DSC 140B
Representation Learning

Lecture 14 | Part 1

Stochastic Gradient Descent

Gradient Descent for Minimizing Risk

- In ML, we often want to minimize a **risk function**:

$$R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

Observation

- The gradient of the risk is the average of the gradient of the losses:

$$\vec{\nabla} R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- The averaging is over **all training points**.

- This can take a long time when n is large.[1]

[1]Trivia: this usually takes $\Theta(nd)$ time.

Idea

- The (full) gradient of the risk uses all of the training data:

$$\vec{\nabla} R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- **Idea:** instead of using all n training points, randomly choose a smaller set, B:

$$\vec{\nabla} R(\vec{w}) \approx \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

Stochastic Gradient

- The smaller set B is called a **mini-batch**.
- We now compute a **stochastic gradient**:

$$\vec{\nabla} R(\vec{w}) \approx \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- "Stochastic," because it is a random.

Stochastic Gradient

$$\vec{\nabla} R(\vec{w}) \approx \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- The stochastic gradient is an **approximation** of the full gradient.
- When $|B| \ll n$, it is **much faster** to compute.
- But the approximation is **noisy**.

Stochastic Gradient Descent for ERM

To minimize empirical risk $R(\vec{w})$:

- Pick starting weights $\vec{w}^{(0)}$, learning rate $\eta > 0$, batch size m.
- Until convergence, repeat:
  - **Randomly sample** a batch B of m training data points.
  - **Compute stochastic gradient:**

$$\vec{g} = \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

  - **Update:** $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \eta \vec{g}$
- When converged, return $\vec{w}^{(t)}$.

Note

- A **new batch** should be randomly sampled on each iteration!
- This way, the entire training set is used over time.
- Size of batch should be **small** compared to n.
  - Think: $m = 64$, $m = 32$, or even $m = 1$.

Live QA

Exercise

Suppose we set $|B| = n$ and sample without replacement. Then we run SGD.

True or False: this is actually just gradient descent.

Example: Least Squares

- We can use SGD to perform least squares regression.

- Need to compute the gradient of the square loss:

$$\ell_{\text{sq}}(H(\vec{x}^{(i)}; \vec{w}), y_i) = \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i\right)^2$$

Example: Least Squares

- The gradient of the square loss of a linear predictor is:

$$
\begin{aligned}
&\vec{\nabla}\ell_{\text{sq}}(H(\vec{x}^{(i)}; \vec{w}), y_i) \\
&\qquad = \vec{\nabla}\left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i\right)^2 \\
&\qquad = 2\left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i\right)\vec{\nabla}\left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i\right) \\
&\qquad = 2\left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i\right)\text{Aug}(\vec{x}^{(i)})
\end{aligned}
$$

Example: Least Squares

- Therefore, on each step we compute the stochastic gradient:

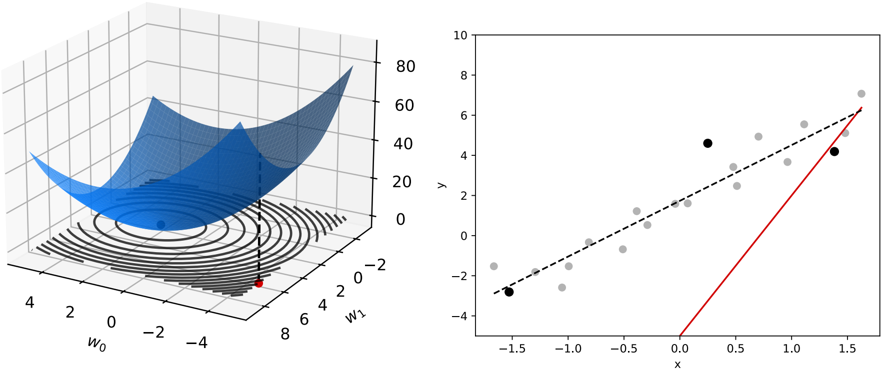
$m = |B|$

$$\vec{g} = \frac{2}{m} \sum_{i \in B} \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i \right) \text{Aug}(\vec{x}^{(i)})$$

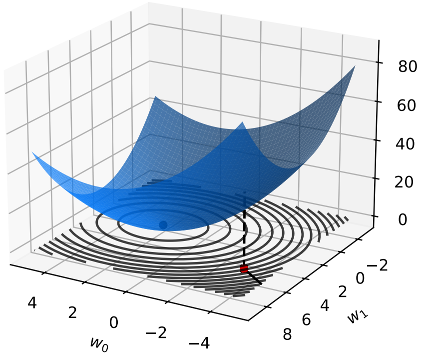
- The update rule is:

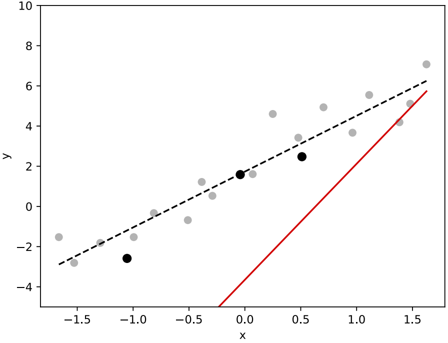
$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \eta \vec{g}$$

