Deep Neural Networks - Lecture 4

Marcin Mucha

April 4, 2018

Gradient Descent

Overfitting

Saturation

Batch Normalization

Plan

Gradient Descent

Overfitting

Saturation

Batch Normalization

Gradient Descent - Recap

(Batch) Gradient Descent works by making steps in direction of the gradient of the loss function, e.g.

$$L(W, B) = \frac{1}{n} \sum_i L_i(W, B),$$

where $L_i(W, B)$ is the loss on the i-th example, and depends on W and B.

Gradient Descent - Recap

(Batch) Gradient Descent works by making steps in direction of the gradient of the loss function, e.g.

$$L(W, B) = \frac{1}{n} \sum_i L_i(W, B),$$

where $L_i(W, B)$ is the loss on the i-th example, and depends on W and B.

Computing the gradient requires going over the whole dataset and is very expensive, especially nowadays when often dealing with massive datasets.

Stochastic Gradient Descent

Stochastic Gradient Descent is instead computing the gradient $\nabla L_i(W, B)$ of the single example loss $L_i(W, B)$ and using it as an estimate of $\nabla L(W, B)$.

Stochastic Gradient Descent

Stochastic Gradient Descent is instead computing the gradient $\nabla L_i(W, B)$ of the single example loss $L_i(W, B)$ and using it as an estimate of $\nabla L(W, B)$.

This is very fast, but also the gradients are very noisy. In practice, with small enough η this has all the good properties of GD and works faster.

Stochastic Gradient Descent

Stochastic Gradient Descent is instead computing the gradient $\nabla L_i(W, B)$ of the single example loss $L_i(W, B)$ and using it as an estimate of $\nabla L(W, B)$.

This is very fast, but also the gradients are very noisy. In practice, with small enough η this has all the good properties of GD and works faster.

Also allows for online-learning, in particular no need to keep train set in memory.

Mini-batch SGD

Mini-batch Gradient Descent is a mix of both approaches. Split the data into mini-batches of fixed size s. For each mini-batch S compute the gradient of $\frac{1}{s}\sum_{i\in S} L_i(W, B)$ and use this as an estimate of the gradient $\nabla L(W, B)$.

Mini-batch SGD

Mini-batch Gradient Descent is a mix of both approaches. Split the data into mini-batches of fixed size s. For each mini-batch S compute the gradient of $\frac{1}{s}\sum_{i\in S} L_i(W,B)$ and use this as an estimate of the gradient $\nabla L(W,B)$.

This estimate is less noisy, but takes longer to compute.

Mini-batch SGD

Mini-batch Gradient Descent is a mix of both approaches. Split the data into mini-batches of fixed size s. For each mini-batch S compute the gradient of $\frac{1}{s}\sum_{i\in S} L_i(W, B)$ and use this as an estimate of the gradient $\nabla L(W, B)$.

This estimate is less noisy, but takes longer to compute.

Important: Computing the gradient for a batch of size 100 is not 100 times slower than computing the single example gradient, it is much faster than that! But the larger the batch size the more linear this behaviour is.

Technical issues

Some important details:

- Before each run through the whole dataset (*epoch*), you want to shuffle the dataset (could be expensive).

Technical issues

Some important details:

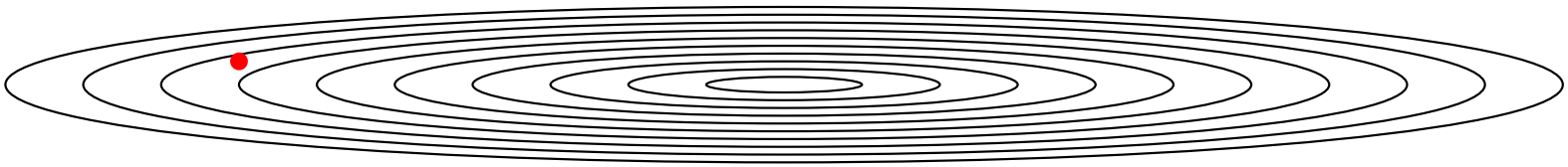
- Before each run through the whole dataset (*epoch*), you want to shuffle the dataset (could be expensive).
- If possible, you want all classes to be equally represented in batches. This is tricky if batches are small, or if classes are very unbalanced.

Technical issues

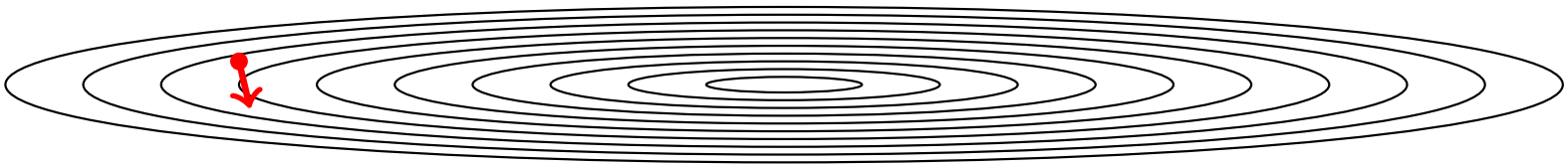
Some important details:

- Before each run through the whole dataset (*epoch*), you want to shuffle the dataset (could be expensive).
- If possible, you want all classes to be equally represented in batches. This is tricky if batches are small, or if classes are very unbalanced.
- The batch size does influence the learning rate, the nature of this relationship is not entirely clear, but experiments suggests that l.r. should be scaled linearly with batch size (up to a certain point, and perhaps not in the initial epochs).

