Discussion 7

CS188 - Fall 19

Objectives

- A review of machine learning topics
  - Loss, risk and training
  - Decision functions and model parameters
  - Cross Validation
  - Local Minima
  - Gradient descent
- An introduction to Deep Learning
  - The setting
  - Forwarding
  - Backpropagation
  - Gradient descent algorithms
    - SGD, NAG, Adagrad, Adadelta, Adam
- Homework 2
  - Any questions about last time?
  - The derivation

The setting

- What function maps 2 to 4 and 4 to 8?

The setting

- What function maps 2 to 4 and 4 to 8?
  - f(x) = 2x
  - And many many others!
- What function maps 0 to 1 and Pi to -1?
  -

The setting

- What function maps 2 to 4 and 4 to 8?
  - f(x) = 2x
  - And many many others!
- What function maps 0 to 1 and Pi to -1?
  - cos(x)
  - And many many others!
- What function maps the evolution of worldwide temperatures to water levels?

The setting

- What function maps 2 to 4 and 4 to 8?
  - f(x) = 2x
  - And many many others!
- What function maps 0 to 1 and Pi to -1?
  - f(x) = cos(x)
  - And many many others!
- What function maps the evolution of worldwide temperatures to water levels?
  - …
  - Maybe we can learn it with enough data? We won't be able to write down a mathematical expression for it, but we could use it to predict new points

A machine learning recipe

- You always start by defining a problem you are trying to solve
  - I want to recognize dogs in images
- You gather a dataset
  - 10,000 pictures from Google Images, with dogs in 25% of them
  - The corresponding labels for all images
  - A training sample is an image and its label
- You define a *decision* and *loss* function
  - A decision function maps inputs to labels **for a given set of parameters**
  - A loss function maps predictions and labels to an **error**
- You train the model
  - What set of parameters leads to the lowest error?

A real problem

- You will train a model on some data
  - No one cares about performance on that data
  - This is a **fake** problem
    - Your data is already labelled
    - Predicting labels you already have is *pointless*

A real problem

- You will train a model on some data
  - No one cares about performance on that data
  - This is a **fake** problem
    - Your data is already labelled
    - Predicting labels you already have is *pointless*
- What you hope to do is to extract information *from the data you have* to gain intuition about data *you don't know*
  - You always want to create a model that performs well 'in the real world'
  - So that you never have to label anything ever again
- Your model's quality is only as good as your data
  - You won't be able to 'create information'
  - This is the **data processing inequality**

What is a loss function?

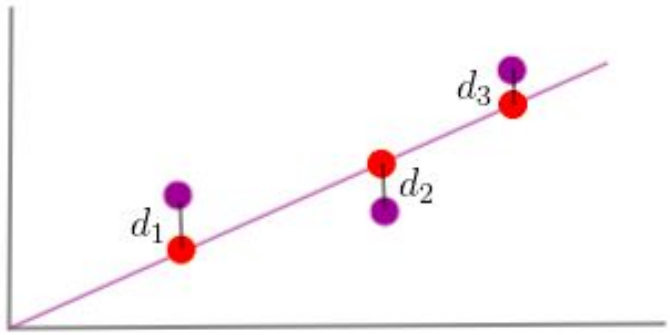
- Machines learn by means of a loss function
  - For a given task, if *predictions* (from your model) deviate from data, the loss function would output a large value
  - If predictions match data, the loss function would be zero
    - A loss function is nonnegative
- Loss functions are task dependent
  - There is no function that fits every problem
  - Designing loss functions is a hard problem, and has to be done for every task
- What is the “loss function” a function of?
  -

What is a loss function?

- Machines learn by means of a loss function
  - For a given task, if *predictions* (from your model) deviate from data, the loss function would output a large value
  - If predictions match data, the loss function would be zero
    - A loss function is nonnegative
- Loss functions are task dependent
  - There is no function that fits every problem
  - Designing loss functions is a hard problem, and has to be done for every task
- What is the “loss function” a function of?
  - Your data (which you have no control over)
  - The parameters of your model (you try to find the best values)
- Given a loss function, the goal is to find its minimum
  - This gives you the best set of parameters
  - This is called **training**

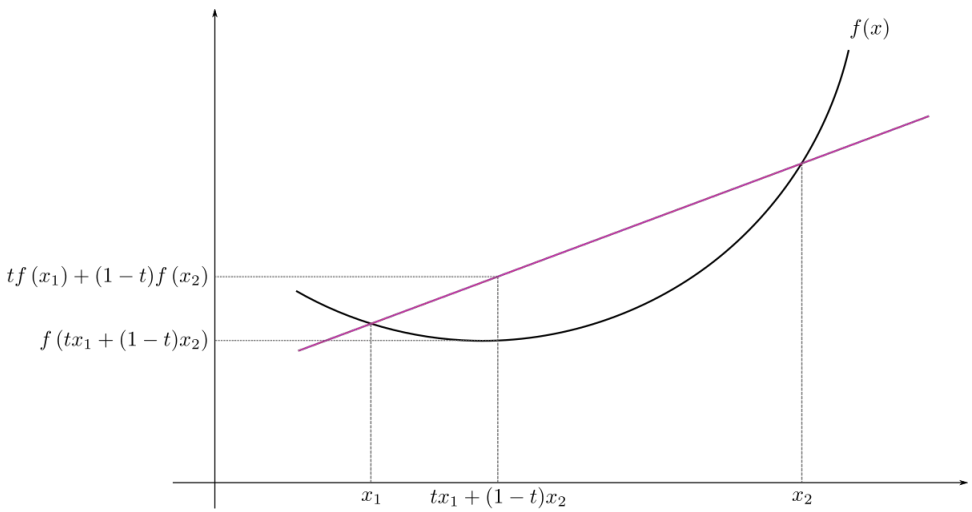
Risk

- Risk is the expected value of your loss over all predictions.
- The optimal parameters minimize risk **over the training data**