Example: SGD

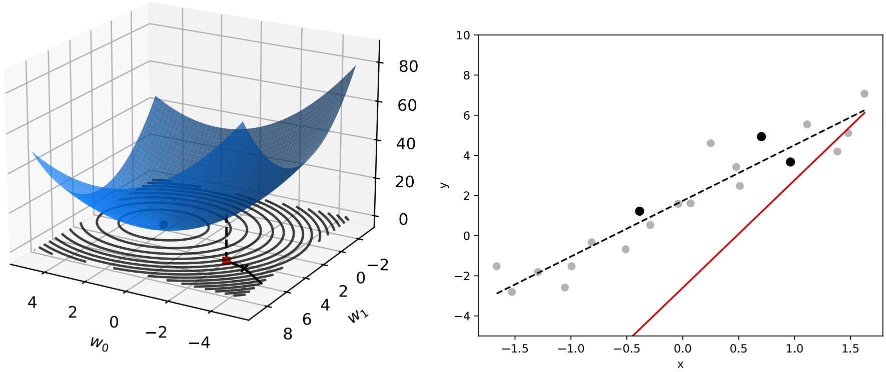


Example: SGD

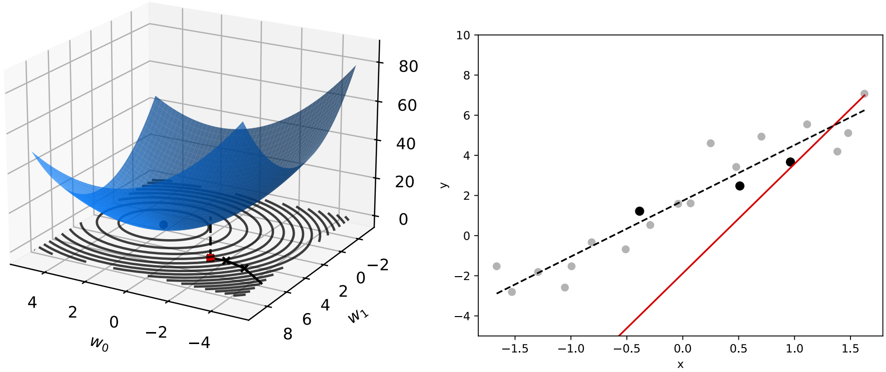




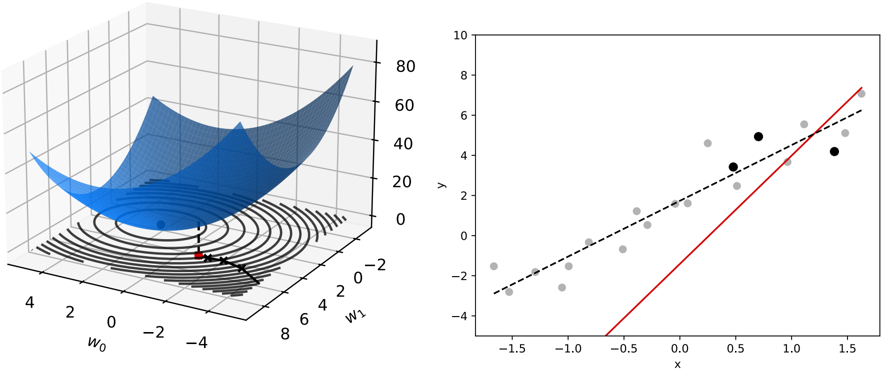
Example: SGD



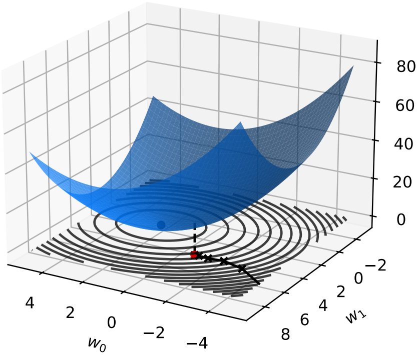
Example: SGD

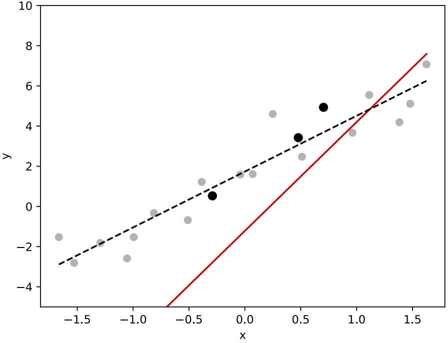


Example: SGD

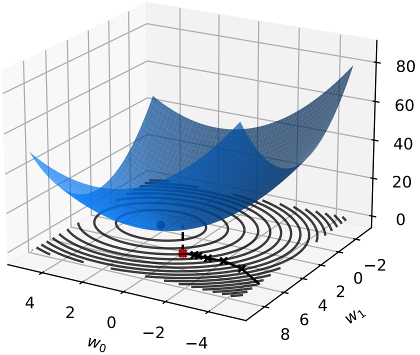


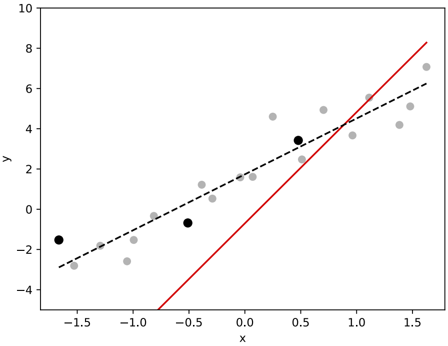
Example: SGD



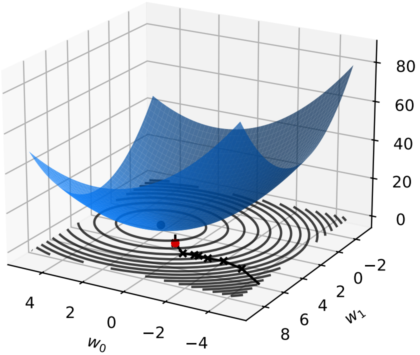


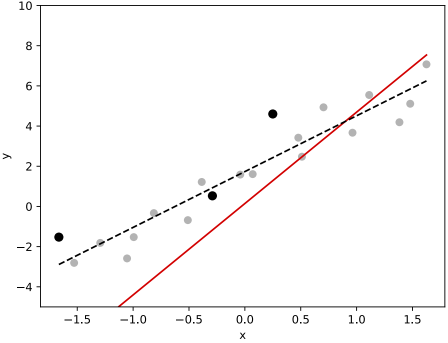
Example: SGD



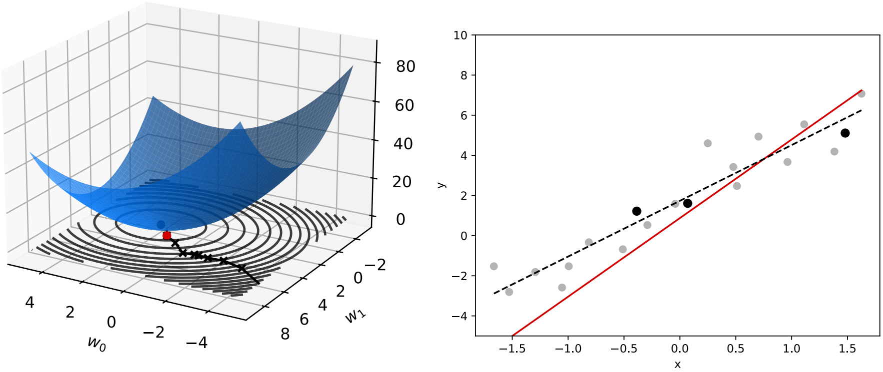


Example: SGD

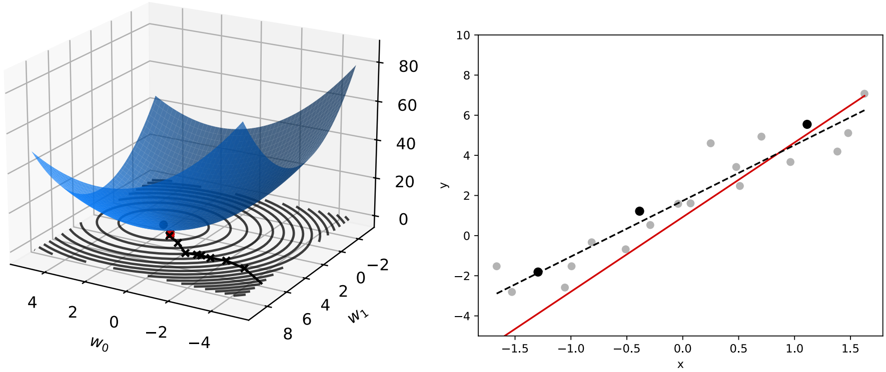




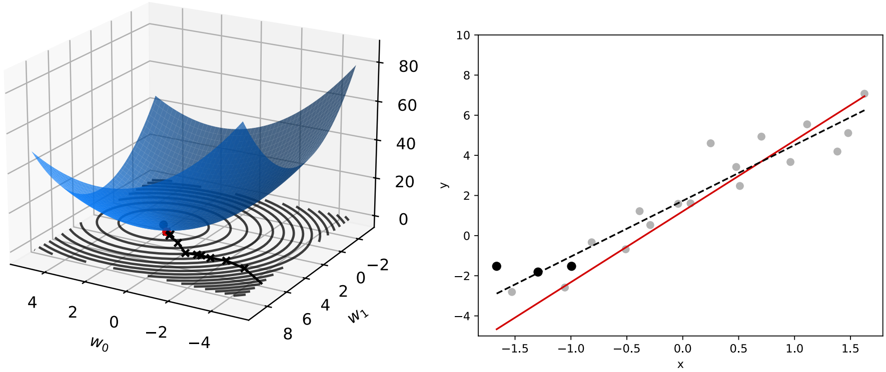
Example: SGD



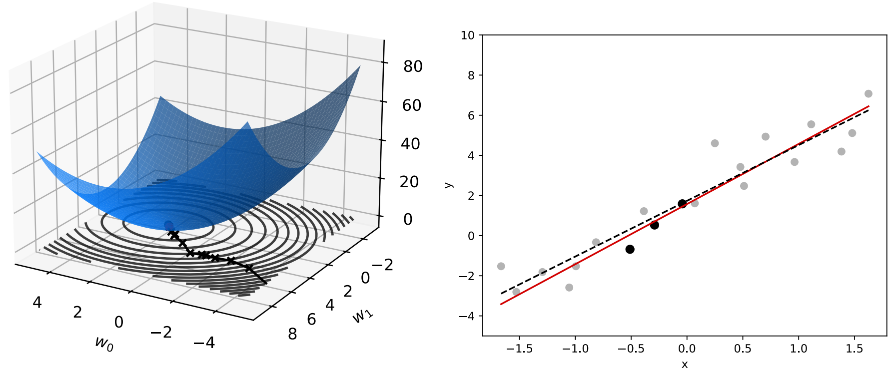
Example: SGD



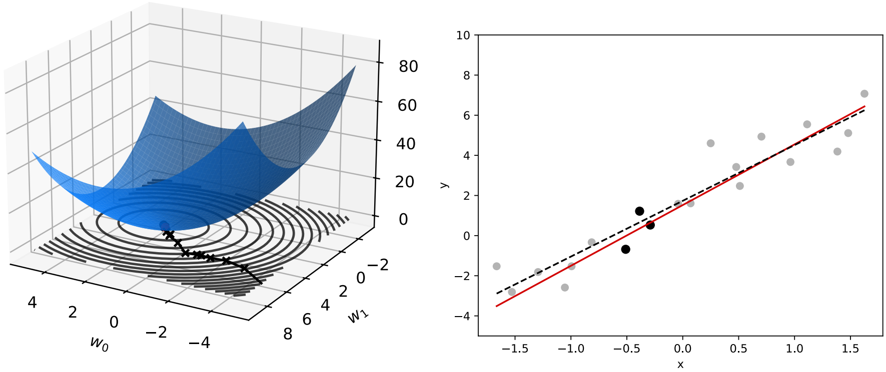
Example: SGD



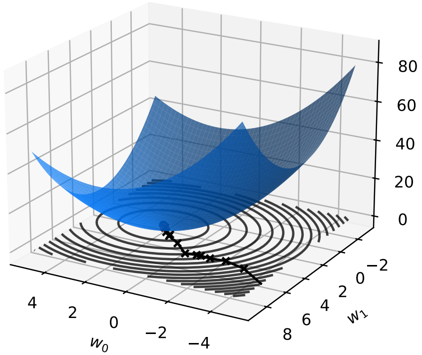
Example: SGD

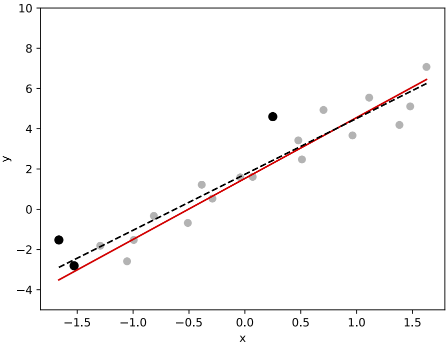


Example: SGD

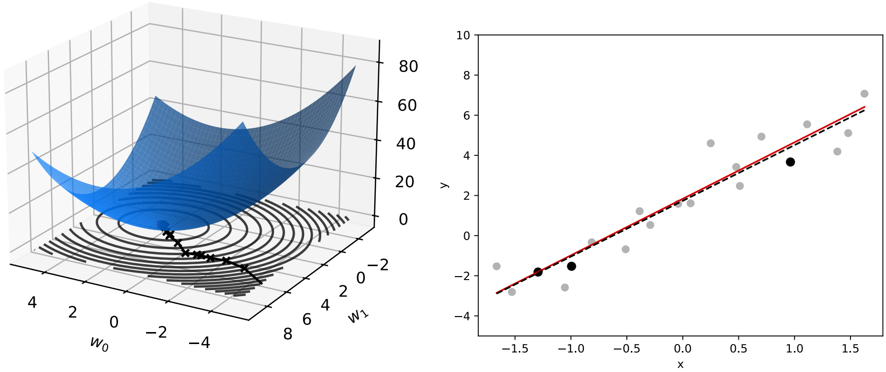


Example: SGD

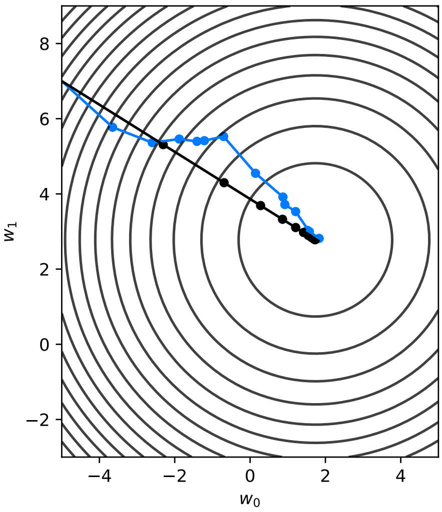




Example: SGD



SGD vs. GD



Tradeoffs

- In each step of GD, move in the “best” direction.
  - But **slowly!**
- In each step of SGD, move in a “good” direction.
  - But **quickly!**
- SGD may take more steps to converge, but can be faster overall.

Example

- Suppose you're doing **least squares regression** on a medium-to-large data set.
- Say, n = 200,000 examples, d = 5,000 features.
- Encoded as 64 bit floats, X is 8 GB.
  - Fits in your laptop's memory, but barely.
- **Example:** predict sentiment from text.

Timing

- Solving the normal equations took **30.7 seconds**.
- Gradient descent took **8.6 seconds**.
  - 14 iterations, ≈ 0.6 seconds per iteration.
- Stochastic gradient descent takes **3 seconds**.
  - Batch size $m = 16$.
  - 13,900 iterations, ≈ 0.0002 seconds per iteration.

Aside: Terminology

- Some people say “stochastic gradient descent” only when batch size is 1.
- They say “mini-batch gradient descent” for larger batch sizes.
- **In this class**: we’ll use “SGD” for any batch size, as long as it’s chosen randomly.

Aside: A Popular Variant

- One variant of SGD uses **epochs**.
- During each epoch, we:
