Biological Applications of Deep Learning
Lecture 6

Alexander Schönhuth

UNIVERSITÄT BIELEFELD

Faculty of Technology

Bielefeld University

November 16, 2022

Contents Today

- Recurrent Neural Networks
  - Long Short Term Memory (LSTM) Networks
- Vanishing Gradients
- Batch Normalization

Recurrent Neural Networks

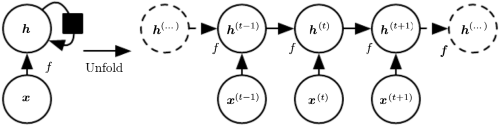
- Unlike CNNs, which specialize in processing grid-/matrix-style input, *recurrent neural networks (RNNs)* specialize in processing sequences of values

$$\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)} \tag{1}$$

- *Advantages*:
  - RNNs can process very long sequences
  - RNNs can process sequences of flexible length
- To make this possible, they also employ *parameter sharing*
- *Additional literature*: "Supervised Sequence Labelling with Recurrent Neural Networks", A. Graves, 2012, `https://www.springer.com/de/book/9783642247965`

Recurrent Neural Networks

Architecture



RNN with one hidden layer and no outputs

- Generating values:

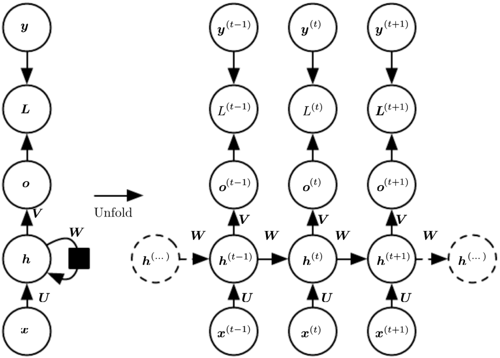
$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta) \tag{2}$$

- *Recurrence*:

$$\begin{aligned}\mathbf{h}^{(t)} &= f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta) = f(f(\mathbf{h}^{(t-2)}, \mathbf{x}^{(t-1)}; \theta), \mathbf{x}^{(t)}; \theta) \\ &= f(f(...(f(\mathbf{h}^{(0)}, \mathbf{x}^{(1)}; \theta), \mathbf{x}^{(2)}; \theta)...), \mathbf{x}^{(t)}; \theta) \\ &=: g^{(t)}(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, ..., \mathbf{x}^{(1)}; \theta)\end{aligned} \tag{3}$$

Recurrent Neural Networks

Architecture



Output at each time step, recurrent connections between hidden units

Recurrent Neural Networks

Forward Propagation

Let σ be a suitable activation function. Then forward propagation in RNN's of the type from the slide before proceeds as follows:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} \tag{4}$$

$$\mathbf{h}^{(t)} = \sigma(\mathbf{a}^{(t)}) \tag{5}$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \tag{6}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}) \tag{7}$$

where $\mathbf{b}, \mathbf{c}$ are the bias vectors along with $\mathbf{W}, \mathbf{U}$ and $\mathbf{V}$, respectively.

$\mathbf{b}, \mathbf{c}, \mathbf{U}, \mathbf{V}, \mathbf{W}$ are to be learnt

Recurrent Neural Networks

Computing Cost

- Let $\mathbf{y} = (y^{(1)}, ..., y^{(t)})$ be true labels for the sequence $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}$.
- Then we compute the cost C as

$$C(\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(\tau)}\}, \{y^{(1)}, ..., y^{(\tau)}\}) = \sum_{t=1}^{\tau} C^{(t)} \tag{8}$$

  where

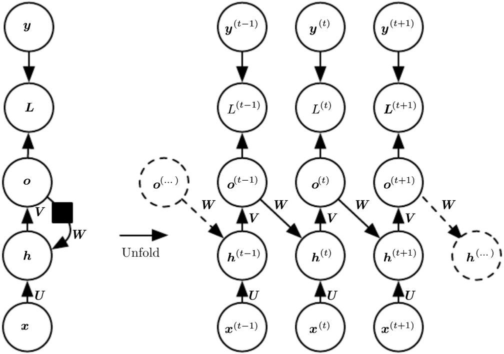
$$C^{(t)} = -\log p_{\text{model}}(y^{(t)} \mid \{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}\}) \tag{9}$$

  and p_{model} refers to the probability computed by application of *softmax* to $\mathbf{o}^{(t)}$, see (7).