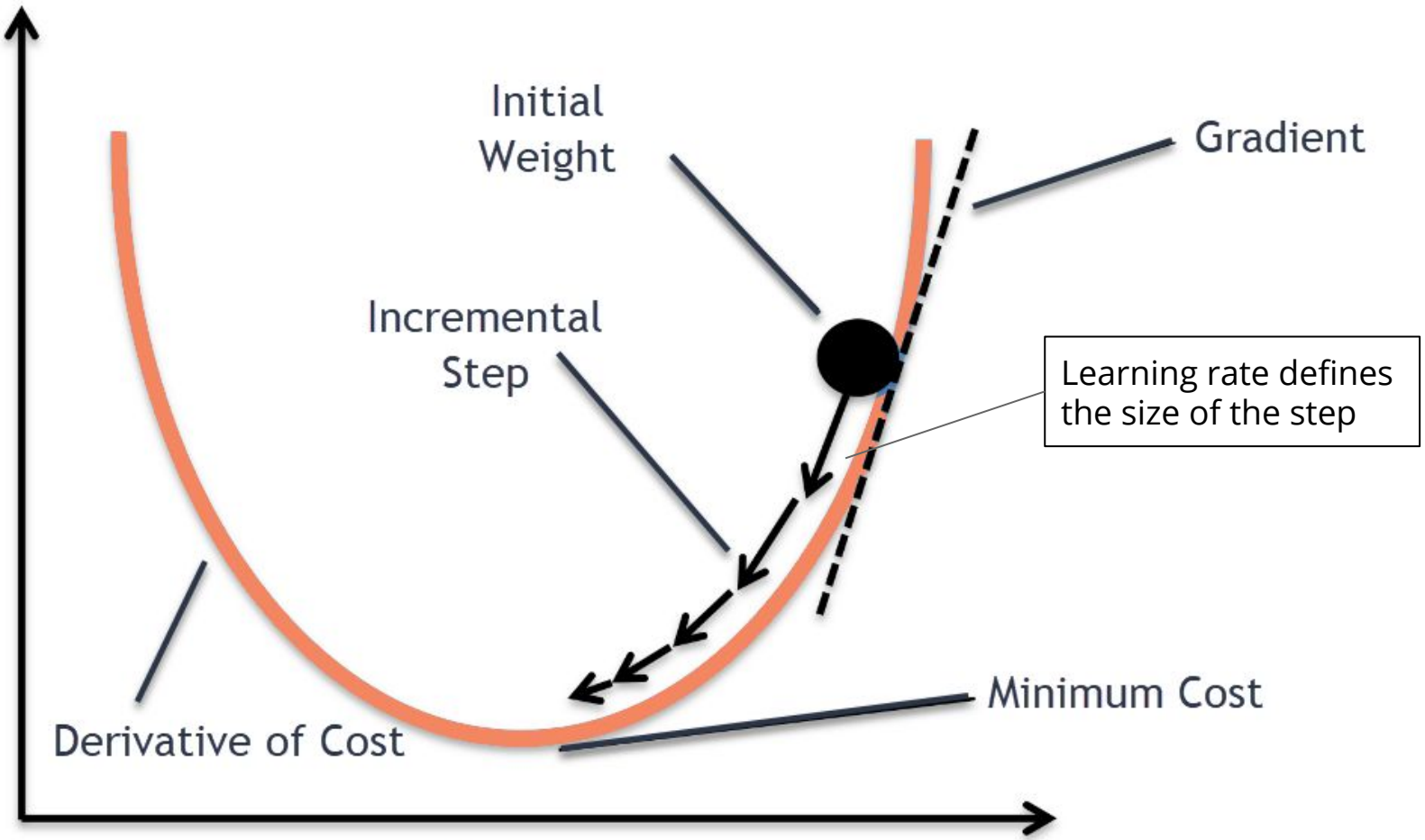
Convex functions

<u>Convex:</u> a straight line between any 2 points is above the curve



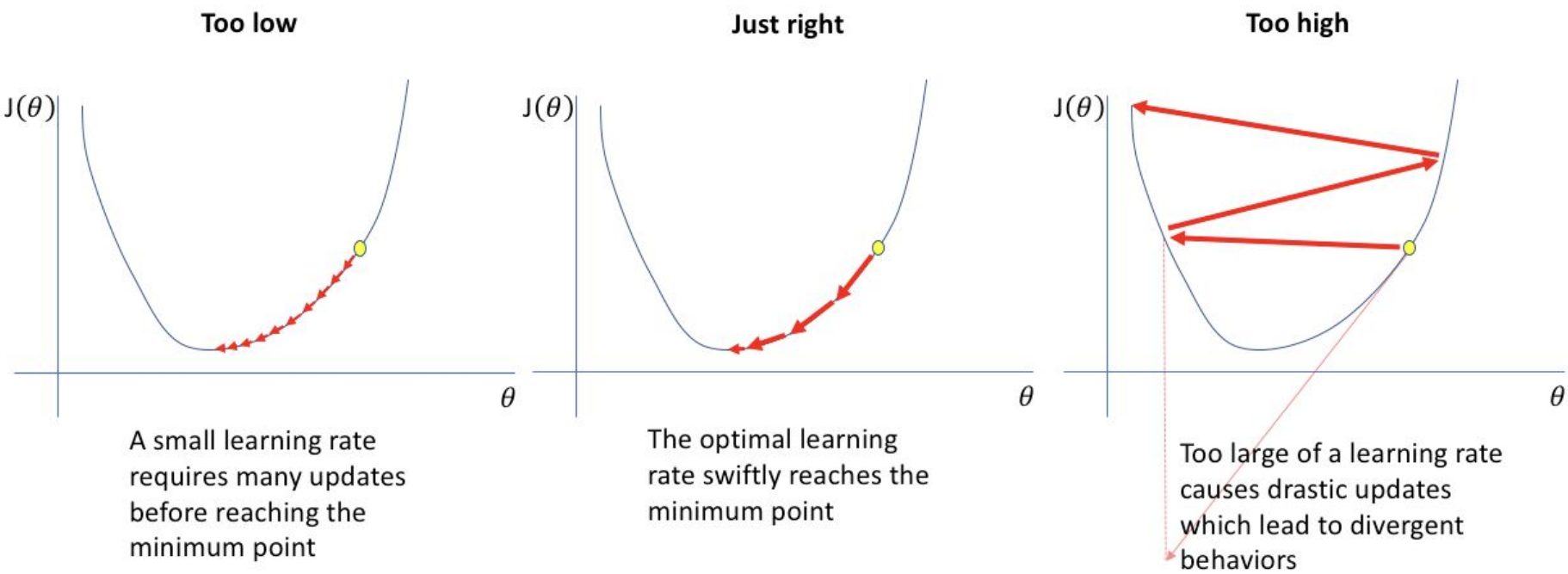
<u>Why this matters:</u> Convex functions have **a single** global minimum, ie: a single set of optimal parameters (for a loss function)

Gradient Descent



Learning rate

The learning rate is the size of the step you take at each iteration



How do I find the best learning rate?

Some bad news

- What happens when we take a step in gradient descent?

Some bad news

- What happens when we take a step in gradient descent?
  - We are changing model parameters
  - The gradient is computed with respect to the parameters
- What is the dimensionality of the loss function?

Some bad news

- What happens when we take a step in gradient descent?
  - We are changing model parameters
  - The gradient is computed with respect to the parameters
- What is the dimensionality of the loss function?
  - (Dimensionality of the input) x (Number of parameters)
  - A deep neural network has parameters in the millions ...
  - This is really optimization in millions of dimensions!
- What guarantees on performance do we have once we found the minimum?

Some bad news

- What happens when we take a step in gradient descent?
  - We are changing model parameters
  - The gradient is computed with respect to the parameters
- What is the dimensionality of the loss function?
  - (Dimensionality of the input) x (Number of parameters)
  - A deep neural network has parameters in the millions …
  - This is really optimization in millions of dimensions!