  - Randomly shuffle the training data.
  - Divide the training data into n/m mini-batches.
  - Perform one step for each mini-batch.

Usefulness of SGD

- SGD **enables** learning on **massive** data sets.
  - Billions of training examples, or more.

- Useful even when exact solutions available.
  - E.g., least squares regression / classification.

SGD in PyTorch

- PyTorch has a built-in implementation of SGD: `torch.optim.SGD`.
- By itself, it actually does (non-stochastic) gradient descent.
- You also need to use `torch.utils.data.DataLoader` to randomly sample mini-batches of data.
- See the demo notebook from last lecture for an example of how to do this.

DSC 140B
Representation Learning

Lecture 14 | Part 2

“Debugging” NN Training

Problem

- A lot more can **go wrong** when training NNs than when training linear models.

- Training them is more of an art than a science.
  - Involves a lot of **trial and error**.

- **Now:** how to debug when things go wrong.

Why is it so hard?

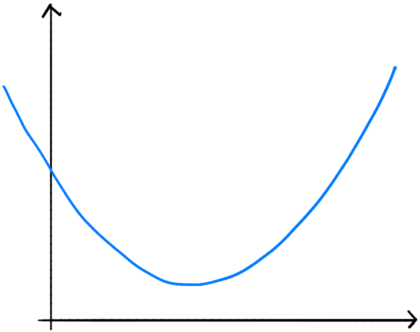
- **Reason 1**: There are a lot of architectural choices to make without much theory to guide you.
  - Number of layers?
  - Number of neurons per layer?
  - Activation function?
  - Initialization scheme?
  - Optimization algorithm?
  - Learning rate?
  - ...

- Cross-validation is expensive.

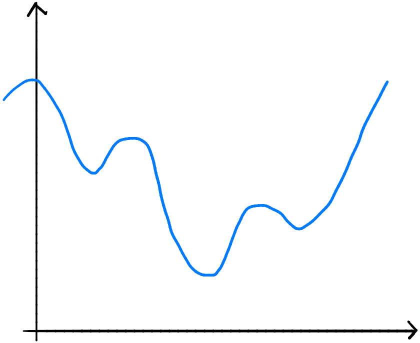
$R(\vec{w})$

Why is it so hard?

- **Reason 2**: The empirical risk is a **non-convex** function of **many** parameters.



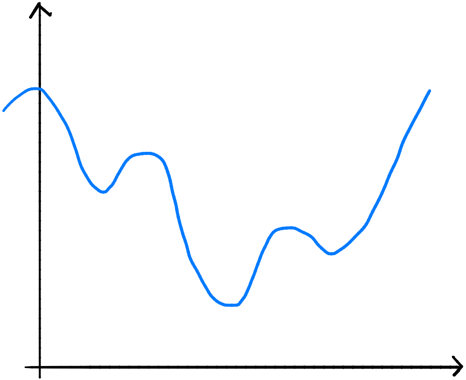
Convex



Non-Convex

Why is it so hard?

- As a result, it's easy to **get stuck** in a local minimum.



We will see...

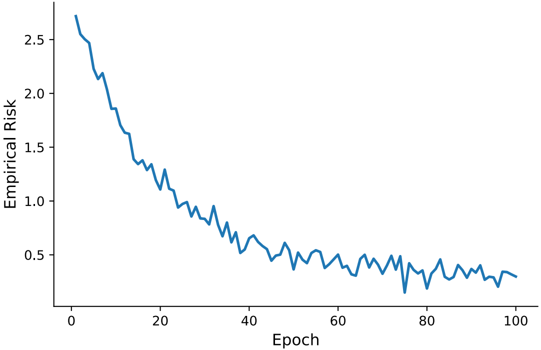
- Several ways in which training can fail.
- How to recognize these failure modes.
- How to fix them.
- **But:** there's not a lot of good theory to guide us, so we'll use **empirical evidence** and **intuition**.

First Diagnostic Tool

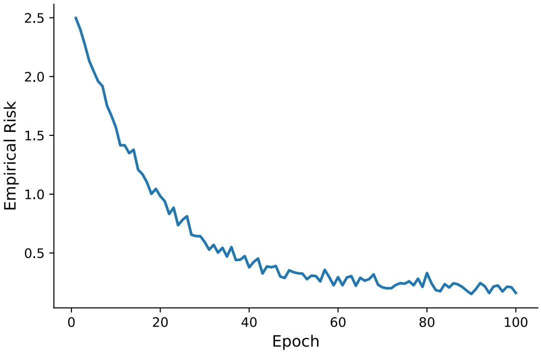
- Inspect the outputs.
  - If they're high dimensional, plot single coordinates or use histograms to visualize the dimensionality.

