TECHNISCHE UNIVERSITÄT DRESDEN

SEMANTIC COMPUTING

Lecture 10: Deep Learning: Optimization and Long-Short Term Memory

Dagmar Gromann

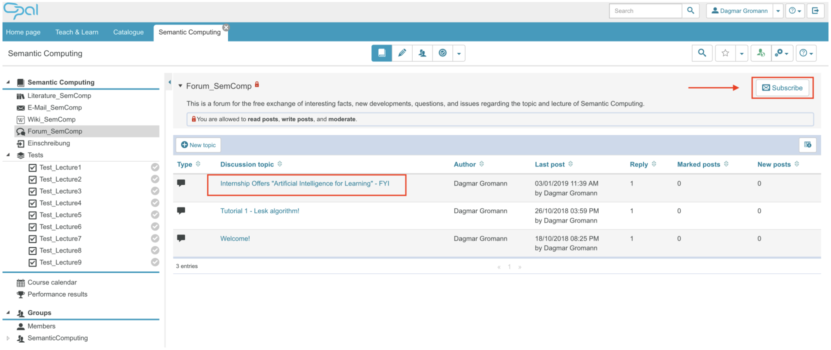
International Center For Computational Logic

TU Dresden, 11 December 2018

Overview

- Optimiziation
- Long-Short Term Memory (LSTM)

How to get updates from OPAL?



Forum_SemComp

Subscribe

This is a forum for the free exchange of interesting facts, new developments, questions, and issues regarding the topic and lecture of Semantic Computing.

You are allowed to **read posts**, **write posts**, and **moderate**.

Type	Discussion topic	Author	Last post	Reply	Marked posts	New posts
	Internship Offers "Artificial Intelligence for Learning" - FYI	Dagmar Gromann	03/01/2019 11:39 AM by Dagmar Gromann	1	0	0
	Tutorial 1 - Lesk algorithm!	Dagmar Gromann	26/10/2018 03:59 PM by Dagmar Gromann	1	0	0
	Welcome!	Dagmar Gromann	18/10/2018 08:25 PM by Dagmar Gromann	1	0	0

3 entries

Optimization

Reminder: What is optimization?

If we knew the distribution of our data, the training would be an optimization problem. However, we only have a sample of training data and do not know the full distribution of the data in machine learning. We try to approximate the distribution of our data (also the ”unseen” examples we use for testing the model’s ability to generalize).

So instead we try to: minimize the expected loss on the training set
First algorithm: (Mini-batch) Gradient Descent
Today: alternatives

TECHNISCHE UNIVERSITÄT DRESDEN

Reminder: Which datasets do we optimize on?

Usual daset split 80 | 10 | 10 into:

- **Training set**: data used as input to train model (80)
- **Validation set**: data used during training to check model's ability to generalize (10)
- **Test set**: data NEVER used during training, but when we finished training to check final model's ability to generalize, after a good training and validation performance has already been achieved (10)

TECHNISCHE UNIVERSITÄT DRESDEN

Optimization Beyond SGD

We have looked at Stochastic Gradient Descent (SGD) and today we will cover further means of optimization:

- Batch normalization
- Parameter initialization
- Hyperparameter settings

Batch Normalization

Definition

Batch normalization accelerates deep network training by stabilizing the data distribution which reduces the internal covariate shift between two layers and smooths the optimization landscape

Why useful for deep learning? **Faster** and **more stable** training.

Covariate Shift

Changes in the input distribution slow down or stop model's convergence because hidden units have to continuously adapt to changing inputs and cannot learn any pattern from the inputs.

Reference: Santurkar et al. (2018) "How Does Batch Normalization Help Optimization?", 32nd Conference on Neural Information Processing Systems (NIPS 2018).

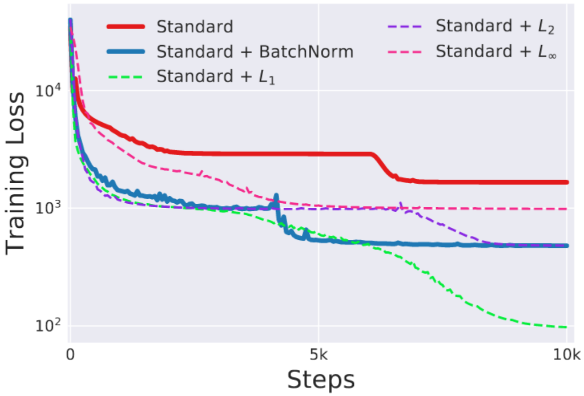
How does Batch Normalization work?

It is a mechanism that aims to stabilize the distribution of inputs by:

- adding an additional hidden layer (weights z) that normalizes the input:
  - set mean to 0: $\mu = \frac{1}{m} \sum_i z^{(i)}$
  - set variance to 1: $\sigma^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$
  - this gives us $z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$
- mean and variance should be controlled but not always constant, so we add **trainable** parameters (γ, β): $z'^{(i)} = \gamma \, z_{norm}^{(i)} + \beta$

TECHNISCHE UNIVERSITÄT DRESDEN

Visualization of Batch Normalization Effects



Source: Santurkar et al. (2018) "How Does Batch Normalization Help Optimization?", 32nd Conference on Neural Information Processing Systems (NIPS 2018).