- What guarantees on performance do we have once we found the minimum of the loss?
  - None!
  - We can't guarantee what will happen on data we haven't seen yet …
  - We (hopefully) extracted as much information as we could from the data we had

Cross-Validation

- We can't know what data our model will be ran on once we are done training
  - But we want to perform well on it
- We want to simulate performance on unknown data
  - To simulate how well our model will work on unseen data
  - Therefore, we split the data we have into a training set and a testing set
    - The testing set is never used to update model parameters
    - It is simply used to see how well our model extrapolates to new examples
  - We are interested in performance on the **testing set**, not the training set
- This is called cross-validation
  - We simulate how well our model extracted the information from the data by testing on unseen data

Cross-Validation methods

Refer to Pr.Scalzo's slides, but the main methods are:

- Holdout method
  - Like Hw1, keep k% (k typically is equal to 10-25) for testing
  - Problem: no guarantee that test and train distributions are equal
  - Problem: a lot of data is 'sacrificed', never used to train

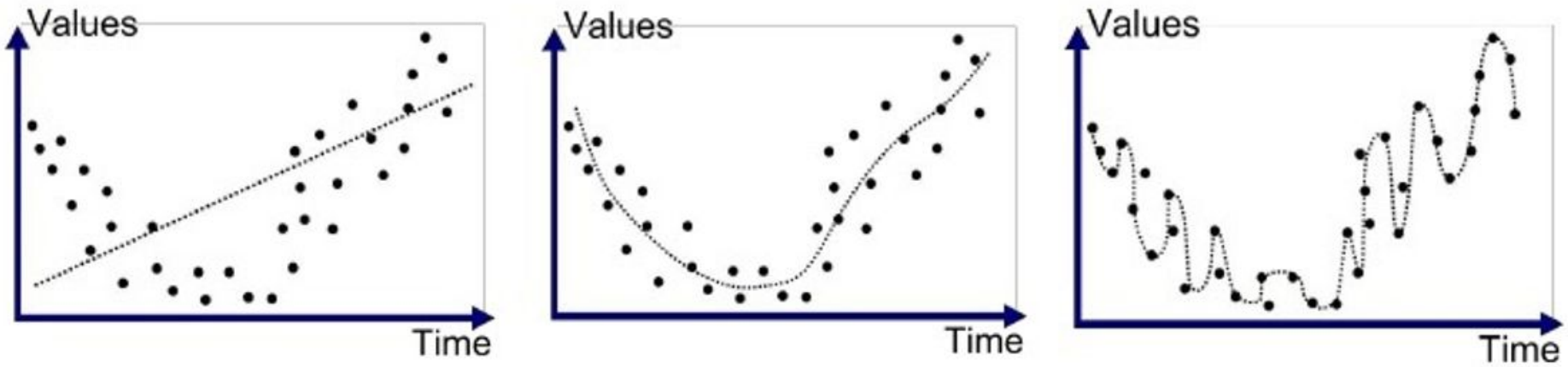
- K-fold cross validation (k typically 5-10)
  - Divide data over k subsets
  - Use k-1 subsets to train and predict the remaining subset
  - Rotate subsets, and average results
    - Every data point is in the validation set exactly once
    - Every data point is in the training set (k-1) times