Second Diagnostic Tool

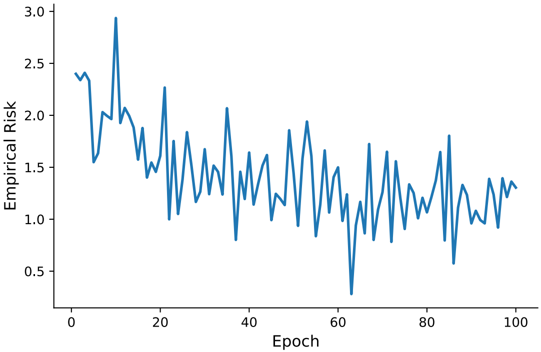
- When training, always record and plot the empirical risk at every epoch.
  - This is called the **training curve**.



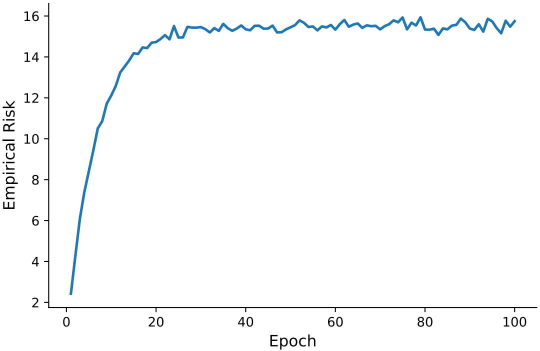
Good



Bad

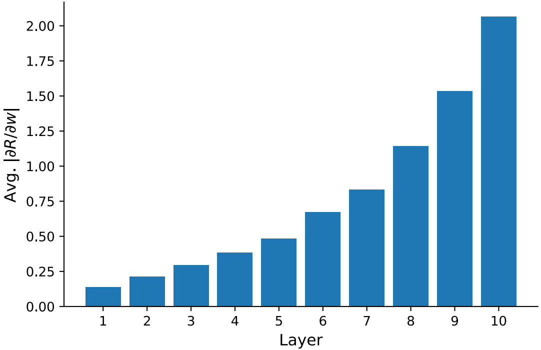


Very Bad



Third Diagnostic Tool

- It is also helpful to plot the average of the gradient entries at every layer.[2]



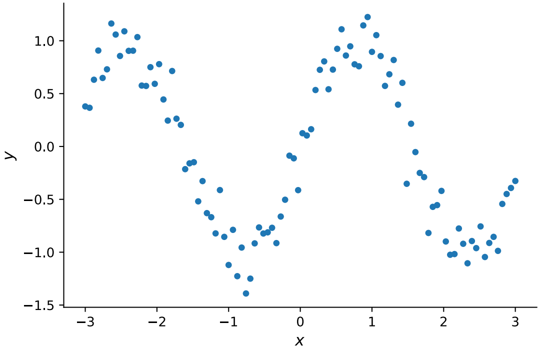
[2] More complicated, see PyTorch tutorial: `https://docs.pytorch.org/tutorials/intermediate/visualizing_gradients_tutorial.html`

What could go wrong?

- Your network is too shallow/narrow.
- Your network is too deep.
- You didn’t train long enough.
- Your learning rate is too high/low.
- Your initialization is bad.
- Your risk is ill-conditioned.
- Your activations are saturated.
- You are stuck in a local minimum.
- ...

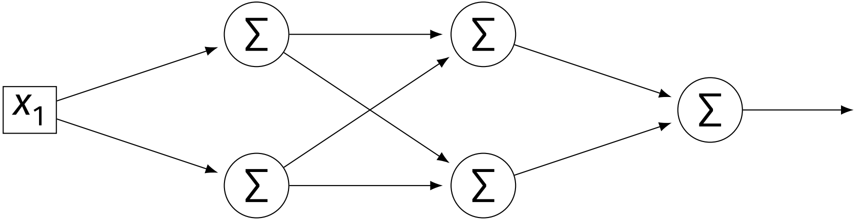
1) Network is too shallow/narrow

- Let's train a NN to perform regression on this data:

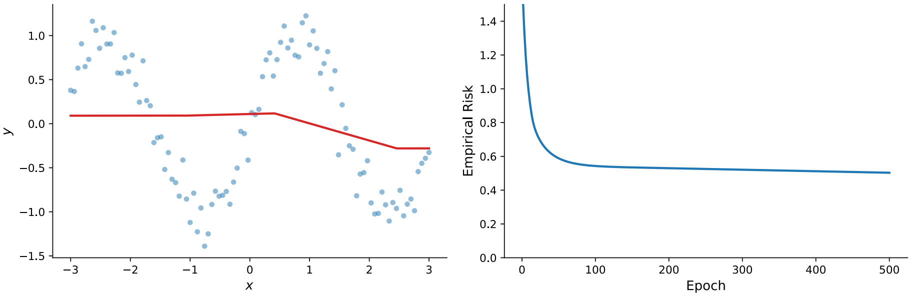


1) Network is too shallow/narrow

- Our network will be relatively small:



1) Network is too shallow/narrow

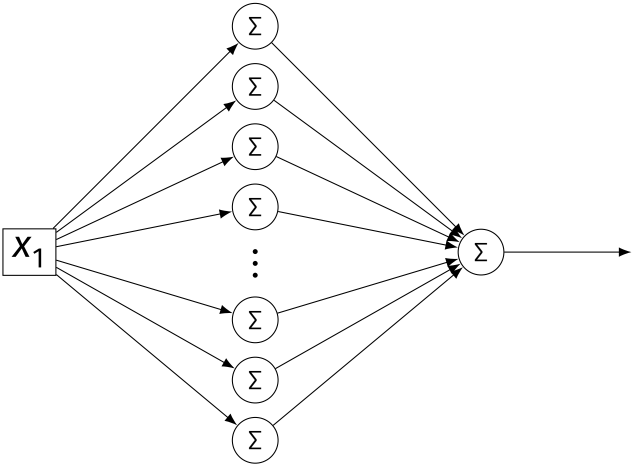


Fix

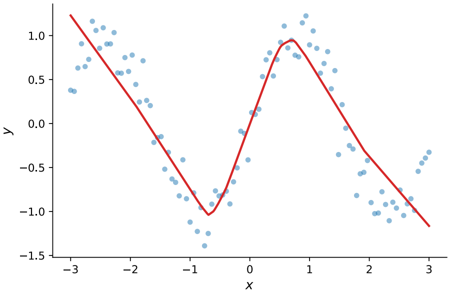
- The model is **underfitting** the data.
  - The model is not complex enough.
- We can fix this by making the network deeper and/or wider.
- In theory, a single hidden layer with enough neurons and ReLU can approximate any function.
- However, typically **prefer depth over width**.

Fix

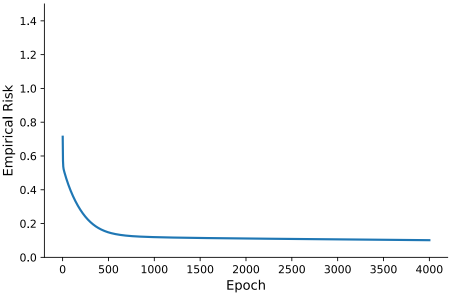
- Let's try one hidden layer with 40 neurons:



Fix



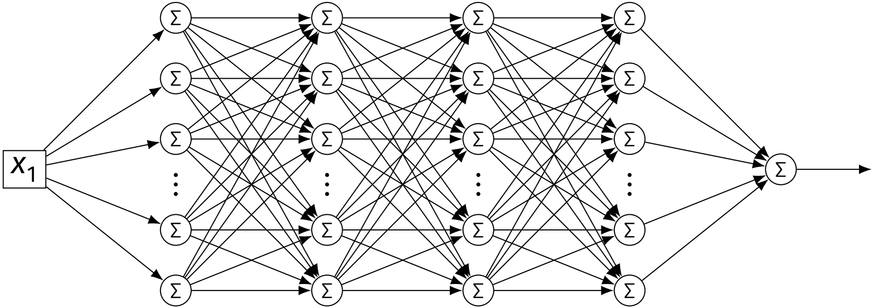
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
−3
−2
−1
0
1
2
3
x



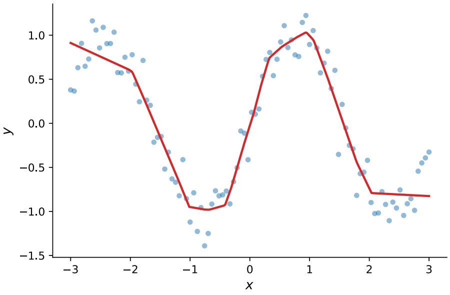
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Empirical Risk
0
500
1000
1500
2000
2500
3000
3500
4000
Epoch

Fix

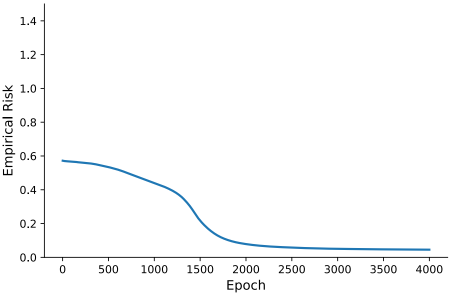
- Or use a deeper network:
  - 4 hidden layers, 10 neurons per layer.