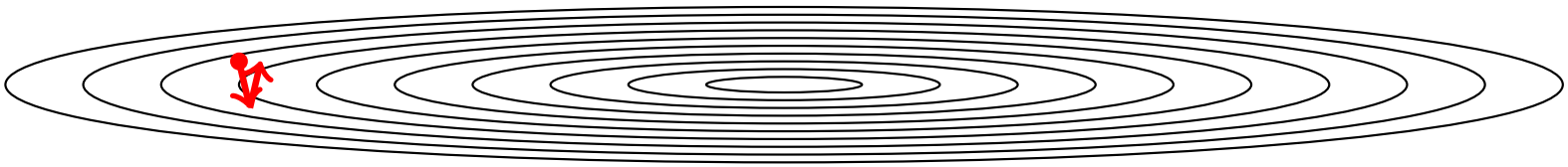
Problems with GD



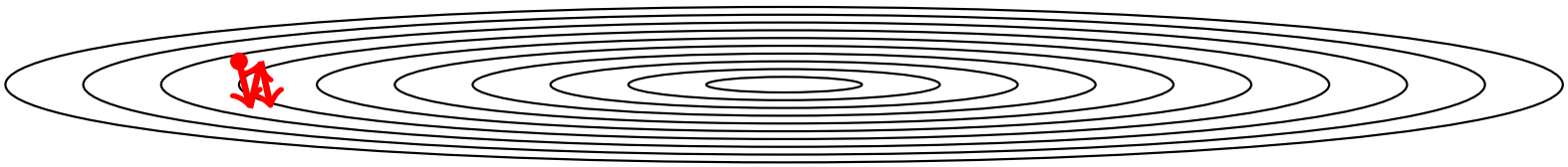
Problems with GD



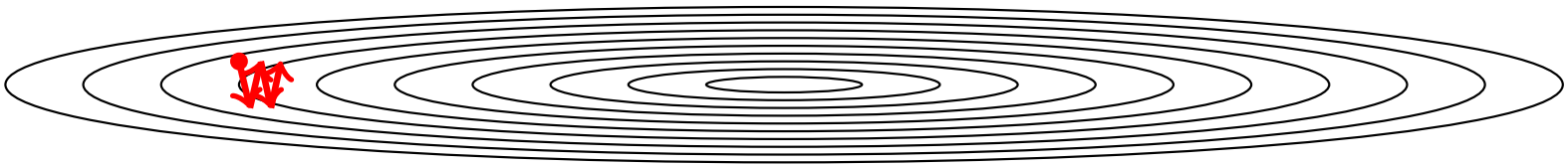
Problems with GD



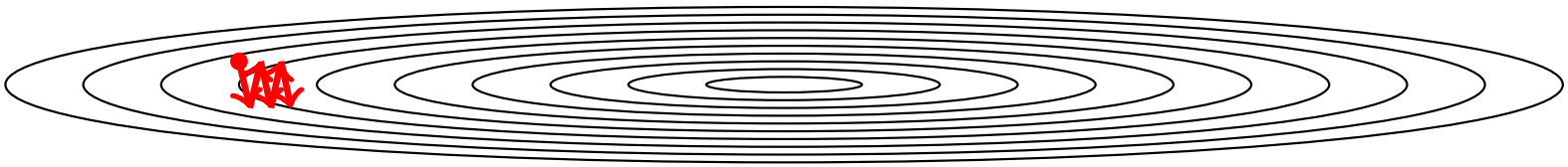
Problems with GD



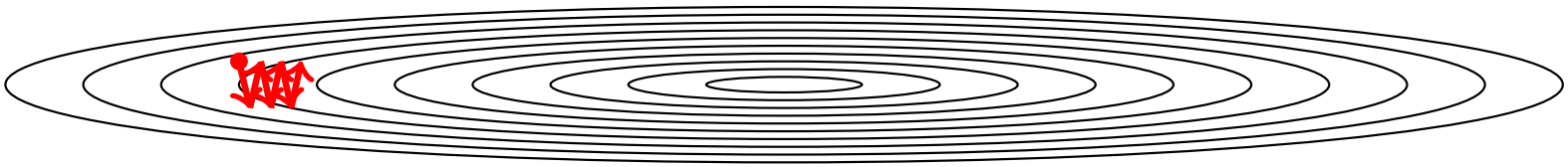
Problems with GD



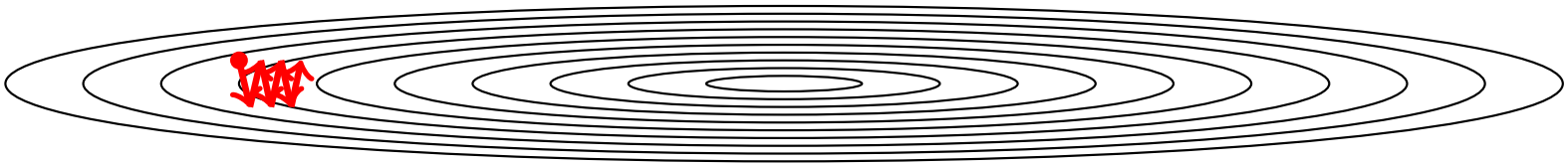
Problems with GD



Problems with GD

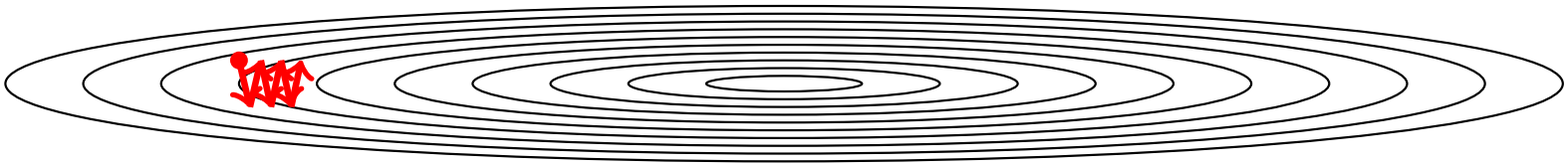


Problems with GD



You can try to avoid this kind of scenario (later).

Problems with GD



You can try to avoid this kind of scenario (later).

Or you can try to deal with it.

Momentum

Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.

Momentum

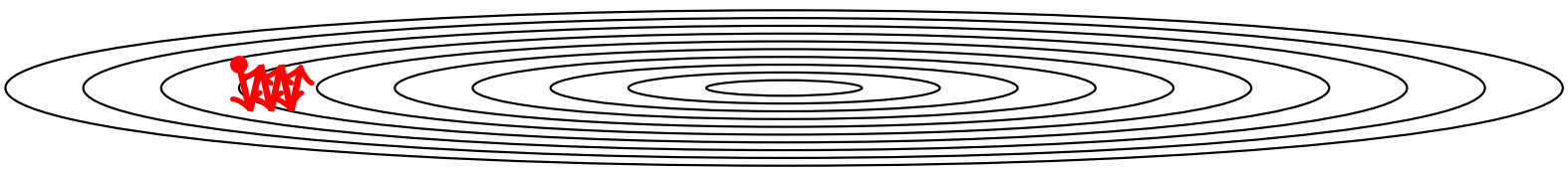
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