Fitting

- <u>Underfitting:</u> Your decision function doesn't have enough parameters to represent the data.
  - No amount of training can fix this
  - Training error does not converge to 0
- <u>Overfitting:</u> Your decision function has too many parameters and starts to memorize the dataset instead of learning from it
  - We're no longer training a model that would perform well on unknown data
  - We're training a model that performs well on the training data
    - It has been tailored to the training data
  - <u>Symptom:</u> Error on the training set decreases, but increases on the testing set
  - Our model no longer generalizes

Fitting

- <u>Underfitting:</u> Your decision function doesn't have enough parameters to represent the data.
  - No amount of training can fix this
  - Training error does not converge to 0
- <u>Overfitting:</u> Your decision function has too many



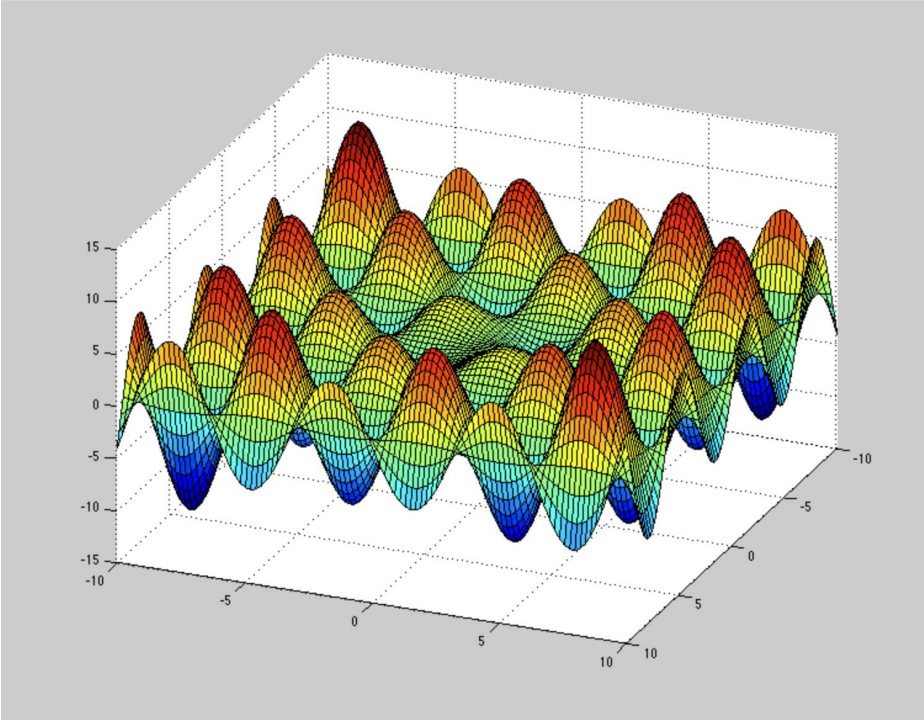
Underfitted Good Fit/Robust Overfitted

Regularization

- Typically, underfitting is not a problem
  - We just use a more complicated model / a model with more parameters
  - That exposes us to overfitting
- Regularization is used to prevent overfitting
  - This is done by limiting the allowable values of parameters
  - You restrict the space of solutions, to force the model to remain simpler
    - Too much regularization: solution space too small, hard to fit
    - Too little regularization: overfitting not mitigated
    - A good balance: is hard to find, it's almost an art …
- A regularization function is:
  - A function of the model parameters
  - Is nonnegative
  - What we optimize is now the sum of the loss and the regularizer

More bad news

- Loss functions are non-convex
  - There are a lot of local minima!



The setting (cont)

- You can see deep learning models as **universal function approximators**
  - With enough data, you can **train** a neural network to approximate **any function**
  - Functions that you wouldn't be able to derive analytically
- What do we mean by train?