Fix



1.0
0.5
0.0
−0.5
−1.0
−1.5
y
−3
−2
−1
0
1
2
3
x



1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Empirical Risk
0
500
1000
1500
2000
2500
3000
3500
4000
Epoch

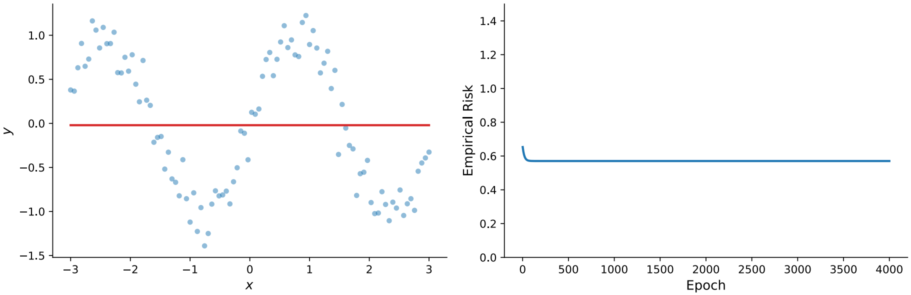
2) Network is too deep

- If depth is good, let's add more layers!
  - 50 hidden layers, 10 neurons per layer.

Exercise

What do you expect to see?

2) Network is too deep

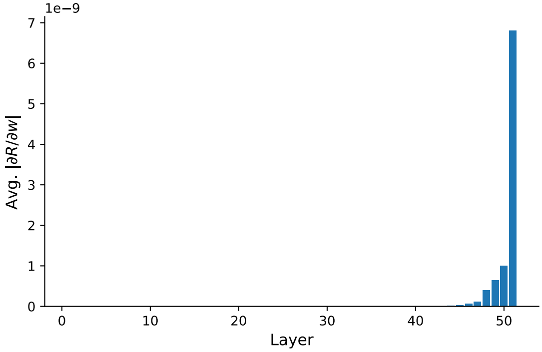


2) Network is too deep

- You might have expected the network to **overfit** the data, but instead it **underfits**.
- SGD has **stalled**.

2) Network is too deep

- The plot of gradients by layer shows why:



Vanishing Gradients

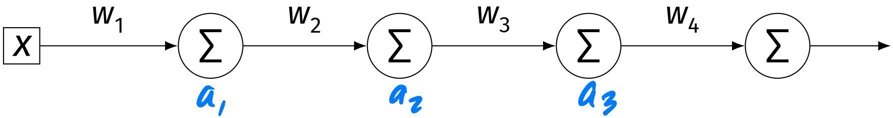
- This is the problem of **vanishing gradients**.
  - As the network gets deeper, the gradients get smaller and smaller until they vanish.
- Remember: we adjust w_i by $w_i \leftarrow w_i - \alpha \frac{\partial R}{\partial w_i}$.
- If $\frac{\partial R}{\partial w_i}$ is very small, then w_i won't change much.
- If w_i doesn't change much, the NN **won't learn**.

Why?

- Remember, we calculate $\frac{\partial R}{\partial w_i}$ using the chain rule.
  - Lots of terms get multiplied together.
  - These terms are often less than 1 (in magnitude).
  - If so, the product vanishes with more terms.

Example

- Consider a simple network with 3 hidden layers, 1 neuron per layer, and linear activations:

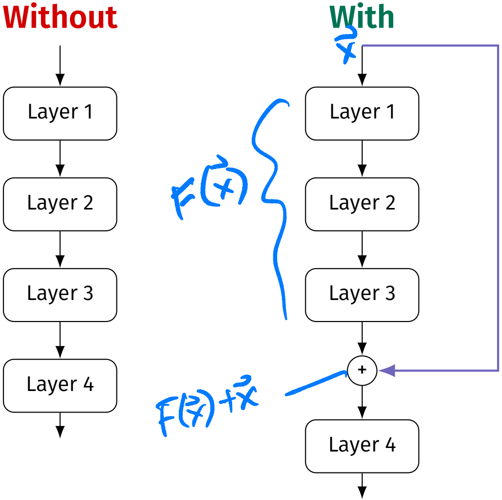


- What is $\partial H / \partial w_4$? a_3
- What is $\partial H / \partial w_3$? $a_2 w_4$
- What is $\partial H / \partial w_2$? $a_1 \; w_3 \; w_4$
- What is $\partial H / \partial w_1$? $x \; w_2 w_3 w_4$

Fixes

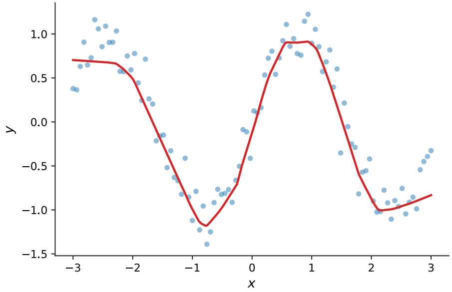
- Use fewer layers.
  - But sometimes we **need** the depth for model capacity.
- Use **skip connections**.
  - A.k.a., **residual connections**.
- Use **batch normalization**.

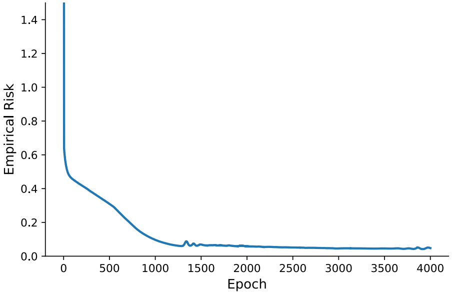
Skip Connections



Fix: Skip Connections

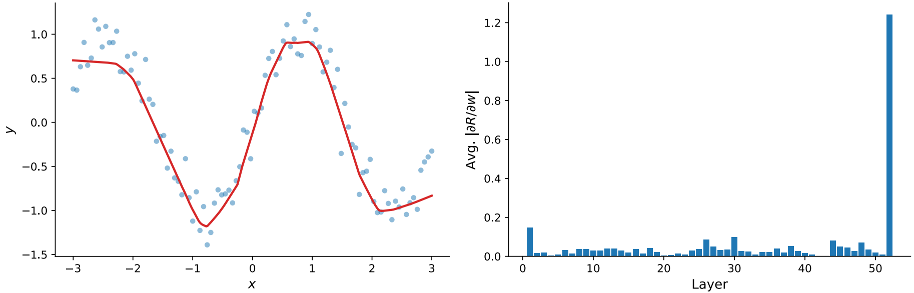
- 25 residual blocks with 2 hidden layers each, 10 neurons per layer.





Fix: Skip Connections

- 25 residual blocks with 2 hidden layers each, 10 neurons per layer.



Batch Normalization

- **Batch normalization** normalizes each neuron's linear output across the batch:

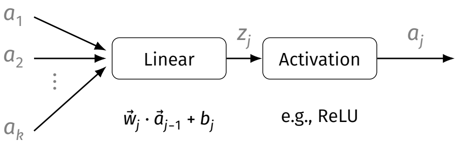
$$\hat{z}_j = \frac{z_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

where μ_j and σ_j^2 are computed within batch.