- Computing gradients does not involve any particular complications.
- See `http://www.deeplearningbook.org/contents/rnn.html`, 10.2.2 (☞ *Homework* if you wish)

Recurrent Neural Networks

Architecture II

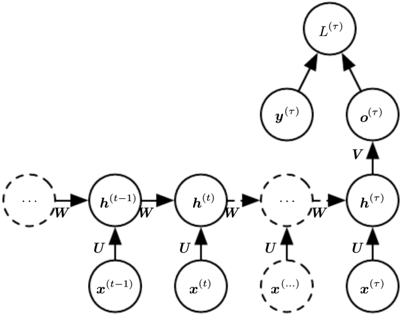


Less powerful, but easier to train:
RNNs where output units connect to hidden units

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{o}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} \tag{10}$$

Recurrent Neural Networks

Architecture II

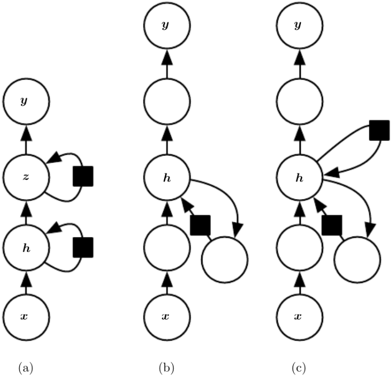


RNN that generates one, summarizing output

- Used for generation of fixed-size representation
- Further used as input for further processing

Recurrent Neural Networks

Going Deep



(a) Extra layer of hidden units, all operating the same way

(b) Introduction of units between hidden units

(c) Like (b), but with skip connections

Recurrent Neural Networks

Motivation / Summary

- *Time Dynamics*: Model that behaviour of a network may vary over time
- *Recurrence*: Model that output derived from input may depend on inputs seen earlier
- Particularly useful for analyzing data / processes that change over time
  - Speech recognition
  - Natural language processing
- NNs have trouble solving certain problems conventional approaches are good at and vice versa
- *Recurrent Neural Networks* are an attempt to have a unifying model that is good at everything

Recurrent Neural Networks

Further Models

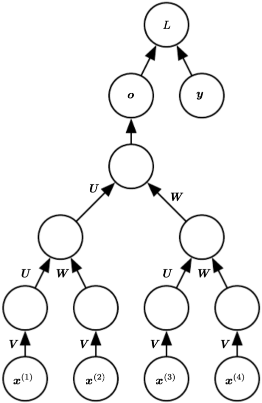
- *Bidirectional RNNs*
  - For computing $\mathbf{h}^{(s)}$ take both earlier ($t = 1, ..., s-1$) and later ($t = s+1, ..., \tau$) values into account
  - Successful in handwriting and speech recognition
- *Encoder-Decoder Sequence to Sequence Architectures*
  - Ordinary RNNs map sequences to sequences of same length
  - Encoder-Decoder RNNs map sequences to sequences of not necessarily the same length
  - Applications: Translations, question answering

Transformers: We will get to that later...

- See `http://www.deeplearningbook.org/contents/rnn.html`, 10.3, 10.4

Recurrent Neural Networks

Recursive Neural Networks



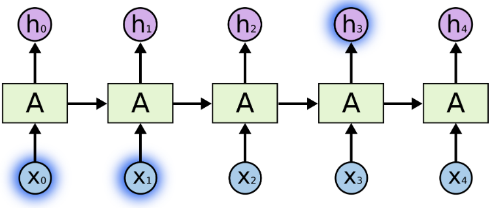
Recursive Net

- Generalize RNNs from sequence-style to tree-shaped input
- Inputs are transformed into outputs according to a hierarchical structure
- *Applications*: Process data structures as input to NNs; language processing; computer vision
- See `http://www.deeplearningbook.org/contents/rnn.html`, 10.6

Long Short Term Memory (LSTM) Networks

Long Short Term Memory Networks

Motivation I



Short term memory: predict h_3 from x_0 and x_1

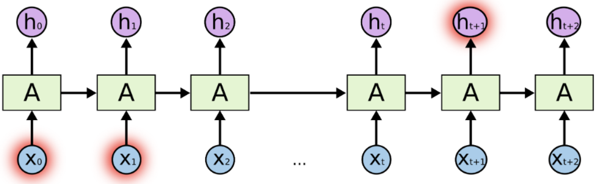
From `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- RNN's have short-term memory
- Predicting h_{t+k} from x_t possible for small enough k
- *Example:* Predict *sky* as last word in

"the clouds are in the ..."

