial s_k^{j+1}} f'(s_k^i)\right] \Rightarrow Var[W^i] = \frac{1}{n^{i+1}}$$

Weight Distributions

$$Var[W^i] = \frac{2}{n^i + n^{i+1}} \Rightarrow \begin{cases} W^i \sim N(0, \sigma^2) \Rightarrow \sigma = \sqrt{\frac{2}{n^i + n^{i+1}}} \\ W^i \sim U(-a, a) \Rightarrow a = \sqrt{\frac{6}{n^i + n^{i+1}}} \end{cases}$$

https://medium.com/data-science/xavier-glorot-initialization-in-neural-networks-math-proof-4682bf5c6ec3

Weights initialization

He initialization

K. Kumar, On weight initialization in deep neural networks (2017)

He et al., Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (2015), CVPR 2015

- It was found that Glorot initialization did not work for networks that used ReLU activations as the backflow of gradients was impacted.

$$\frac{\frac{fan_{in}+fan_{out}}{2}}{2}.Var(w)=1$$

$$Var(w)=\frac{4}{fan_{in}+fan_{out}}$$

$$w \in \mathcal{U}\left[-\sqrt{\frac{12}{fan_{in}+fan_{out}}}, \sqrt{\frac{12}{fan_{in}+fan_{out}}}\right]$$

Forward Pass Equation

$$Var[y_k] = Var\left[\sum_{i=1}^{n_k} x_k^i W_k^{ij} + b_k^j\right] \Rightarrow Var[W_k] = \frac{2}{n_k}$$

Backward Pass Equation

$$Var[\Delta y_k] = Var\left[\Delta x_{k+1} f'(y_k)\right] \Rightarrow Var[W_k] = \frac{2}{n_k}$$

Weight Distributions

$$Var[W_k] = \frac{2}{n_k} \Rightarrow \begin{cases} W_k \sim N(0,\sigma^2) \Rightarrow \sigma = \sqrt{\frac{2}{n_k}} \\ W_k \sim U(-a,a) \Rightarrow a = \sqrt{\frac{6}{n_k}} \end{cases}$$

https://towardsdatascience.com/kaiming-he-initialization-in-neural-networks-math-proof-73b9a0d845c4/

Weights initialization

Summing up

- Initializing the weights to zero or a constant value leads to the symmetry-breaking problem. This problem stems from the weights receiving the same updates at each step and updating symmetrically as the training proceeds.
- Initializing the weights to small random values leads to the problem of vanishing gradients. This is because the gradients flowing into a particular neuron are proportional to the activation that it receives. On the other hand, initializing the weights to large random values causes the activations to get saturated, resulting in vanishing gradients during backpropagation.
- To prevent the weights from being drawn from a distribution whose variance is neither too large nor too small, the variance of the distribution must be approximately 1.
- Xavier or Glorot initialization works well for networks using activations with zero mean, such as the sigmoid and tanh functions.
- When using ReLU activation that does not have zero mean, it's recommended to use the He initialization.