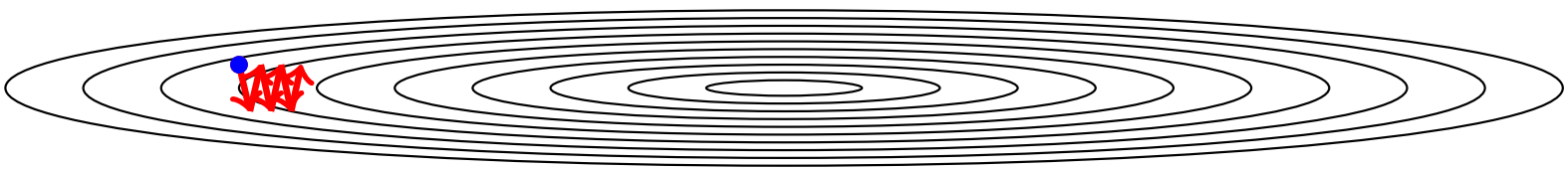
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

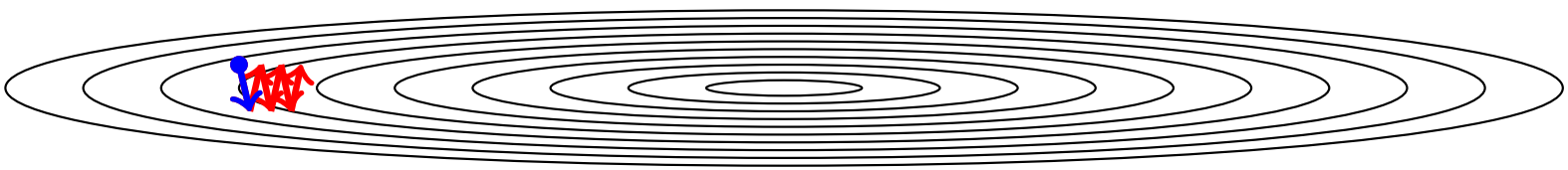
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

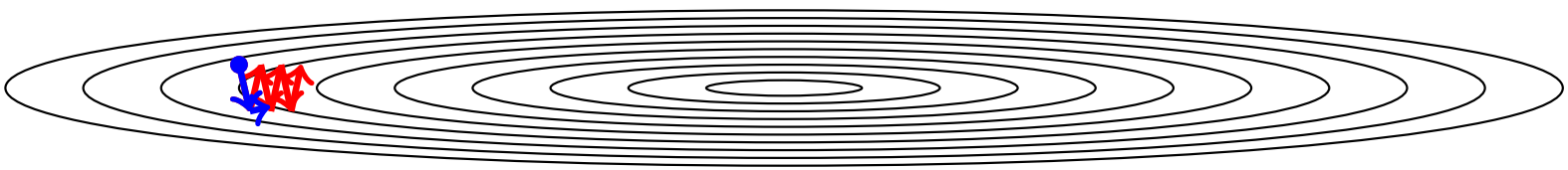
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

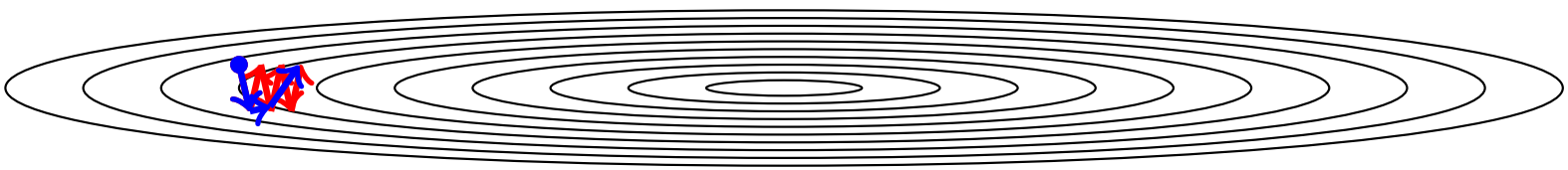
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

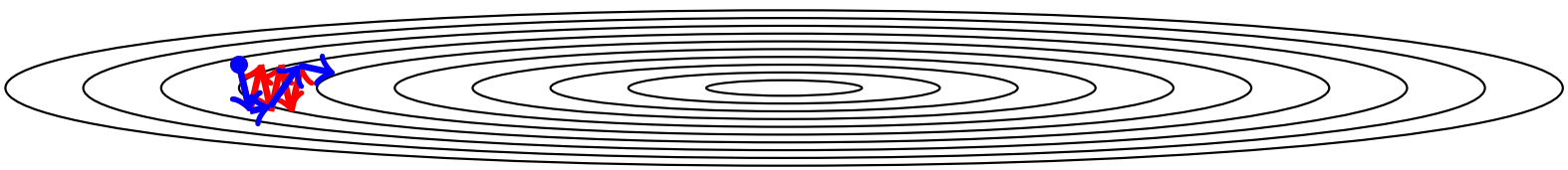
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

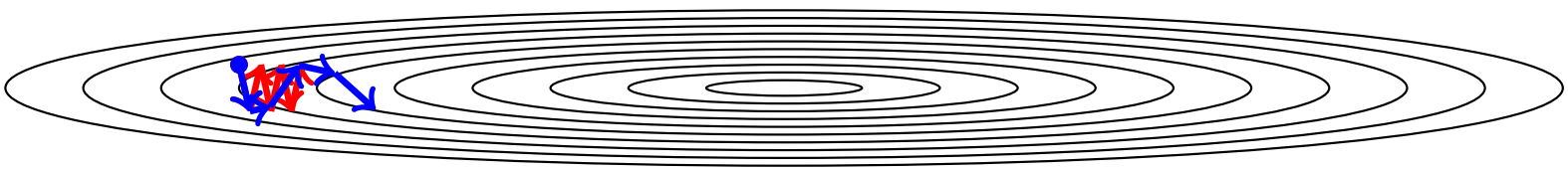
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

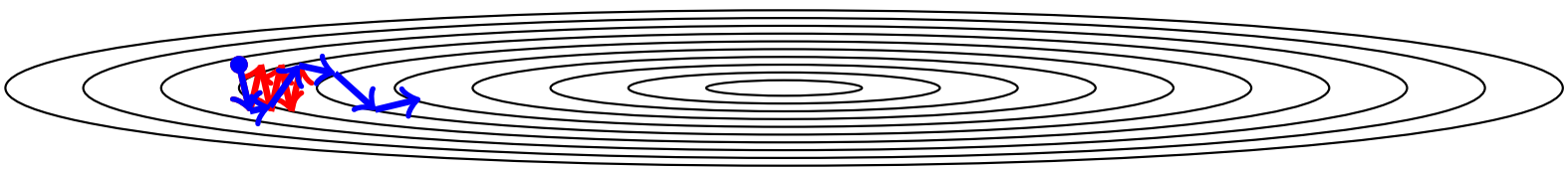
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Momentum