Long Short Term Memory Networks

Motivation II



Predicting h_{t+k} from x_t no longer possible for large k

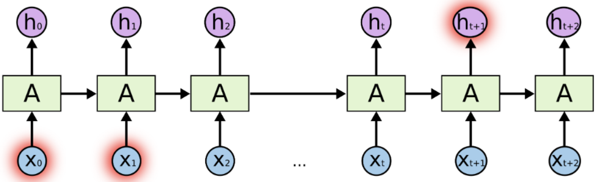
From `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- RNN's have weak long-term memory
- Predicting h_{t+k} from x_t not possible for large k
- *Example:* Predict *French* as last word in sentences

"I grew up in France. [...several sentences...] I speak fluently ... ."

Long Short Term Memory Networks

Motivation III



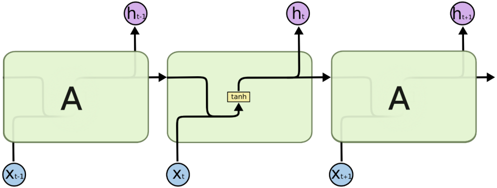
Predicting h_{t+k} from x_t no longer possible for large k

From `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- Memory fading on increasing input length
- In theory, RNN's have long-term memory
- In practice, however, they do not

Long Short Term Memory Networks

Idea I



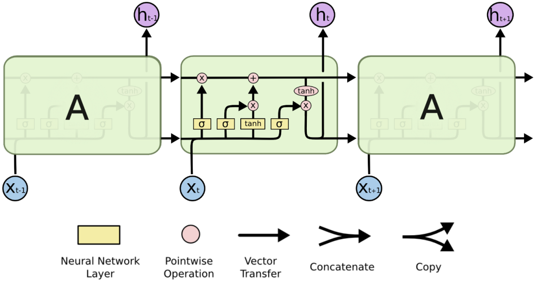
LSTM's: Operation $f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta)$ as cell A

From https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- Consider the joining operation $f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta)$ as a cell A
- *Idea:* Modify A to increase memory duration
- In particular, in A, maintain *cell state* C_t:
  - C_t is like "conveyor belt"
  - C_t keeps things in mind unchanged
  - C_t only changed when changes imperative

Long Short Term Memory Networks

Definition



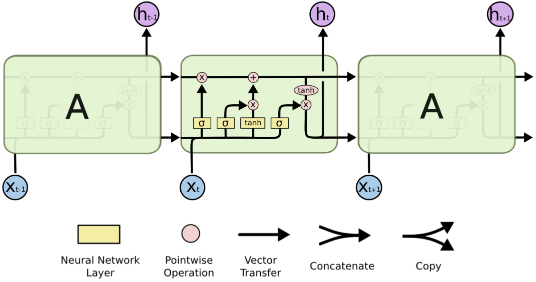
LSTM: Cell A has four interacting neural network layers

From `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- Four different neural network layers $f_i(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta_i), i = 1, 2, 3, 4$
  - Each indicated by yellow box
  - Simplest version: each f_t reflects one neuron
  - Separate learned parameters $\theta_1, \theta_2, \theta_3, \theta_4$

Long Short Term Memory Networks

Definition



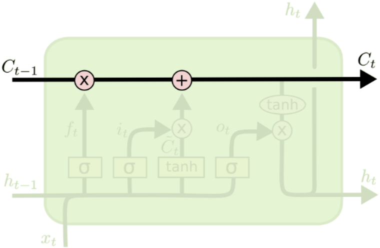
LSTM: Cell A has four interacting neural network layers

From `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- Upper line: cell state C_t
- Each $f_i(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta_i)$ has particular influence on C_t
  - As per arrangement possible scenario: no f_i changes C_t
  - So, C_t may remain unchanged in some cells

Long Short Term Memory Networks

Cell State



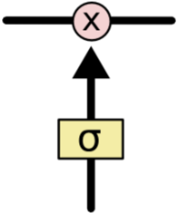
LSTM's: Cell state C_t

From `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- Cell state C_t is like conveyor belt
- Runs straight through cell A, with only minor interactions
  - Information can flow unchanged
- LSTM adds / removes information, regulated by *gates*