Additional notes: https://www.pinecone.io/learn/weight-initialization/

Demo code

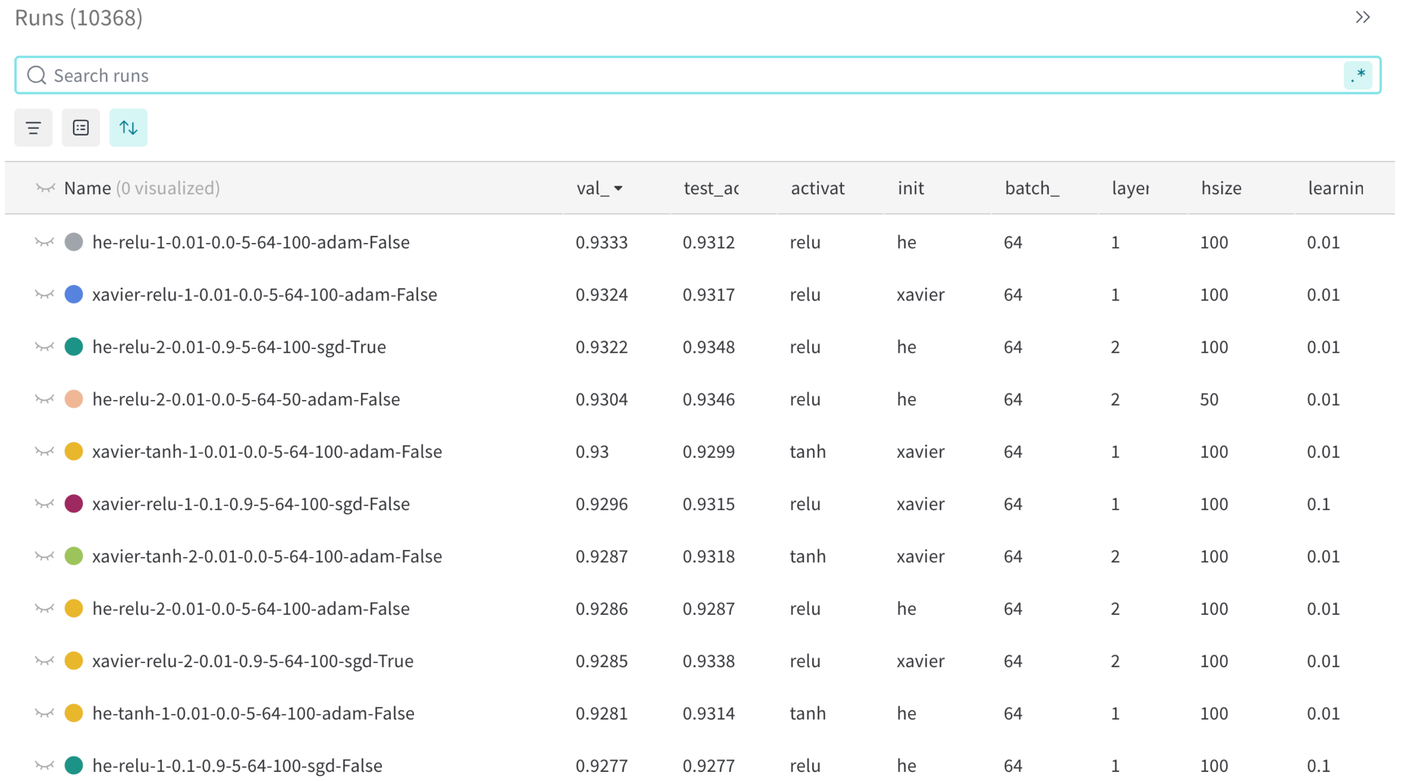
LinearLayer

- 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)
```

Torch

- see project in W&B:
  - https://wandb.ai/rno/mlp-mnist-200-steps



Runs (10368)

Search runs

Name (0 visualized)	val_	test_ac	activat	init	batch_	layer	hsize	learnin
he-relu-1-0.01-0.0-5-64-100-adam-False	0.9333	0.9312	relu	he	64	1	100	0.01
xavier-relu-1-0.01-0.0-5-64-100-adam-False	0.9324	0.9317	relu	xavier	64	1	100	0.01
he-relu-2-0.01-0.9-5-64-100-sgd-True	0.9322	0.9348	relu	he	64	2	100	0.01
he-relu-2-0.01-0.0-5-64-50-adam-False	0.9304	0.9346	relu	he	64	2	50	0.01
xavier-tanh-1-0.01-0.0-5-64-100-adam-False	0.93	0.9299	tanh	xavier	64	1	100	0.01
xavier-relu-1-0.1-0.9-5-64-100-sgd-False	0.9296	0.9315	relu	xavier	64	1	100	0.1
xavier-tanh-2-0.01-0.0-5-64-100-adam-False	0.9287	0.9318	tanh	xavier	64	2	100	0.01
he-relu-2-0.01-0.0-5-64-100-adam-False	0.9286	0.9287	relu	he	64	2	100	0.01
xavier-relu-2-0.01-0.9-5-64-100-sgd-True	0.9285	0.9338	relu	xavier	64	2	100	0.01
he-tanh-1-0.01-0.0-5-64-100-adam-False	0.9281	0.9314	tanh	he	64	1	100	0.01
he-relu-1-0.1-0.9-5-64-100-sgd-False	0.9277	0.9277	relu	he	64	1	100	0.1