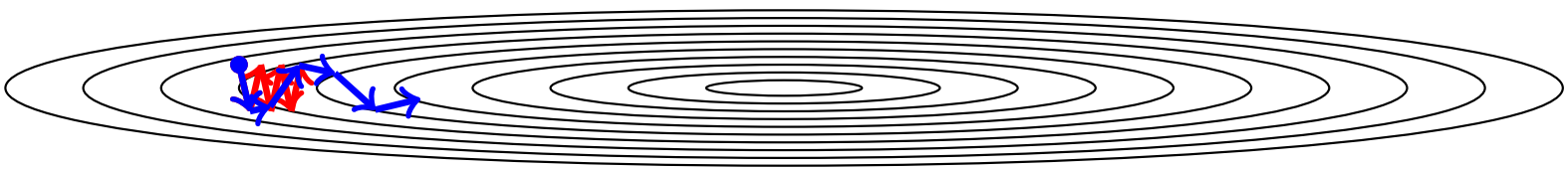
Idea: Accumulate gradients from recent iterations. Update parameters like this:

$$x_{t+1} = x_t + v_{t+1}$$

with

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

where γ is a parameter (momentum), e.g $\gamma = 0.9$. Start with a zero vector v_0.



Note: With momentum you might get different kind of oscillation, usually not a big problem.

Nesterov's momentum

This is the update vector again:

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

$$x_{t+1} = x_t + v_{t+1}.$$

Note that we will always move by γv_t and then some.

Nesterov's momentum

This is the update vector again:

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t),$$

$$x_{t+1} = x_t + v_{t+1}.$$

Note that we will always move by γv_t and then some.

Here is a different approach (*Nesterov's momentum*)

$$v_{t+1} = \gamma v_t - \eta \nabla L(x_t + \gamma v_t).$$

We move first to see what is there!

Nesterov's momentum

Might be useful to reformulate like this:

$$\tilde{x}_t = x_t + \gamma v_t,$$

Nesterov's momentum

Might be useful to reformulate like this:

$$\tilde{x}_t = x_t + \gamma v_t,$$

and then

$$v_{t+1} = \gamma v_t + \eta \nabla L(\tilde{x}_t),$$

Nesterov's momentum

Might be useful to reformulate like this:

$$\tilde{x}_t = x_t + \gamma v_t,$$

and then

$$v_{t+1} = \gamma v_t + \eta \nabla L(\tilde{x}_t),$$

and

$$\tilde{x}_{t+1} = x_{t+1} + \gamma v_{t+1} = x_t + v_{t+1} + \gamma v_{t+1} = \tilde{x}_t + (1+\gamma) v_{t+1} - \gamma v_t.$$

Nesterov's momentum

Might be useful to reformulate like this:

$$\tilde{x}_t = x_t + \gamma v_t,$$

and then

$$v_{t+1} = \gamma v_t + \eta \nabla L(\tilde{x}_t),$$

and

$$\tilde{x}_{t+1} = x_{t+1} + \gamma v_{t+1} = x_t + v_{t+1} + \gamma v_{t+1} = \tilde{x}_t + (1+\gamma) v_{t+1} - \gamma v_t.$$

Often works better.

Learning rate decay

Ideally, we want to use the largest η that does not lead to erratic behaviour.

Learning rate decay

Ideally, we want to use the largest η that does not lead to erratic behaviour.

When closer to optimum, might want to use smaller η to avoid overshooting, etc.

Learning rate decay

Ideally, we want to use the largest η that does not lead to erratic behaviour.

When closer to optimum, might want to use smaller η to avoid overshooting, etc.

Typically one uses an exponential schedule, e.g. decreasing the rate by a constant factor each epoch.

Learning rate decay

Ideally, we want to use the largest η that does not lead to erratic behaviour.

When closer to optimum, might want to use smaller η to avoid overshooting, etc.

Typically one uses an exponential schedule, e.g. decreasing the rate by a constant factor each epoch.

Other options: polynomial schedule, stages, cyclic learning rate. Additional parameters to tune.

Adagrad

Adagrad controls the learning rate for each parameter separately.

Adagrad

Adagrad controls the learning rate for each parameter separately.

Consider a single parameter w. The GD rule for w is

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}.$$

Adagrad

Adagrad controls the learning rate for each parameter separately.

Consider a single parameter w. The GD rule for w is

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}.$$

In Adagrad we use

$$w \leftarrow w - \frac{\eta}{\sqrt{G + \varepsilon}} \frac{\partial L}{\partial w},$$

where G is the sum of squares of all the gradients for w, and ε is just there to avoid division by zero, usually $\varepsilon = 10^{-8}$.

Adagrad

Adagrad controls the learning rate for each parameter separately.

Consider a single parameter w. The GD rule for w is

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}.$$

In Adagrad we use

$$w \leftarrow w - \frac{\eta}{\sqrt{G + \varepsilon}} \frac{\partial L}{\partial w},$$

where G is the sum of squares of all the gradients for w, and ε is just there to avoid division by zero, usually $\varepsilon = 10^{-8}$.

We do it for all parameters, by keeping G matrices and vectors on top of weight and bias matrices.

Adagrad

Advantages of Adagrad:

Adagrad

Advantages of Adagrad:

- Automatic learning rate decay.

Adagrad

Advantages of Adagrad:

- Automatic learning rate decay.
- Boosts rates of rare weights - useful for sparse learning.

Adagrad

Advantages of Adagrad:

- Automatic learning rate decay.
- Boosts rates of rare weights - useful for sparse learning.
- Quickly stabilizes fast moving parameters - controls oscillations.

Adagrad

Advantages of Adagrad:

- Automatic learning rate decay.
- Boosts rates of rare weights - useful for sparse learning.
- Quickly stabilizes fast moving parameters - controls oscillations.
- Boosts slow moving parameters - helps escape narrow ravines and saddles.

Adagrad

Advantages of Adagrad:

- Automatic learning rate decay.
- Boosts rates of rare weights - useful for sparse learning.
- Quickly stabilizes fast moving parameters - controls oscillations.
- Boosts slow moving parameters - helps escape narrow ravines and saddles.

Adagrad

Advantages of Adagrad:

- Automatic learning rate decay.
- Boosts rates of rare weights - useful for sparse learning.
- Quickly stabilizes fast moving parameters - controls oscillations.
- Boosts slow moving parameters - helps escape narrow ravines and saddles.

One problem: Sometimes the learning rates die to quickly.

RMSProp

RMSProp uses an exponential moving average of squares of past updates to help this.

$$G_t = \gamma G_{t-1} + (1 - \gamma) \left(\frac{\partial L}{\partial w} \right)^2 .$$

Typically, $\gamma = 0.9$.