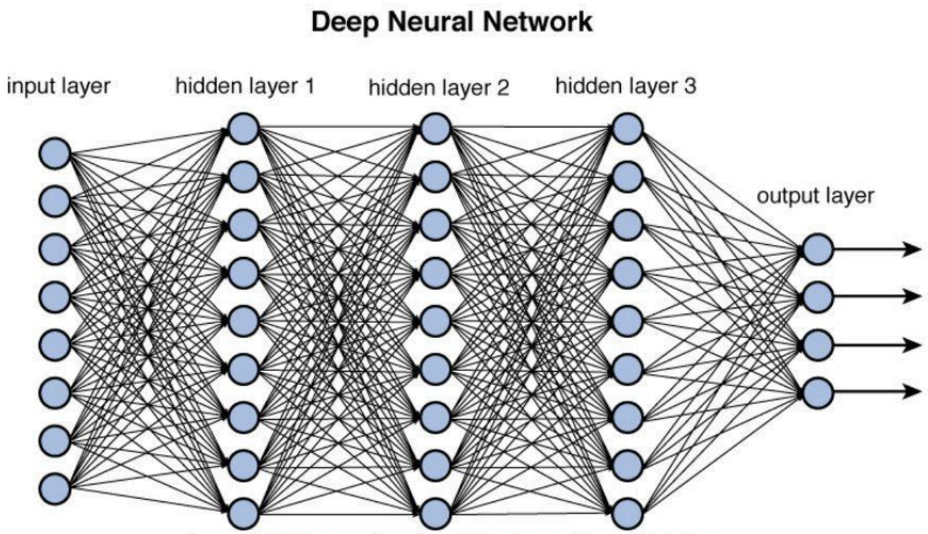


Figure 12.2 Deep network architecture with multiple layers.

The loss function

Gradient descent

- What is the goal of gradient descent?

Gradient descent

- What is the goal of gradient descent?
  - It is an optimization algorithm, which we use here to find minima of the loss function
- What gradients are we computing?

Gradient descent

- What is the goal of gradient descent?
  - It is an optimization algorithm, which we use here to find minima of the loss function
- What gradients are we computing?
  - The gradients of the loss function, with respect to ...

Gradient descent

- What is the goal of gradient descent?
  - It is an optimization algorithm, which we use here to find minima of the loss function
- What gradients are we computing?
  - The gradients of the loss function, with respect to **the weights of the network**
  - We are trying to find the **optimal set of weights**
  - The weights define what function we are approximating
- We will consider the following setting:
  - The loss function is $J(\theta)$
  - The parameters we are trying to optimise are $\theta \in \mathbb{R}^d$
  - The gradients we are computing are $\nabla_\theta J(\theta)$

Gradient

W.r.t the parameters

The variants

- The idea is always the same:
  - Same loss function
  - Same parameters to optimize
  - Same gradients computed
  - Same goal: find a minimum
- However, different algorithms exist to perform the same task
  - The updates are different
  - Some have different hyperparameters
  - Some work on different sets of inputs
- It is important to know that the variants exist, and what the differences are between them

(Batch) Gradient descent

<u>Batch gradient descent:</u> Computes gradient over **entire dataset**

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Learning rate

- Entire dataset has to fit in memory
  - Generally not a valid assumption for vision
- Redundant computation
  - Gradients of similar inputs are recomputed
- Hard to incorporate new data
  - Adding data means recomputing the full gradient from scratch!

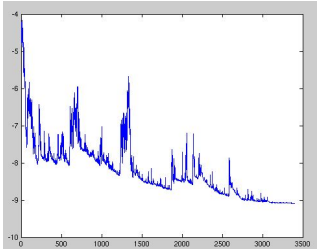
Not used in practice

Stochastic Gradient Descent

- Compute an update **for each training example**

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

- Faster, no need for huge memory
- Able to incorporate new data easily
  - The new gradient to be computed doesn't affect the others
- However, no guarantee that gradients of different examples are the same
  - And in fact, a guarantee that the gradients will be very different
- This results in a high variance =>



<u>Not used in practice</u>

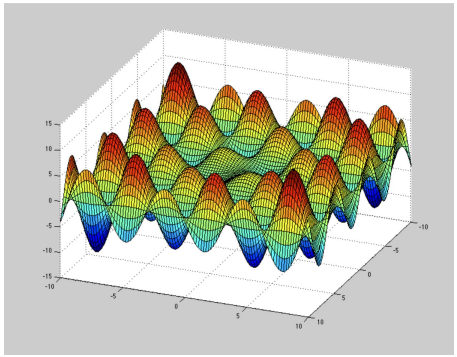
Mini-Batch gradient descent

- The one used in practice
  - So ubiquitous that it is typically called SGD or just gradient descent
- The best of both worlds
  - Reduce variance by computing gradient on multiple examples
  - Reduce computational cost by computing gradient on a subset of the dataset, which also allow to add examples on the fly
- A mini-batch is a fixed subset size of the data
  - GD => N datapoints
  - SGD => 1 datapoint
  - MBSGD => 1 < n < N datapoints

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

Ideas from physics

- You can picture the loss landscape as a series of hills and valleys =>
- SGD is like dropping a ball at a random point, and waiting until it stops moving.



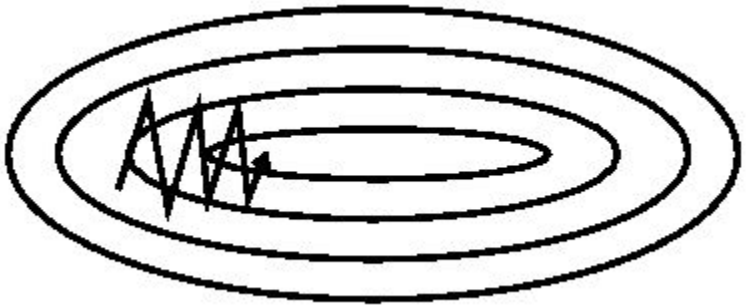
Other algorithms took this a little further to come up with different weight updates.