Long Short Term Memory Networks

Gates



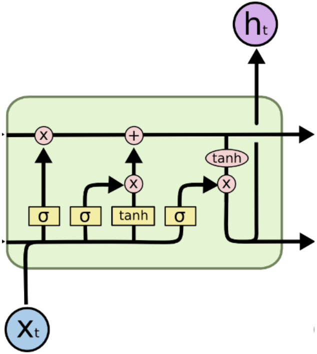
LSTM's: Gate structure

From https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- Gates control flow of information
- Gates consist of
  - Sigmoid neural net layer
  - Pointwise multiplication operation (earlier: Hadamard product)
- Values between 0 and 1
  - Values near 0: remove information
  - Values near 1: let information pass

Long Short Term Memory Networks

Different Gates



- Leftmost gate: *forget gate*
  - Removes information from C_t
- Middle gate: *input gate*
  - Adds new information to C_t
- Rightmost gate: *output gate*
  - Uses (already modified) C_t to control output to next cell

LSTM cell: three different gates (where are they? – spot them...)

From `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

LSTMs: Summary

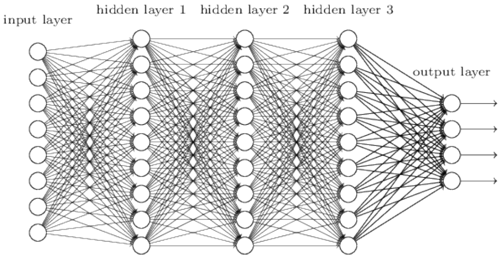
- Looking tricky at first glance, but...
- ... all recent successes of RNN's achieved by LSTM's
- LSTM's have substantially longer memory than ordinary RNN's
- Further advances:
  - Grid LSTM's [Kalchbrenner et al., 2015]
  - Attention networks; we will get to that later...
- Further references:
  - `https://www.deeplearningbook.org/`, 10.10
  - `http://d2l.ai/`, 10.1, 10.2

The Vanishing Gradient

Why is Deep Learning Tough?

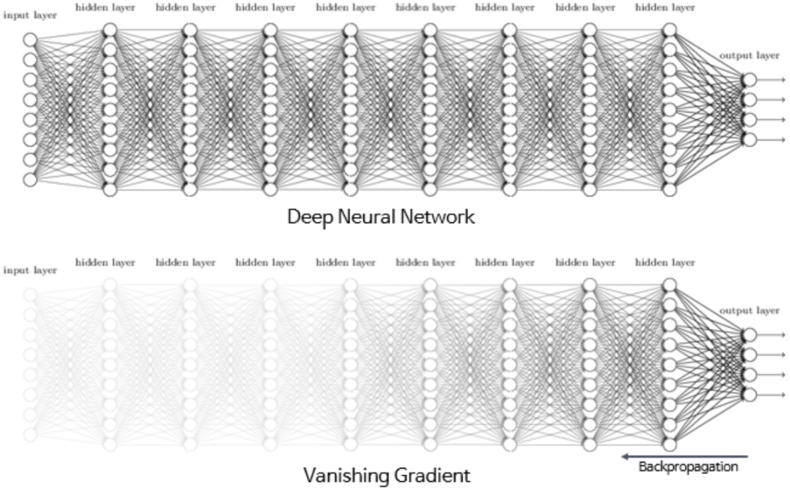
- Deep is supposed to better than shallow
  - Less hidden nodes necessary to approximate the true functional relationship
  - See the “Universal Approximation Theorem” by Montufar, 2014
  - See further “Learning Deep Architectures”, Bengio, 2009, `http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf` for a more informal discussion
- *However*: On increasing depth in a naive way, performance usually drops
- What is going wrong?

WHY IS DEEP LEARNING TOUGH?



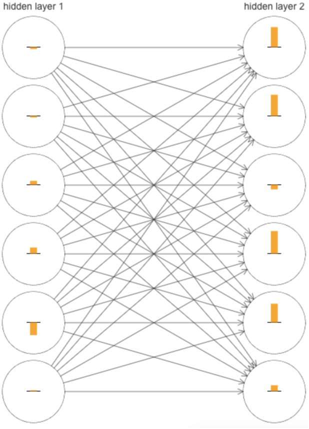
Training Deeper NN's: either the earlier layers (more common; here hidden layer 1) or the later layers (here: hidden layer 3) do not train well

THE VANISHING GRADIENT PROBLEM



Most commonly: gradients converge to zero in earlier layers

The Vanishing Gradient Problem



- Changes larger in later hidden layer
- Learning works better in later layers
- Are neurons likely to learn at different rates in different layers in general?