RMSProp

RMSProp uses an exponential moving average of squares of past updates to help this.

$$G_t = \gamma G_{t-1} + (1 - \gamma) \left(\frac{\partial L}{\partial w} \right)^2 .$$

Typically, $\gamma = 0.9$.

Update rule is the same as in Adagrad.

RMSProp

RMSProp uses an exponential moving average of squares of past updates to help this.

$$G_t = \gamma G_{t-1} + (1-\gamma)\left(\frac{\partial L}{\partial w}\right)^2.$$

Typically, $\gamma = 0.9$.

Update rule is the same as in Adagrad.

Very efficient implementation of GD in practice.

Adam

Adam - a recent (2015) algorithm that seems to beat all others in many applications.

Adam

Adam - a recent (2015) algorithm that seems to beat all others in many applications.

It uses an EMA of squared gradients (like RMSProp) as well as EMA of gradients (like momentum methods).

Adam

Adam - a recent (2015) algorithm that seems to beat all others in many applications.

It uses an EMA of squared gradients (like RMSProp) as well as EMA of gradients (like momentum methods).

It also uses special correction terms to fight initial bias towards zero of the EMAs.

Which algorithm?

Vanilla GD and momentum methods require choosing the learning rate and learning rate decay. They also tend to give slower convergence, especially for sparse data.

Which algorithm?

Vanilla GD and momentum methods require choosing the learning rate and learning rate decay. They also tend to give slower convergence, especially for sparse data.

Adagrad based methods often work with less tuning, and in general better, but not always.

Which algorithm?

Vanilla GD and momentum methods require choosing the learning rate and learning rate decay. They also tend to give slower convergence, especially for sparse data.

Adagrad based methods often work with less tuning, and in general better, but not always.

Interestingly, in many top research papers, plain SGD or momentum is used.

Plan

Gradient Descent

Overfitting

Saturation

Batch Normalization

Motivation

Goal: We want our network to generalize - learn to recognize 6s and 9s and not *overfit* to the exact 6s and 9s from the training data.

Motivation

Goal: We want our network to generalize - learn to recognize 6s and 9s and not *overfit* to the exact 6s and 9s from the training data.

Exercise: Consider data with $N = 1000$ different examples in $x^i \in \{-1, 1\}^m$ and binary outputs $y^i \in \{0, 1\}$. Argue that a perceptron network with a single hidden layer of size N can learn to discriminate perfectly. Is this useful?

Motivation

Goal: We want our network to generalize - learn to recognize 6s and 9s and not *overfit* to the exact 6s and 9s from the training data.

Exercise: Consider data with $N = 1000$ different examples in $x^i \in \{-1, 1\}^m$ and binary outputs $y^i \in \{0, 1\}$. Argue that a perceptron network with a single hidden layer of size N can learn to discriminate perfectly. Is this useful?

Solution: The i-th perceptron in the hidden layer triggers only for x_i, by setting $w = x^i$ and $b = m$.

Motivation

Exercise: Can you simulate it using sigmoid units?

Motivation

Exercise: Can you simulate it using sigmoid units?

Solution: Setting the i-th perceptron to $\sum x_k^i x_k - m + \frac{1}{2}$ gives $\frac{1}{2}$ for x^i and at most $-\frac{1}{2}$ for other inputs. To get binary behaviour multiply weights and bias by a huge constant.

Motivation

Exercise: Can you simulate it using sigmoid units?

Solution: Setting the i-th perceptron to $\sum x^i_k x_k - m + \frac{1}{2}$ gives $\frac{1}{2}$ for x^i and at most $-\frac{1}{2}$ for other inputs. To get binary behaviour multiply weights and bias by a huge constant.

You can simulate any perceptron network this way. Bad, because you can achieve "this and nothing else" behaviour.

Motivation II

Another point of view on generalization: general concepts (6 vs a specific 6 from the input data) are resistant to noise.

How to create a network that does not change much when fed noisy data?

Motivation II

Another point of view on generalization: general concepts (6 vs a specific 6 from the input data) are resistant to noise.

How to create a network that does not change much when fed noisy data?

Use small weights!

L2 regularization

Standard way to avoid large weights is to include a penalty term in the objective function:

$$L(W, B) = \frac{1}{n} \sum L_i(W, B) + \lambda \sum_{w \in W} w^2.$$

Note: biases are not regularized!

L2 regularization

Standard way to avoid large weights is to include a penalty term in the objective function:

$$L(W, B) = \frac{1}{n} \sum L_i(W, B) + \lambda \sum_{w \in W} w^2.$$

Note: biases are not regularized!

How to implement this in GD?

L2 regularization - gradient

The gradient of the regularization part over the weights is independent from the mini-batch data and equals

$$\frac{\partial L}{\partial w} = \lambda w.$$

L2 regularization - gradient

The gradient of the regularization part over the weights is independent from the mini-batch data and equals

$$\frac{\partial L}{\partial w} = \lambda w.$$

In effect all we need to do is to multiply all weights by $(1 - \lambda)$ before applying the update - how convenient!

L2 regularization - gradient

The gradient of the regularization part over the weights is independent from the mini-batch data and equals

$$\frac{\partial L}{\partial w} = \lambda w.$$

In effect all we need to do is to multiply all weights by $(1 - \lambda)$ before applying the update - how convenient!

This is sometimes called *weight decay* for obvious reasons.

L1 regularization

Another idea - use L1 norm instead:

$$L(W, B) = \frac{1}{n} \sum L_i(W, B) + \lambda \sum_{w \in W} |w| .$$

L1 regularization

Another idea - use L1 norm instead:

$$L(W, B) = \frac{1}{n} \sum L_i(W, B) + \lambda \sum_{w \in W} |w| .$$

Exercise: Note that this does not really penalize large weights, but only their total! What is the effect on the learned weights?

L1 regularization

Another idea - use L1 norm instead:

$$L(W, B) = \frac{1}{n} \sum L_i(W, B) + \lambda \sum_{w \in W} |w| .$$

Exercise: Note that this does not really penalize large weights, but only their total! What is the effect on the learned weights?

Answer: L1 penalty induces sparseness of weights - only the "important" weights are non-zero, and can even be large, the rest are zero.

L1 regularization

Another idea - use L1 norm instead:

$$L(W, B) = \frac{1}{n} \sum L_i(W, B) + \lambda \sum_{w \in W} |w| .$$

Exercise: Note that this does not really penalize large weights, but only their total! What is the effect on the learned weights?