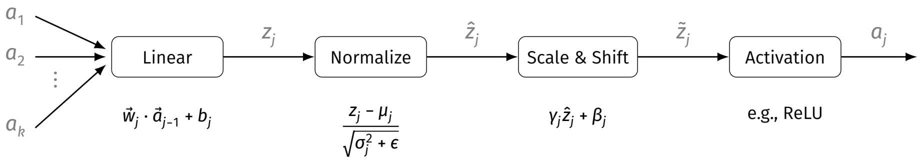
- Then applies a learnable scale and shift:

$$\tilde{z}_j = \gamma_j \hat{z}_j + \beta_j$$

Batch Normalization



Batch Normalization

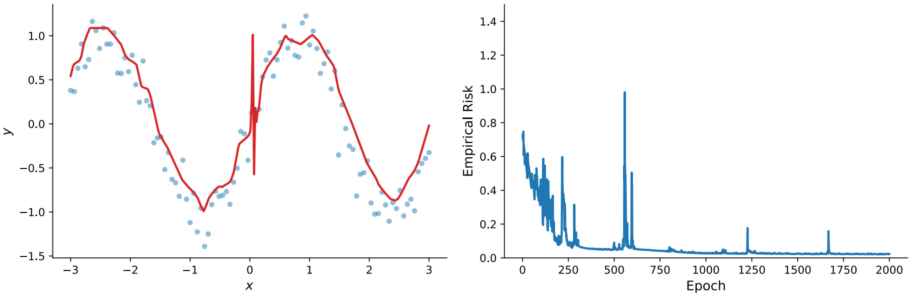


- Typically applied at each hidden layer.

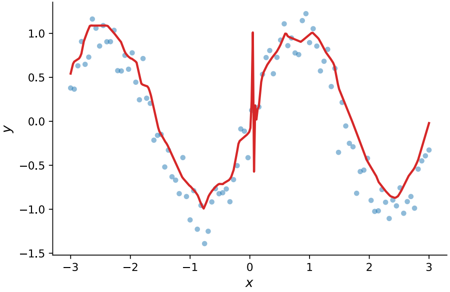
Why does it help?

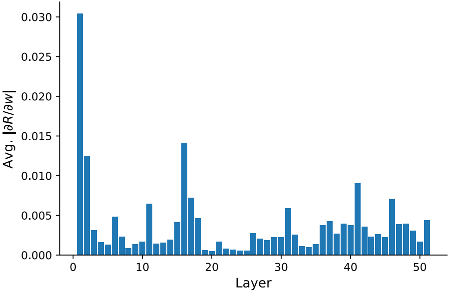
- The activation function receives normalized $\tilde{z}_j$ instead of z_j.
- Keeps activations in a “reasonable” range.
- Useful when activation can “saturate”.
  - We’ll see this in a moment with the sigmoid.

Fix: Batch Normalization

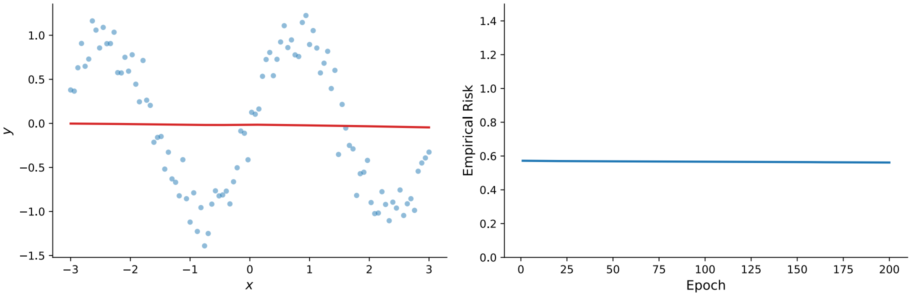


Fix: Batch Normalization

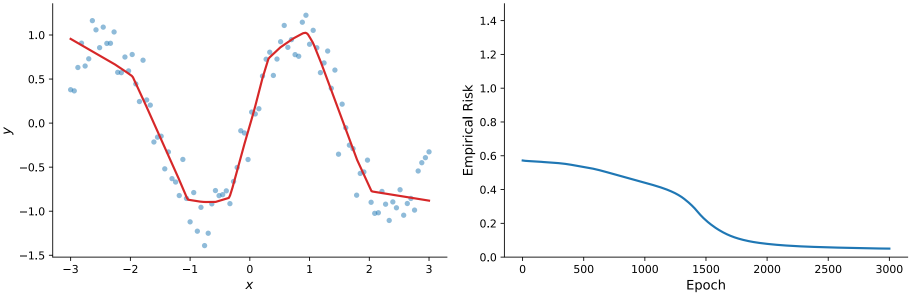




3) You didn’t train long enough



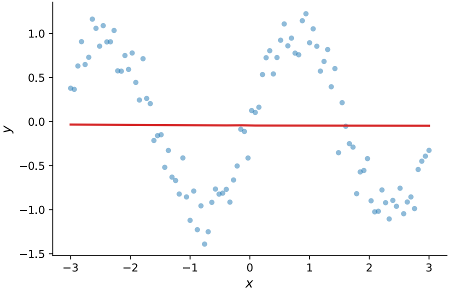
Fix: Train longer

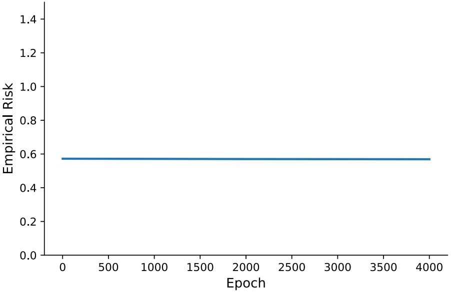


4) Learning rate is too high/low

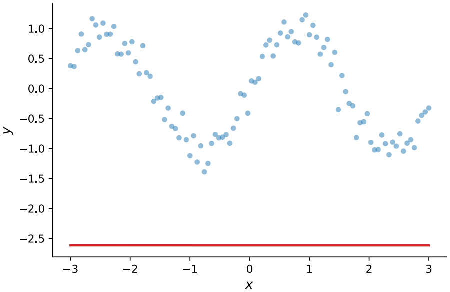
- When you run SGD, you must choose the learning rate α.
- What happens if α is **too low** or **too high**?

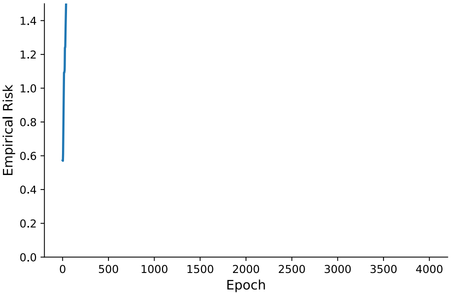
Learning rate too low



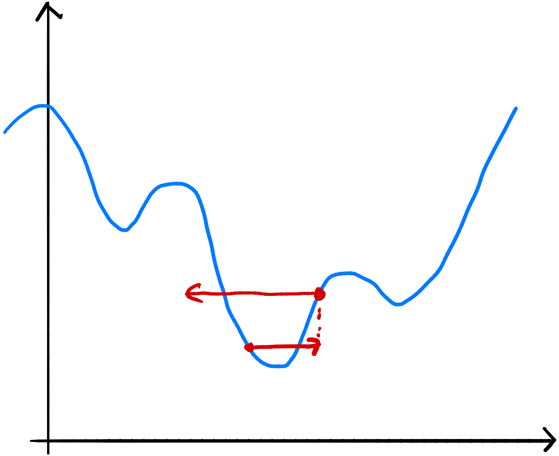


Learning rate too high





What is happening?



Fix: Tune learning rate

- Try different learning rates and see which one works best.
- Try a logarithmic scale: $\alpha \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$.

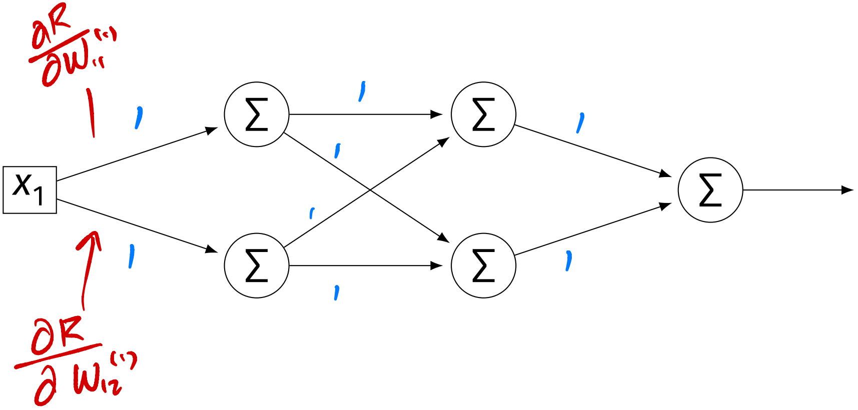
5) Bad initialization

- We have to start SGD from somewhere.
- How should we choose $\vec{w}^{(0)}$?

Exercise

What happens if we initialize all weights to the same number? For example, $w_i^{(0)} = 1$ for all i.

Result



Bad initializations

- If weights are **too small**, gradients will **vanish**.
- If weights are **too large**, gradients will **explode**.