Yellow bars: $\frac{\partial C}{\partial b}$ for each hidden neuron

The Vanishing Gradient Problem

- Let b_j^l be the j-th bias in layer l, and $\frac{\partial C}{\partial b_j^l}$ be the respective partial derivative of the cost C.
- Let
$$\nabla_{\mathbf{b}^l}^{(l)} C := \left(\frac{\partial C}{\partial b_1^l}, ..., \frac{\partial C}{\partial b_{d(l)}^l}\right) \tag{11}$$
- Then, in the example from the slide before:
$$||\nabla_{\mathbf{b}}^{(1)} C|| = 0.07 \text{ and } ||\nabla_{\mathbf{b}}^{(2)} C|| = 0.31 \tag{12}$$

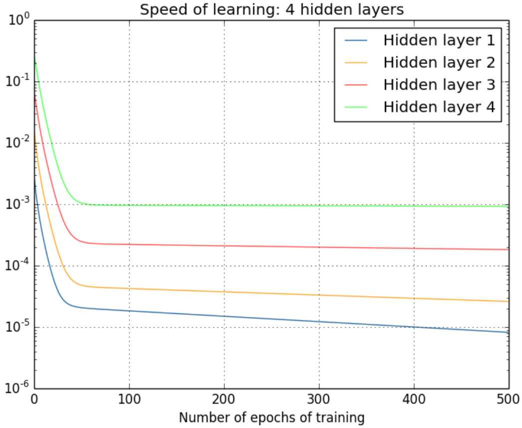
The Vanishing Gradient Problem

- Then, in this example,

$$||\nabla_{\mathbf{b}}^{(1)}C|| = 0.07 \text{ and } ||\nabla_{\mathbf{b}}^{(2)}C|| = 0.31 \tag{13}$$

- Formal quantification shows: learning faster in hidden layer 2.
- When running the identical training task (MNIST), we obtain
  - $||\nabla_{\mathbf{b}}^{(1)}C|| = 0.012, ||\nabla_{\mathbf{b}}^{(2)}C|| = 0.06, ||\nabla_{\mathbf{b}}^{(3)}C|| = 0.283$ for three hidden layers
  - $||\nabla_{\mathbf{b}}^{(1)}C|| = 0.003, ||\nabla_{\mathbf{b}}^{(2)}C|| = 0.017, ||\nabla_{\mathbf{b}}^{(3)}C|| = 0.07, ||\nabla_{\mathbf{b}}^{(4)}C|| = 0.285$ for four hidden layers
  - and so on...

The Vanishing Gradient Problem



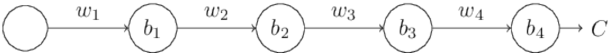
Training speed in [784,30,30,30,30,10]-NN on MNIST

The Vanishing Gradient Problem

- *Vanishing gradient problem*: Neurons in earlier layers learn more slowly
- *Exploding gradient problem*: Neurons in earlier layers learn faster
- In general, gradients in NN's are unstable across layers
- And: vanishing gradients do not mean that there is nothing left to be learnt
- ☞ Fundamental problem for gradient-based learning in NN's

The Vanishing Gradient Problem

Explanation



Simple NN with 3 hidden layers of one neuron each

Let w_1, w_2, w_3, w_4 be the weights, b_1, b_2, b_3, b_4 be the biases and C the cost. Let all neurons be sigmoid, so the output a_j from the j-the neuron is $\sigma(z_j)$ where $z_j = w_j a_{j-1} + b_j$ is the input of the j-th neuron (notation as usual earlier).