Answer: L1 penalty induces sparseness of weights - only the "important" weights are non-zero, and can even be large, the rest are zero.

This tends to avoid overfitting as well.

L1 regularization

Another idea - use L1 norm instead:

$$L(W, B) = \frac{1}{n} \sum L_i(W, B) + \lambda \sum_{w \in W} |w| .$$

Exercise: Note that this does not really penalize large weights, but only their total! What is the effect on the learned weights?

Answer: L1 penalty induces sparseness of weights - only the "important" weights are non-zero, and can even be large, the rest are zero.

This tends to avoid overfitting as well.

How to implement this in GD?

L1 regularization - GD

The gradient of the regularization part is again independent from the mini-batch data and equals

$$\frac{\partial L}{\partial w} = \lambda \text{sign}(w).$$

L1 regularization - GD

The gradient of the regularization part is again independent from the mini-batch data and equals

$$\frac{\partial L}{\partial w} = \lambda \text{sign}(w).$$

Before applying the usual update we need to shift all weights towards zero by λ - how convenient again!

L1 regularization - GD

The gradient of the regularization part is again independent from the mini-batch data and equals

$$\frac{\partial L}{\partial w} = \lambda \text{sign}(w).$$

Before applying the usual update we need to shift all weights towards zero by λ - how convenient again!

For $w = 0$ the penalty term is not differentiable, makes sense to not touch the weight. One might argue that weights w with $|w| < \lambda$ should be just set to zero.

Augmentation

The easiest way to avoid overfitting is to get more data - the model can no longer learn specific examples. What if there is no more data?

Augmentation

The easiest way to avoid overfitting is to get more data - the model can no longer learn specific examples. What if there is no more data?

Create synthetic data that *resembles the real data*. This is usually done by perturbing the real data - we make the network resistant to perturbations directly by adding perturbed inputs to the data!

Augmentation

The easiest way to avoid overfitting is to get more data - the model can no longer learn specific examples. What if there is no more data?

Create synthetic data that *resembles the real data*. This is usually done by perturbing the real data - we make the network resistant to perturbations directly by adding perturbed inputs to the data!

Exercise: Propose ways to create synthetic data for MNIST.

Augmentation

The easiest way to avoid overfitting is to get more data - the model can no longer learn specific examples. What if there is no more data?

Create synthetic data that *resembles the real data*. This is usually done by perturbing the real data - we make the network resistant to perturbations directly by adding perturbed inputs to the data!

Exercise: Propose ways to create synthetic data for MNIST.

Answer: There are many reasonable ideas: tiny rotations, shifts.

Dropout

Dropout: In each mini-batch step randomly remove a subset of neurons from the network, e.g. each with $p = \frac{1}{2}$ independently.

Dropout

Dropout: In each mini-batch step randomly remove a subset of neurons from the network, e.g. each with $p = \frac{1}{2}$ independently.

During prediction on test set, use all neurons. This generates f^l that are larger than when training so scale them by $1 - p$.

Dropout

Dropout: In each mini-batch step randomly remove a subset of neurons from the network, e.g. each with $p = \frac{1}{2}$ independently.

During prediction on test set, use all neurons. This generates f^l that are larger than when training so scale them by $1 - p$.

A generalization of this idea is to drop connections, not nodes - we will not discuss details.

Why dropout works?

- "This prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate." – G. Hinton

Why dropout works?

- "This prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate." – G. Hinton
- Averaging predictions over many networks is a good way to boost accuracy. Dropout trains many networks at the same time (sort of).

Why dropout works?

- "This prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate." – G. Hinton
- Averaging predictions over many networks is a good way to boost accuracy. Dropout trains many networks at the same time (sort of).
- Dropout introduces noise, so features train to be more resistant to this kind of noise.

Why dropout works?

- "This prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate." – G. Hinton
- Averaging predictions over many networks is a good way to boost accuracy. Dropout trains many networks at the same time (sort of).
- Dropout introduces noise, so features train to be more resistant to this kind of noise.
- Problem can have an obvious plateau of an easy solution. Dropout (especially on the input layer) can force the network to look beyond that.

Plan

Gradient Descent

Overfitting

Saturation

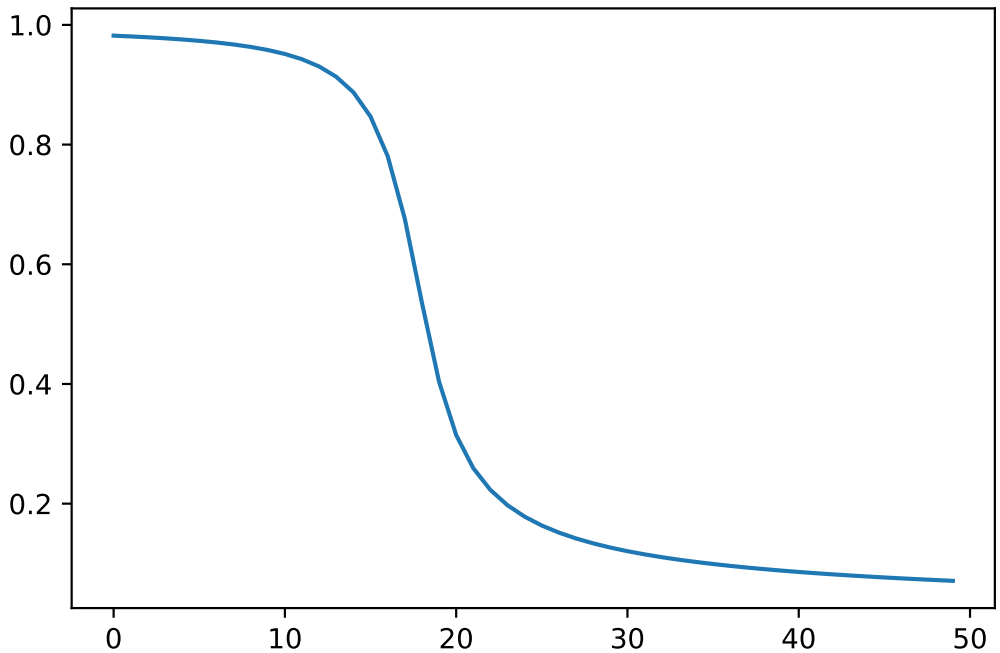
Batch Normalization

Motivation

Consider a single sigmoid unit trying to learn on a single example $x = 1, y = 0$. The initial weight is $w = 2.0$ and bias is $b = 2.0$ (initial prediction is $\frac{1}{1+e^{-5}} \approx 0.99$). We use a quadratic loss function. This is how loss behaves over 50 epochs.