This is just a way to take different steps in the landscape to reach a minimum more quickly.

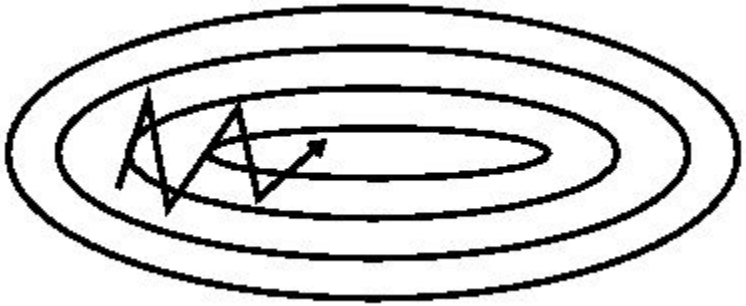
Ideas like **momentum** and **friction** were added to the update rules

Momentum

- If the ball rolls down a hill:
  - It builds speed as it goes
  - It becomes harder to change its direction
- What would be the benefit of applying this to SGD?



<u>No momentum:</u>
SGD is 'greedy' and doesn't take into account the overall horizontal trend



<u>Momentum:</u>
Consistently going right, the ball gains speed in that direction

Momentum (cont)

How would you interpret this?

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Momentum (cont)

How would you interpret this?

Previous update

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Normal update

You now have a short ‘memory’ of the last update, this allows you to build traction.

Can you foresee any problems with this?

Momentum (cont)

How would you interpret this?

Previous update

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Normal update

You now have a short 'memory' of the last update, this allows you to build traction.

Can you foresee any problems with this?

What if you just gained a lot of acceleration into a wall...

Nesterov Accelerated Gradient

- We would like to have a notion of where we are going **before** we reach an increasing slope
  - How would we do that?

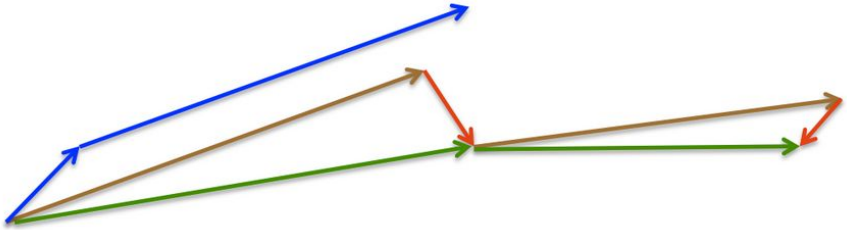
Nesterov Accelerated Gradient

- We would like to have a notion of where we are going **before** we reach an increasing slope
  - How would we do that?
- We can update our parameters as such:

Momentum update: memory

Prediction: gradient at our position **after the update**

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$



Is the prediction exact?

What we achieved

- We can now adjust the size of our steps in a smart way, taking into account more details of the loss landscape
- However, we are still updating the full set of parameters at once!
  - This is not necessarily the best thing, all parameters are not equal!
    - Some of them will be used more frequently than others
    - If a parameter represents ‘an eye’, it will be used frequently for face detection, not so much for car detection …
  - How should we adapt this?
    - Parameters used frequently would cause a lot of instability if updated brutally => they need smaller learning rates
    - Parameters used infrequently need higher learning rates, otherwise it would take a long time to learn them!

Adagrad

- Does just this: weighted updates of parameters
  - Each parameter is now updated individually
- How?
  - Adagrad keeps track of the past gradients for a given parameter, and sums their norms, used as a normalization

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

- If a parameter was updated frequently, its associated norm will be high, and the next update will be small
  - Infrequent parameters will not be affected by this
- Can you spot a problem?

Adadelta

- Under Adagrad, the learning rate is monotonically decreasing!
  - When it gets to small, we lose our ability to learn
- Adadelta introduces **forgetting**
  - Instead of summing over all past history, sum over a window of fixed size
  - Instead of maintaining all gradients in that window, keep only a running average

Decay < 1

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Giving the update rule:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Are we done?

- No, there are a lot of others
  - But you get the idea: the goal is to make smarter updates
- Adam is the most popular algorithm right now
  - Along with vanilla SGD
- Builds off of Adadelta, adds *friction*
  - Adadelta keeps a decaying average of gradient *norms*
  - Adadelta does not keep track of past gradient *directions*
- Why take past direction into account?
  - If you add friction, you tend to slow down in flat areas
  - Adam therefore privileges flat areas, which are exactly the local minima
  - To do so it keeps a running average of mean and variance of the gradients
    - You can find the equations online if you are interested

Conclusion

- Given gradients of a function, we can now efficiently navigate the loss landscape
- Navigating the loss landscape implies changing the parameters of our model, which are the weights of the network
- This **does not** tell you how to choose an architecture or an activation function
  - Indeed, computing gradients implies having a network already!
- Since the loss is highly nonconvex, there are no guarantees of optimality to our solution
  - And we can say even fewer things of performance of real data!
- Our loss function is very high dimensional
  - How do we get the gradients anyway?