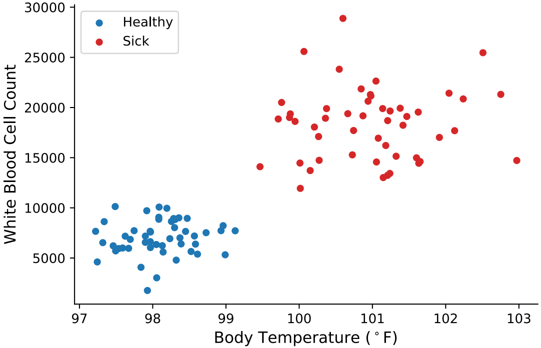
Fix: Random Initialization

- Initialize each weight randomly from a Gaussian distribution with mean zero.

- Common choices for variance.
  - **Xavier initialization**: $1/n_{\text{in}}$. Designed for sigmoid.
  - **He initialization**: $2/n_{\text{in}}$. Designed for ReLU.
    - A.k.a., **Kaiming initialization**.

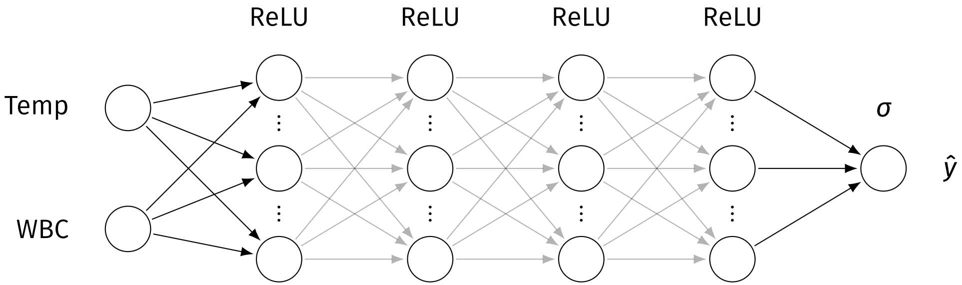
- PyTorch (mostly) uses He initialization by default.

6) Risk is ill-conditioned

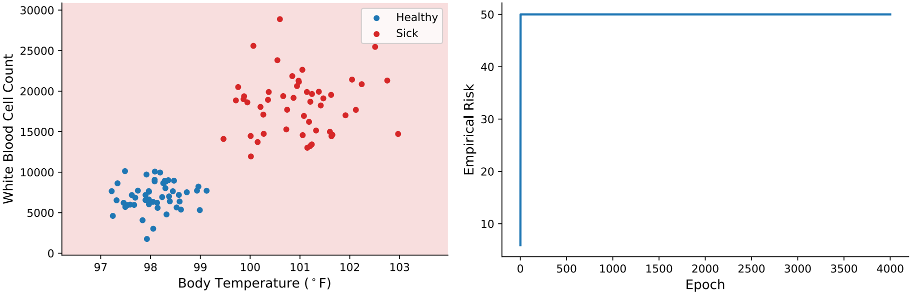


6) Risk is ill-conditioned

- 4 hidden layers, 10 neurons each, ReLU activations.
- Sigmoid output activation.

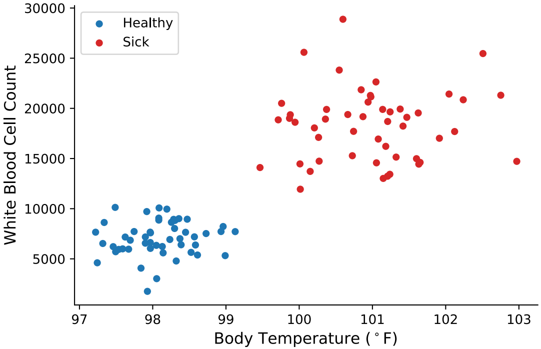


6) Risk is ill-conditioned



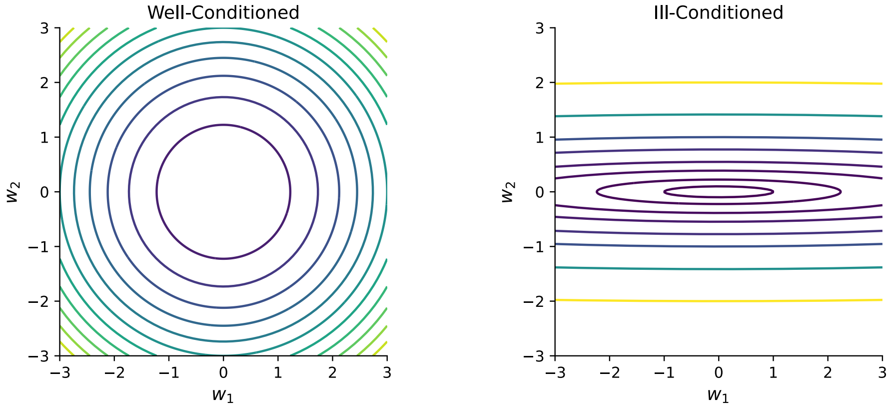
What is happening?

- Notice: the features are on very different scales.



What is happening?

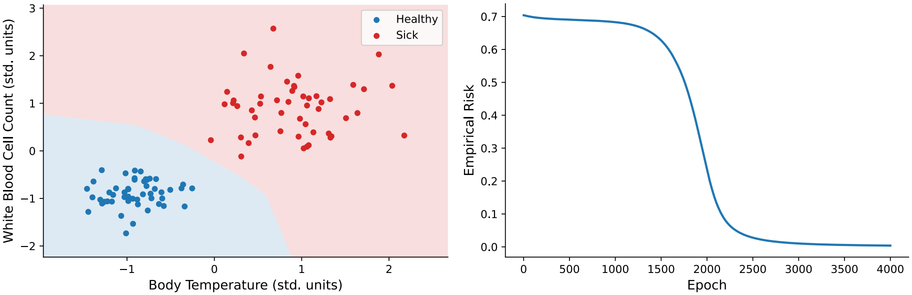
- The risk surface is **ill-conditioned**.
  - Some directions are shallow, others are steep.



Fix: Preconditioning

- Standardize the training data:
  - For each feature, subtract mean, divide by s.d.
- Will need to do the same to test data.
- Could also use batch normalization.

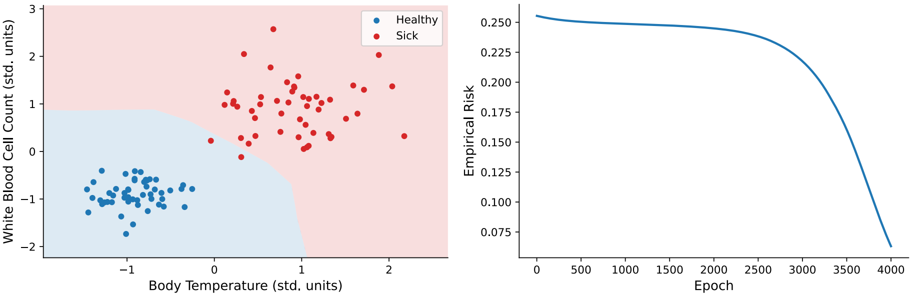
Fix: Preconditioning



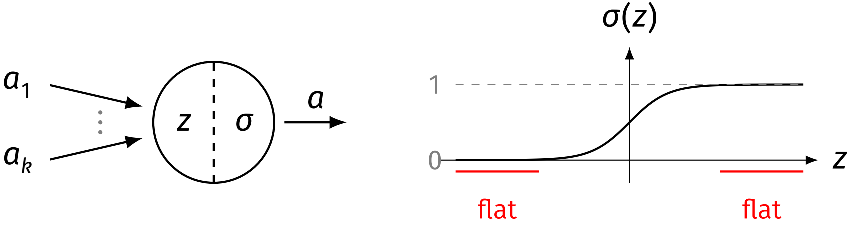
7) Activation function saturates

- Sigmoid output activation is useful for binary classification.
- But the sigmoid can **saturate**.
- Let's train the same network, still with sigmoid output activation, now using MSE.

7) Activation function saturates

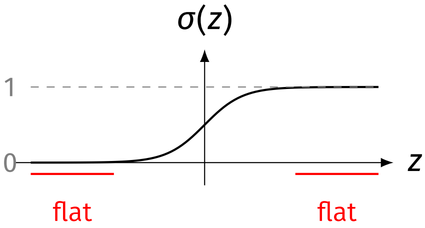


Why?



- When $|z|$ is large, the sigmoid is **saturated**.

Why?



- Remember: in backprop, we compute:

$$\frac{\partial H}{\partial z} = \frac{\partial H}{\partial a} \cdot \sigma'(z)$$

- **Problem**: $\sigma'(z) \approx 0$ when saturated, so gradients **vanish**.