Motivation

Consider a single sigmoid unit trying to learn on a single example $x = 1, y = 0$. The initial weight is $w = 2.0$ and bias is $b = 2.0$ (initial prediction is $\frac{1}{1+e^{-5}} \approx 0.99$). We use a quadratic loss function. This is how loss behaves over 50 epochs.



This unit is a *saturated* unit.

Motivation

What is the reason for this weird behaviour?

Motivation

What is the reason for this weird behaviour?

$$L(w, b) = (\sigma(w + b))^2 .$$

Motivation

What is the reason for this weird behaviour?

$$L(w, b) = (\sigma(w + b))^2 .$$

$$\frac{\partial L(w, b)}{\partial w} = 2\sigma(w + b) \cdot \sigma(w + b)\,(1 - \sigma(w + b)) .$$

The second term makes this really small when prediction is close to one - very counter-intuitive.

Log-loss

What happens if we use the log-loss instead?

Log-loss

What happens if we use the log-loss instead?

$$L(w, b) = -\log\left(1 - \sigma(w + b)\right).$$

Log-loss

What happens if we use the log-loss instead?

$$L(w, b) = -\log\left(1 - \sigma(w + b)\right).$$

$$\frac{\partial L(w, b)}{\partial w} = \frac{1}{1 - \sigma(w + b)} \cdot \sigma(w + b)\left(1 - \sigma(w + b)\right) = \sigma(w + b).$$

Log-loss

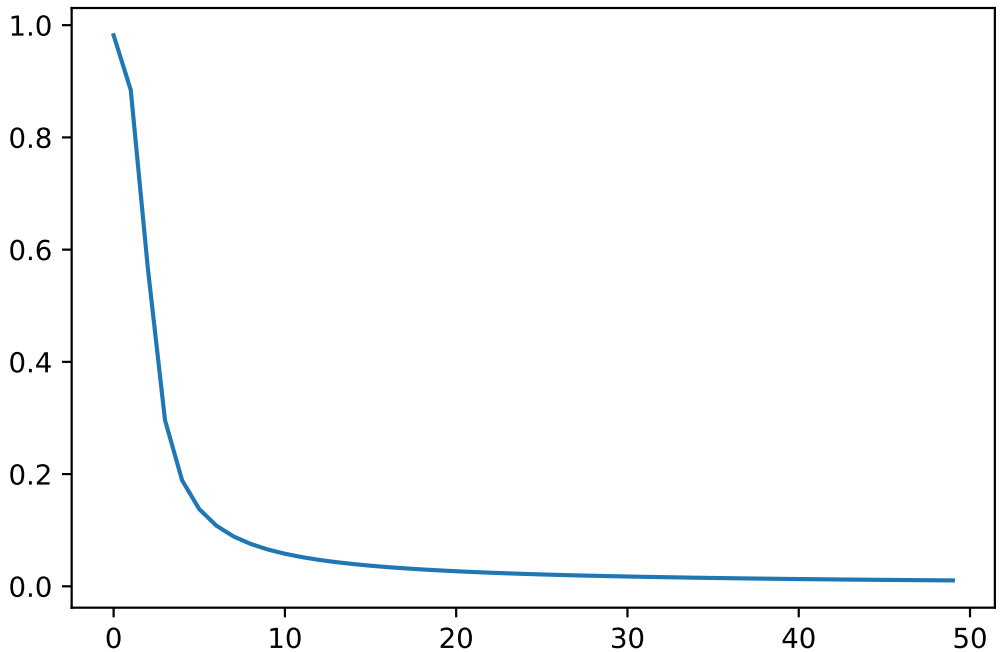
What happens if we use the log-loss instead?

$$L(w, b) = -\log\left(1 - \sigma(w + b)\right).$$

$$\frac{\partial L(w, b)}{\partial w} = \frac{1}{1 - \sigma(w + b)} \cdot \sigma(w + b)\left(1 - \sigma(w + b)\right) = \sigma(w + b).$$

This is perfect!

Log-loss



The message

What does it all mean?

The message

What does it all mean?

Sigmoid (and tanh) units can learn in a very strange way when using GD. But sometimes you can help it.

The message

What does it all mean?

Sigmoid (and tanh) units can learn in a very strange way when using GD. But sometimes you can help it.

When using sigmoid (or tanh) activations in the output layer, L2 loss is a bad idea. Note: loss is often dictated by other considerations.

The message

What does it all mean?

Sigmoid (and tanh) units can learn in a very strange way when using GD. But sometimes you can help it.

When using sigmoid (or tanh) activations in the output layer, L2 loss is a bad idea. Note: loss is often dictated by other considerations.

Real message: when facing difficulties, think about what GD is really doing and figure out the problem!

Exercise

Exercise: Consider the first hidden layer of sigmoid units. Can saturation happen there? Can you avoid it?

Exercise

Exercise: Consider the first hidden layer of sigmoid units. Can saturation happen there? Can you avoid it?

Answer: Of course it can. You can control the initial values in f^l as follows:

- Set biases to 0.
- Rescale inputs so that they are zero-centered and have unit variance.
- Set initial weights to small random values. How small?

Exercise, ctd

Consider input vector $X_1, \ldots, X_m$ and weights $W_1, \ldots, W_m$.

Exercise, ctd

Consider input vector $X_1, \ldots, X_m$ and weights $W_1, \ldots, W_m$.

When $EX_i = 0$ and $EW_i = 0$ and they are independent, we have $\mathrm{Var}(X_i W_i) = \mathrm{Var} X_i \mathrm{Var} W_i = \mathrm{Var} W_i$.

Exercise, ctd

Consider input vector $X_1, \ldots, X_m$ and weights $W_1, \ldots, W_m$.

When $EX_i = 0$ and $EW_i = 0$ and they are independent, we have $\mathrm{Var}(X_i W_i) = \mathrm{Var} X_i \mathrm{Var} W_i = \mathrm{Var} W_i$.

So $\mathrm{Var} \sum (X_i W_i) = \sum \mathrm{Var} W_i$. Obvious solution is to set $\mathrm{Var} W_i = \frac{1}{m}$, or $sd(W_i) = \frac{1}{\sqrt{m}}$.

Exercise, ctd

Consider input vector $X_1, \ldots, X_m$ and weights $W_1, \ldots, W_m$.

When $EX_i = 0$ and $EW_i = 0$ and they are independent, we have $\mathrm{Var}(X_i W_i) = \mathrm{Var} X_i \mathrm{Var} W_i = \mathrm{Var} W_i$.