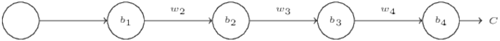
For understanding the Vanishing Gradient Problem, consider $\frac{\partial C}{\partial b_1}$. By repeated application of the backpropagation rules, we see that

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4} \qquad (14)$$

The Vanishing Gradient Problem

Explanation

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



Computing $\frac{\partial C}{\partial b_1}$

There is an alternative explanation for (14). Let Δ indicate small changes. We know that

$$\frac{\partial C}{\partial b_1} \approx \frac{\Delta C}{\Delta b_1} \tag{15}$$

From $a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b_1)$ we further obtain

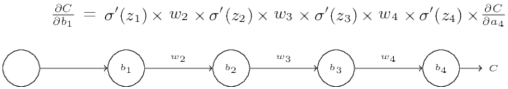
$$\Delta a_1 \approx \frac{\partial \sigma(w_1 a_0 + b_1)}{\partial b_1} \Delta b_1 = \sigma'(z_1) \Delta b_1 \tag{16}$$

further leading to

$$\Delta z_2 \approx \frac{\partial z_2}{\partial a_1} \Delta a_1 = w_2 \Delta a_1 \quad \text{implying} \quad \Delta z_2 \approx \sigma'(z_1) w_2 \Delta b_1 \tag{17}$$

The Vanishing Gradient Problem

Explanation



Computing $\frac{\partial C}{\partial b_1}$

Repeated application of the computations from the slide before eventually yield

$$\Delta C \approx \sigma'(z_1) w_2 \sigma'(z_2) \ldots \sigma'(z_4) \frac{\partial C}{\partial a_4} \Delta b_1 \tag{18}$$

Dividing by b_1 results in the desired expression (14):

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4} \tag{19}$$

The Vanishing Gradient Problem

Explanation

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4} \qquad (20)$$

Except from the last term, this is a product of terms of the form

$$w_j \sigma'(z_j) \qquad (21)$$

It holds that $0 \leq \sigma'(z_j) \leq 1/4$, while, in practice, when employing standard initialization of weights, typically $|w_j| < 1$, so

$$|w_j \sigma'(z_j)| \leq \frac{1}{4} \qquad (22)$$

so in combination

$$\frac{\partial C}{\partial b_1} \leq \sigma'(z_1)(\frac{1}{4})^3 \frac{\partial C}{\partial a_4} \qquad (23)$$

The Vanishing Gradient Problem

Explanation

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\overbrace{w_2\sigma'(z_2)}^{<\frac{1}{4}}\ \overbrace{w_3\sigma'(z_3)}^{<\frac{1}{4}}\ \underbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}_{\updownarrow\ \text{common terms}}$$

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3)\overbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

Comparing $\frac{\partial C}{\partial b_1}$ with $\frac{\partial C}{\partial b_3}$

So, $\frac{\partial C}{\partial b_1}$ is about a factor of 16 (or more) smaller than $\frac{\partial C}{\partial b_3}$. Similar conclusions are drawn for $\frac{\partial C}{\partial w_j}$.

THE EXPLODING GRADIENT PROBLEM

The “Exploding Gradient Problem” occurs when

- Weights are too large (say on the order of 100 each)
- Biases b_j are such that $\sigma'(z_j)$ never is small
- *Example*: $b_j = -100 \times a_{j-1}$, so $z_j = 100 \times a_{j-1} - 100 \times a_{j-1} = 0$, implying $\sigma'(z_j) = 1/4$, yielding $w_j\sigma'(z_j) > 20$ as a gradient
- In such situations gradients iteratively explode

Gradients are Unstable

- The fundamental problem is that gradients in earlier layers are products of gradients from (all the) later layers.
- If there are many layers, the situation is unstable, unless the gradients are *balanced out*.
- Balancing is very unlikely to happen by chance, so one needs to fix this explicitly.
- Fixing this seems daunting at first glance: when making weights w_j large,

  $$\sigma'(z_j) = \sigma'(w_j a_{j-1} + b_j)$$

  will get small.

- *Solutions*:
  - *Rectified Linear Units* instead of sigmoid activation
  - *Batch Normalization* (discussed later in the lecture)

WHY IS DEEP LEARNING TOUGH?