Forward

- Forwarding is the action of feeding inputs to the network, with *fixed parameters*
  - You are given x, the input and f(x, p) with p fixed
  - You compute y = f(x, p)
  - This is just evaluating the function at given *points*
    - Plural? We are feeding a mini-batch through the function
    - A small subset of the data
- Why a small subset of the data?

Forward

- Forwarding is the action of feeding inputs to the network, with *fixed parameters*
  - You are given x, the input and f(x, p) with p fixed
  - You compute y = f(x, p)
  - This is just evaluating the function at given *points*
    - Plural? We are feeding a mini-batch through the function
    - A small subset of the data
- Why a small subset of the data?
  - At some point we will need to compute gradients
  - Ideally we want a subset large enough for the gradient of this subset to be representative of the full gradient
- Why not the full data?
  - It would take a very long time to compute
  - When we update our parameters p, we change f !
  - All of our previous calculation must be thrown away!

Backwards

- We now have y = f(x, p)
- We go backwards and compute the gradient of f **with respect to p**
- We try to minimize the gradients, which changes p
- We now have a new function f(x, p2)
  - Training a network is just a sequence of forwards and backwards
- The act of going backwards is called **backpropagation**
- An **epoch** is the number of time steps it takes to have forwarded the full dataset through the network
  - After an epoch, the entire data has been seen
- Models can train for hundreds of epochs

Backpropagation: example

On the board ...

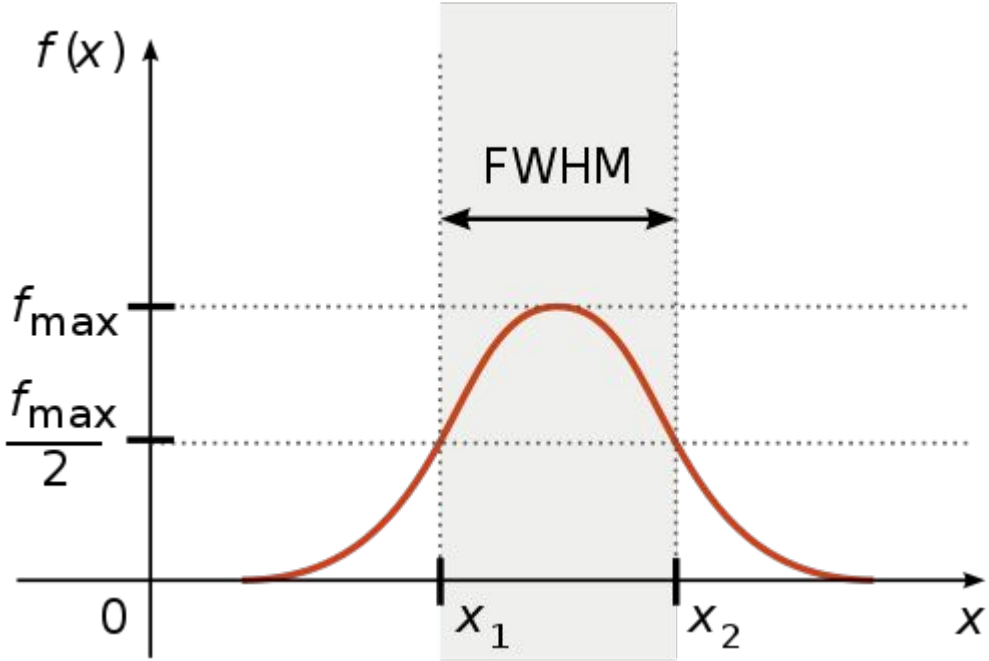
Any questions?

About what we covered last lecture:

- Depth of Field
- Relationship between depth of field and aperture
- Pinhole camera model
- Lens camera model
- Focal length
- … ?

Full Width at Half Maximum (FWHM)

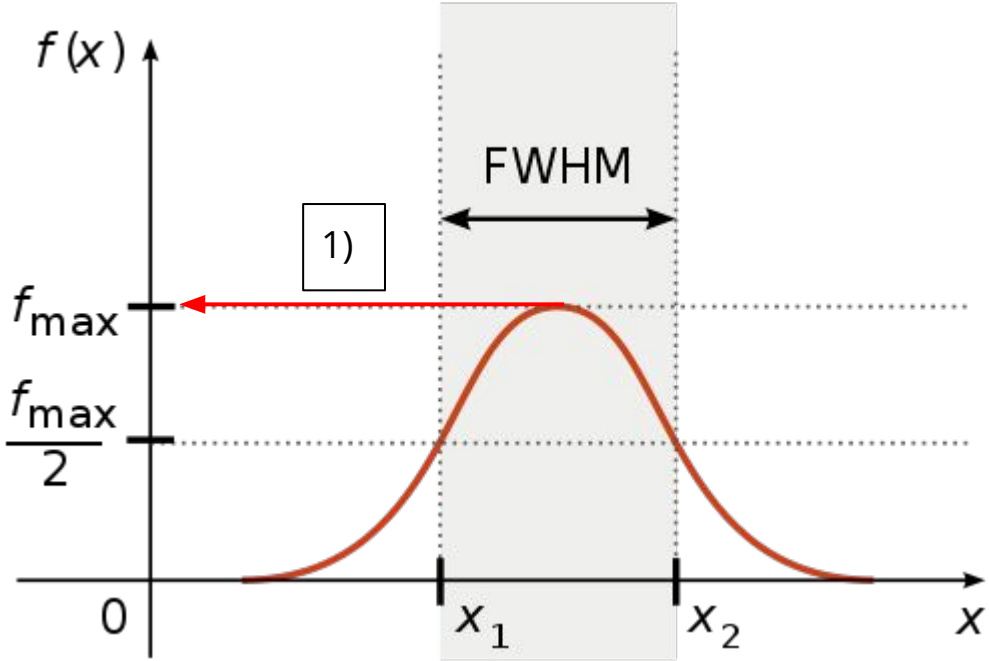
For a Gaussian:



Full Width at Half Maximum (FWHM)

For a Gaussian:

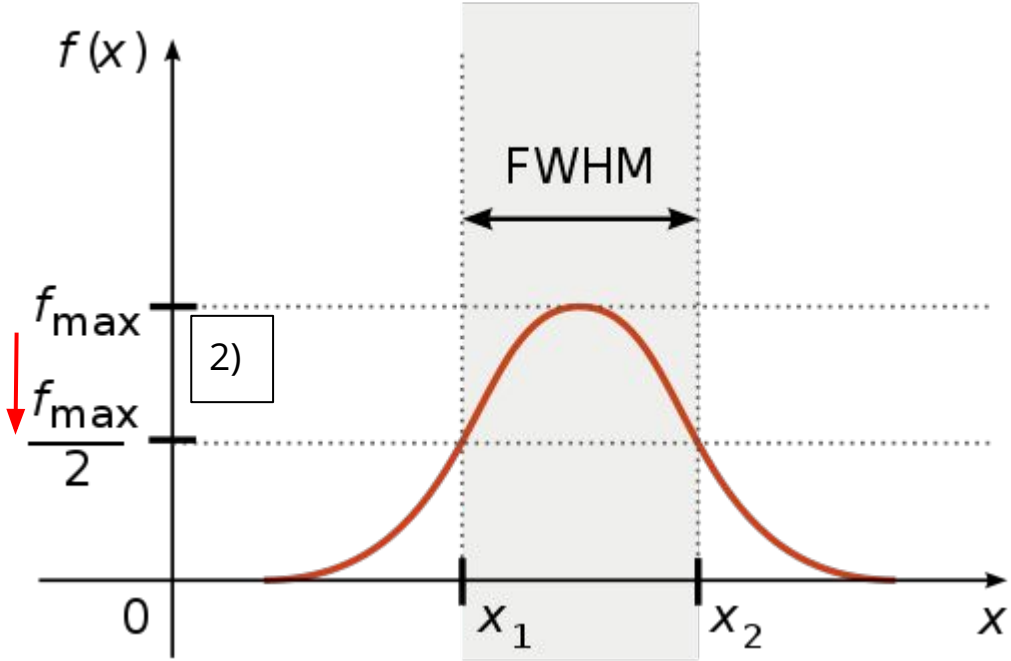
1) Find the max



Full Width at Half Maximum (FWHM)

For a Gaussian:

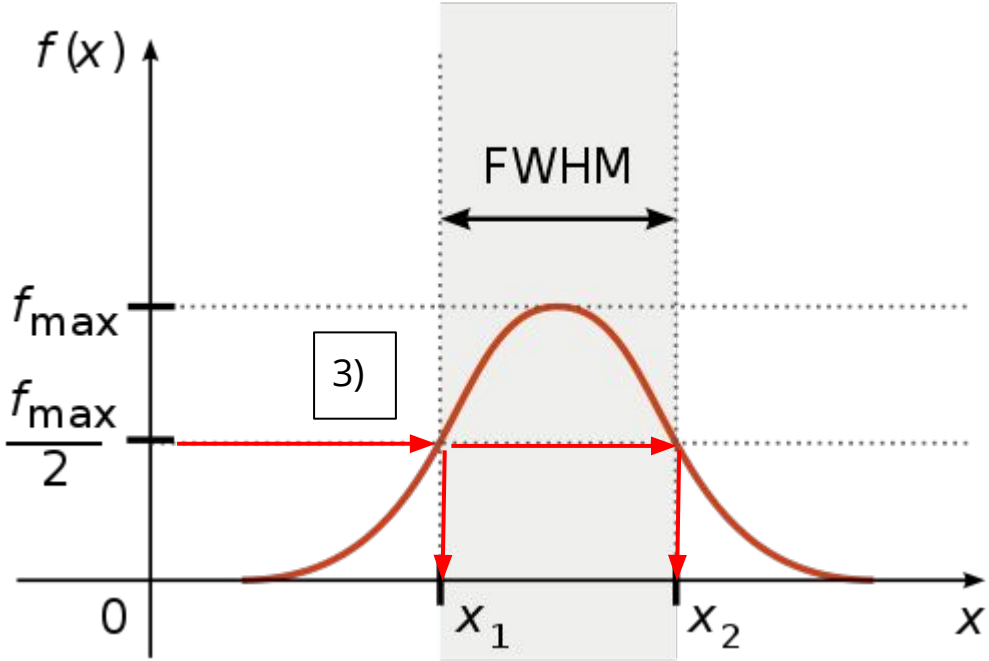
1) Find the max
2) compute max/2



Full Width at Half Maximum (FWHM)

For a Gaussian:

1) Find the max
2) compute max/2
3) Find corresponding x



This metric tells us how 'large' our kernel is, and therefore how much blurring we are adding

Derivation and kernel shape

On the board ...