So $\mathrm{Var} \sum (X_i W_i) = \sum \mathrm{Var} W_i$. Obvious solution is to set $\mathrm{Var} W_i = \frac{1}{m}$, or $sd(W_i) = \frac{1}{\sqrt{m}}$.

The same reasoning can be repeated for all layers, not only input layers. You can also use a similar reasoning for backpropagation step, and get inverse of the number of outputs.

Sigmoid units?

Because of saturation problems, sigmoid units are very rarely used nowadays, especially in deep networks. Alternatives: ReLU, max-out and other.

Sigmoid units?

Because of saturation problems, sigmoid units are very rarely used nowadays, especially in deep networks. Alternatives: ReLU, max-out and other.

This does not solve the problem completely - instead of saturation we get dead units etc. Reasoning like this is still very useful.

Plan

Gradient Descent

Overfitting

Saturation

Batch Normalization

Motivation

Each layer tries to learn a function, but the inputs are changing! - *(internal) covariate shift*.

Motivation

Each layer tries to learn a function, but the inputs are changing! - *(internal) covariate shift*.

Idea: Normalize mean and variance of f^l.

Motivation

Each layer tries to learn a function, but the inputs are changing! - *(internal) covariate shift*.

Idea: Normalize mean and variance of f^l.

Explanation: f^l as combinations of many factors are likely to resemble Gaussians (CLT). By normalizing we keep the distribution of inputs to g^l stable.

Motivation

Each layer tries to learn a function, but the inputs are changing! - *(internal) covariate shift*.

Idea: Normalize mean and variance of f^l.

Explanation: f^l as combinations of many factors are likely to resemble Gaussians (CLT). By normalizing we keep the distribution of inputs to g^l stable.

Exercise: Why it makes less sense to normalize g^l? Give a handwaving argument.

Motivation

Each layer tries to learn a function, but the inputs are changing! - *(internal) covariate shift*.

Idea: Normalize mean and variance of f^l.

Explanation: f^l as combinations of many factors are likely to resemble Gaussians (CLT). By normalizing we keep the distribution of inputs to g^l stable.

Exercise: Why it makes less sense to normalize g^l? Give a handwaving argument.

Answer: Outputs of g^l can have rather complicated distributions. Scaling and shifting might not guarantee stable distributions. That said, recent papers actually advocate this. Apparently it often actually works better!

Implementation

First idea might be to manually normalize f^l outside the backpropagation engine.

Implementation

First idea might be to manually normalize f^l outside the backpropagation engine.

No longer clear what is really happening inside.

Implementation

First idea might be to manually normalize f^l outside the backpropagation engine.

No longer clear what is really happening inside.

Also, one can argue that the network will learn to counteract such measures. Instead...

Solution

Introduce an intermediate computation between f^l and g^l in our graph, denoted BN^l.

Solution

Introduce an intermediate computation between f^l and g^l in our graph, denoted BN^l.

Consider a layer l and a mini-batch of size s.

$$\mu_i^l = \frac{1}{s} \sum_{j=1}^{s} f_{i,j}^l.$$

So, μ^l is a vector of mini-batch averages for each linearity.

Solution

Introduce an intermediate computation between f^l and g^l in our graph, denoted BN^l.

Consider a layer l and a mini-batch of size s.

$$\mu_i^l = \frac{1}{s}\sum_{j=1}^{s} f_{i,j}^l.$$

So, μ^l is a vector of mini-batch averages for each linearity.

$$v_i^l = \frac{1}{s}\sum_{j=1}^{s}(f_{i,j}^l - \mu_i^l)^2.$$

This is a vector of mini-batch estimates of variances of each f_i^l.

Solution, ctd.

Can we set $BN^l = \frac{f^l - \mu^l}{\sqrt{v^l + \varepsilon}}$ and activate g^l on BN^l?

Solution, ctd.

Can we set $BN^l = \frac{f^l - \mu^l}{\sqrt{v^l + \varepsilon}}$ and activate g^l on BN^l?

This is bad since then the non-linearities will receive restricted inputs. Instead $BN^l = \gamma^l \frac{f^l - \mu^l}{\sqrt{v^l + \varepsilon}} + \beta^l$ and we activate g^l on these BN^l. Note that each node has its own γ and β - learned parameters.

Solution, ctd.

Can we set $BN^l = \frac{f^l - \mu^l}{\sqrt{v^l + \varepsilon}}$ and activate g^l on BN^l?

This is bad since then the non-linearities will receive restricted inputs. Instead $BN^l = \gamma^l \frac{f^l - \mu^l}{\sqrt{v^l + \varepsilon}} + \beta^l$ and we activate g^l on these BN^l. Note that each node has its own γ and β - learned parameters.

This is a more complicated computation graph: each $BN^l_{i,j}$ depends on all $f^l_{i,j}$, but still a DAG. Can compute gradients of L over g^l, BN^l and f^l, and consequently over w's, b's, γ's and β's (lab).

Extra details

When using batch normalization, we do not need biases, they get removed anyway. β's acts as biases.

Extra details

When using batch normalization, we do not need biases, they get removed anyway. β's acts as biases.

When performing prediction, one might consider computing global mean and variance for the whole dataset (or at least better estimates than a single batch) and using them instead of batch statistics. Unfortunately this introduces a discrepancy between training and testing (same as dropout).

Extra details

When using batch normalization, we do not need biases, they get removed anyway. β's acts as biases.

When performing prediction, one might consider computing global mean and variance for the whole dataset (or at least better estimates than a single batch) and using them instead of batch statistics. Unfortunately this introduces a discrepancy between training and testing (same as dropout).

In a recent paper, Laarhoven [2017] argues that:

- If using batch normalization, using L2 regularization as well might be crucial for convergence, BUT
- The L2 regularization coefficient only influences effective learning rate!

Advantages

Advantages

- Speeds up learning.

Advantages

- Speeds up learning.
- Allows for higher learning rates (more stable gradients).

Advantages

- Speeds up learning.
- Allows for higher learning rates (more stable gradients).
- Acts as a regularizer (can even be used as the only regularizer).

Advantages

- Speeds up learning.
- Allows for higher learning rates (more stable gradients).
- Acts as a regularizer (can even be used as the only regularizer).
- Avoids saturation for sigmoid and tanh units and dying ReLU units. This makes training deeper networks possible.

Advantages

- Speeds up learning.
- Allows for higher learning rates (more stable gradients).
- Acts as a regularizer (can even be used as the only regularizer).
- Avoids saturation for sigmoid and tanh units and dying ReLU units. This makes training deeper networks possible.
- No careful initialization required.