LITERATURE

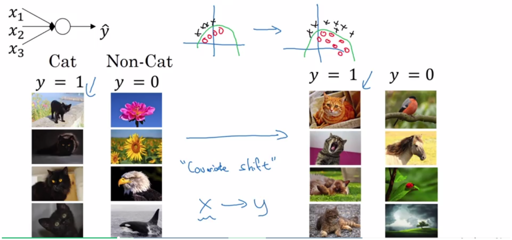
There are other issues that prevent easy training of neural networks with deep architectures. For further reading, see for example

- "Understanding the difficulty of training deep feedforward neural networks", X. Glorot, Y. Bengio, 2010, `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`

or the earlier

- "Efficient BackProp", Y. LeCun, L. Bottou, G. Orr, K.-R. Müller, 1998, `http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf`
- "On the importance of initialization and momentum in deep learning", I. Sutskever, J. Martens, G. Dahl, G. Hinton, 2013, `http://www.cs.toronto.edu/~hinton/absps/momentum.pdf`

Batch Normalization

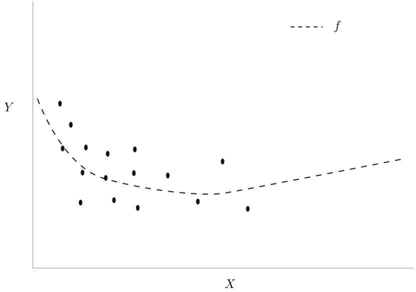


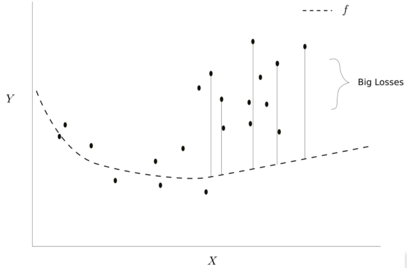
Learning black cats might not help to recognize cats of other colors

- The network might not be able to predict well if presented with examples not present in the training data (batch)
- The function learned can only be guaranteed to predict well in certain areas of feature space

Batch Normalization

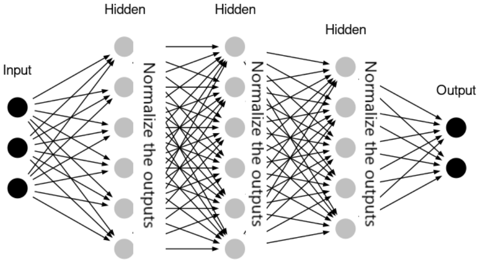
Motivation





Batch Normalization

Solution



Batch Normalization: Insert normalization layers between normal-type layers

- After each layer, normalize output values
- There are parameters to be learned for normalization layers
- Parameters for normalization layers can be easily learnt with backpropagation

Batch Normalization

Definition

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

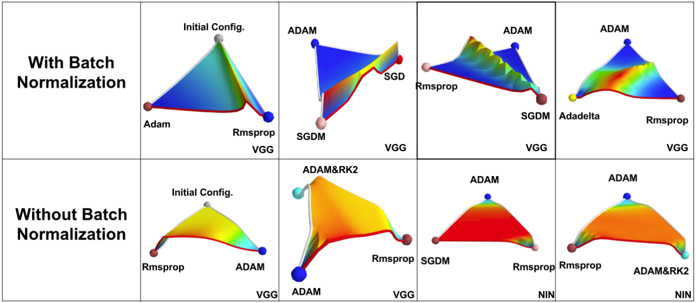
https://arxiv.org/abs/1502.03167

(Ioffe & Szegedy, original paper)

- Compute $\hat{x}_i$ when forwarding training samples
- Learn γ, β during backpropagation

Batch Normalization

Explanation



[From: http://www.aifounded.com/machine-learning/deep-loss]

- Low error regions are larger
- Boundaries are more clearly / sharply defined
- The reshaping of the cost function surface leads to accelerated training

Batch Normalization

Summary Benefits

- *Gradient Vanishing*: Batch Normalization prevents gradients from vanishing
- *Internal Covariate Shift*: controversial debate whether it helps (although it is motivated by it)
- Boundaries of error regions are more clearly / sharply defined
- Reshapes cost function surface: accelerated training

LECTURE5: SUMMARY I

- Recurrent neural networks
  - See http://www.deeplearningbook.org/, chapter 10
  - *Long Short Term Memory Networks:* see also https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- The vanishing gradient problem
  - http://neuralnetworksanddeeplearning.com/, Chapter 5
- Batch normalization
  - See http://www.deeplearningbook.org/, 8.7.1
  - See also http://www.aifounded.com/machine-learning/deep-loss, for example

OUTLOOK

- Deep neural networks
  - http://neuralnetworksanddeeplearning.com/, Chapter 6, "Recent progress in image recognition"
  - http://d2l.ai/chapter_convolutional-modern/index.html, Chapter 8

Thanks for your attention