One Fix: Batch Norm

- Recall: batch normalization.
- Normalizes z, helps prevent saturation.
- This is one fix, but there's a better way.

Fix: the Cross-Entropy

- When using sigmoid for output nodes, we often use **cross-entropy** as loss.
- Let $y^{(i)} \in \{0, 1\}$ be true label of ith example.
- The average cross-entropy loss:

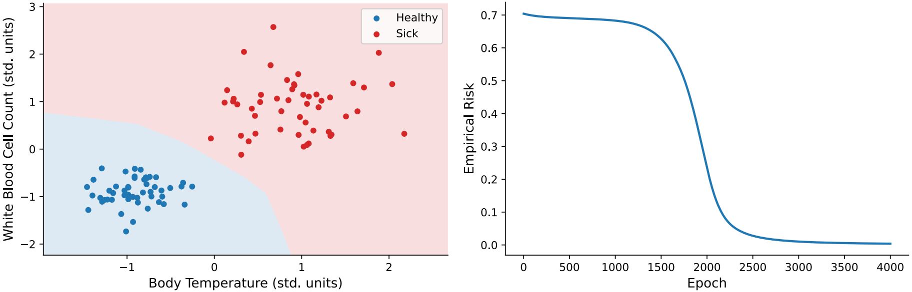
$$-\frac{1}{n} \sum_{i=1}^{n} \begin{cases} \log H(\vec{x}^{(i)}), & \text{if } y^{(i)} = 1 \\ \log\left[1 - H(\vec{x}^{(i)})\right], & \text{if } y^{(i)} = 0 \end{cases}$$

Why cross-entropy?

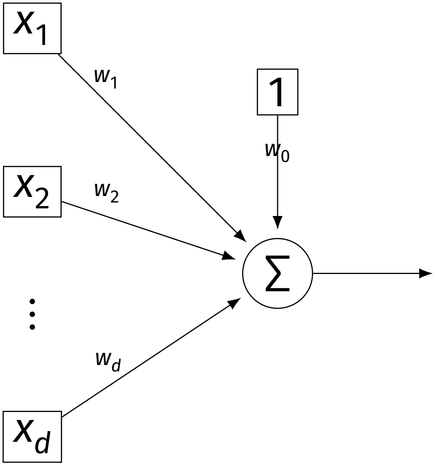
- The log in the cross-entropy cancels the exp in the sigmoid.

- Now, gradient does not vanish even when sigmoid is saturated.

Fix: Cross-Entropy



Special Case: Logisitic Regression

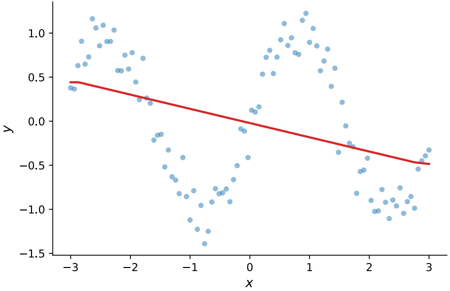


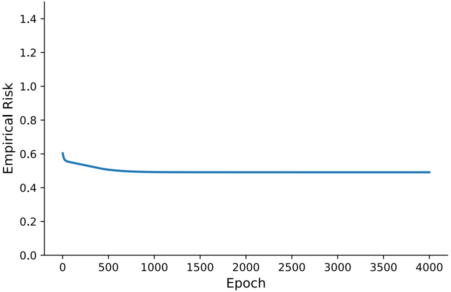
The case of:

- a one layer neural network
- with sigmoid activation
- trained with cross-entropy loss

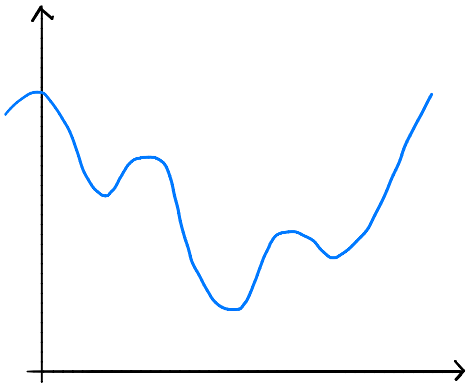
is also called **logistic regression**.

8) Stuck in a local minimum

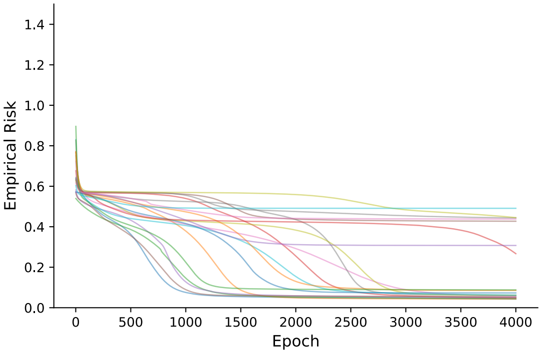




8) Stuck in a local minimum



8) Stuck in a local minimum



Fix: Momentum

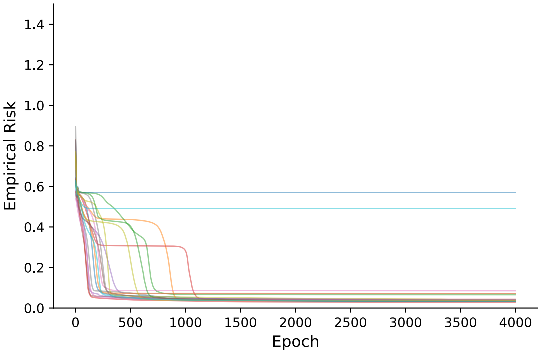
- Instead of updating weights with just the gradient, accumulate a **velocity**:

$$\vec{v}^{(t+1)} = \beta \vec{v}^{(t)} + \nabla R(\vec{w}^{(t)})$$
$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \alpha \vec{v}^{(t+1)}$$

- The velocity builds up in consistent directions, helping SGD "roll through" shallow local minima.

- Typical value: $\beta = 0.9$.

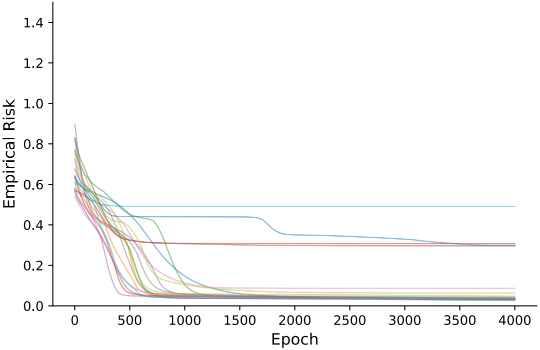
Fix: Momentum



Another Fix: Adam

- There are more sophisticated optimizers that also help with local minima, such as **Adam**.

Another Fix: Adam

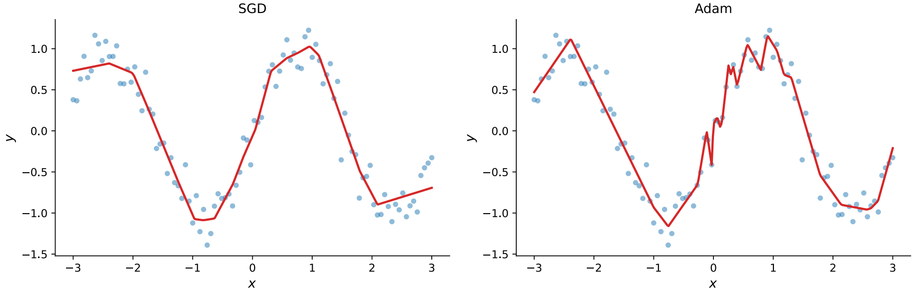


Adam vs. SGD

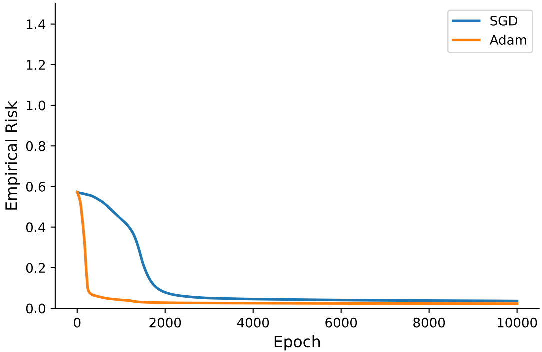
- Why not always use Adam instead of SGD?

Adam vs. SGD

- Adam tends to fit the training data more aggressively than SGD.
- This can lead to **overfitting**.



Adam vs. SGD



SGD
Adam
Empirical Risk
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000
Epoch

Summary

- **Advice:** start simple, with linear models.
- Graduate to deep learning only if you need to.