Parameter (Weight) Initialization

Careful choice for the random initialization for the neural network can optimize. Can determine whether and how quickly the learning converges. Typical settings:

- different initialization for different layers when using the same activation function (“break symmetry” between different hidden units)
- bias for each unit is heuristically constant
- weights initialized with random weights drawn from a Gaussian or uniform distribution

Parameter (Weight) Initialization

One heuristic to initialize is where x is our input and y is the output:

$$U(-\frac{1}{\sqrt{x}}, \frac{1}{\sqrt{x}})$$

And an extended normalized initialization is:

$$W_{i,j} \sim U(-\sqrt{\frac{6}{x+x}}, \sqrt{\frac{6}{x+y}})$$

Sparse initialization: have exactly k nonzero weights (downside: maxout problems)

Hyperparameters: Adaptive Learning Rates

Problems of classical SGD are:

- too small learning rate = very slow convergence
- too large learning rate = constant overstepping of local minimum
- decaying learning rate by schedule or certain threshold if an objective between epochs is met, needs to be defined in advance (not dynamic)
- same learning rate applies to all parameter updates
- likely to get stuck on a saddle point

One solution: decay the learning rate by multiplying it by the inverse squared root gradient sum matrix or other optimization method with adaptive learning rate

Optimization Methods with Adaptive Learning Rates

- Momentum
- Adagrad
- Adadelta
- Adam

Animation of Differences

Momentum

Method to accelerate SGD in the relevant direction of the local minimum while reducing the strong oscillating by adding a fraction γ of the vector of the past time step to the current vector of the current time step:

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

Usual value of momentum term $\gamma = 0.9$ or similar; ϵ is our learning rate and θ are the model parameters; idea: ball rolling down a hill

AdaGrad

Decays learning rate by inverse squared roots; performs per-parameter updates idea:

- frequently occurring features: small updates (low learning rates)
- infrequent features: large updates (high learning rates)
- particularly well suited for sparse data
- update for each parameter θ_i at teach time step t: $\theta_{t+1,i} = \theta_{t,i} - \epsilon \nabla J(\theta_{t,i})$
- decays learning rate based on the sum of the squares of past gradients, which are stored along the diagonal of matrix G_t and a small constant δ to avoid zeros: $\theta_{t+1} = \theta_t - \frac{\epsilon}{\sqrt{G_t + \delta}} \nabla J(\theta_t)$

Adadelta: extension of AdaGrad that restricts the window to accumulate past gradients to some fixed size w.

Adam

Adaptive Moment Estimation (Adam) keeps an exponentially decaying average of past gradients m_t and past squared gradients v_t - estimates of the first moment (mean) and the second moment (uncentered variance) of gradients:

$$m_t = \beta_1 \ m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 \ v_{t-1} + (1 - \beta_2) g_t^2$$

Update rule: $\theta_{t+1} = \theta_t - \frac{\epsilon}{\sqrt{v_t} + \delta} \ m_t$ where m_t is usually normalized to $\frac{m_t}{1 - \beta_1^t}$ as is v_t with β_2
Idea: ball with heave friction, i.e., prefers flat minima in the error surface

Long-Short Term Memory (LSTM)

TECHNISCHE UNIVERSITÄT DRESDEN

Long-Short Term Memory (LSTM)

RNN problem: In RNNs, gradients propagated over many stages tend to vanish or explode - the difficulty of long-term dependencies is generated by the exponentially smaller weights given to long-term interactions compared to short-term ones.

LSTM solution: LSTMs are explicitly designed to avoid the long-term dependency problem. The hidden layer is replaced by four interacting layers that regulate the information flow more tightly.

Seminal paper: Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

LSTM

The repeating, recurrent module in an LSTM consists of four interacting layers. Information is removed or added to the cell state (C) based on carefully regulated gates.

Forget gate: which information to let through and which to ignore;

Input gate: which information will be stored in the cell state.

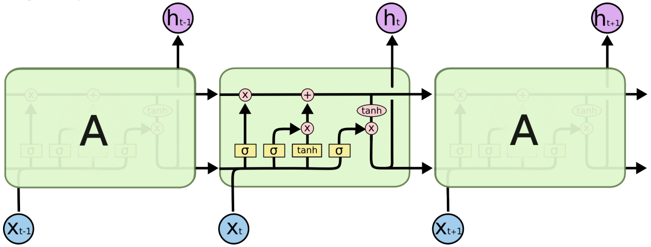
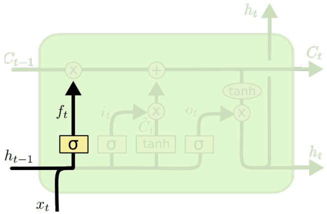


Image Source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Forget Gate

Controls which information to keep from the cell state; sigmoid layer (zero = let nothing through, one = let everything through) and a pointwise multiplication

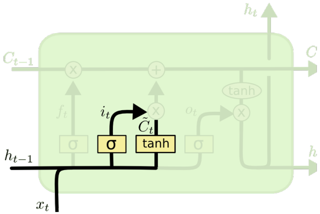


$$f_t = \sigma(W_f x_t + U_f h_{(t-1)} + b_f)$$

Image Source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Input Gate

Controls which information to store in the cell state; sigmoid layer decides what to update, the tanh layer creates a vector of new candidate values (values between -1 and 1).

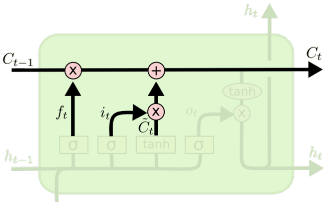


$$i_t = \sigma(W_i x_t + U_i h_{(t-1)} + b_i)$$
$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{(t-1)} + b_c)$$

Image Source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

From Old to New Cell State

Update the old cell stat C_{t-1} into the new cell state C_t. We multiply the old state with f_t to forget the things that were decided to be forgotten earlier. Then $i_t * \tilde{C}_t$ (where $*$ denotes the Hadamard product) are added.

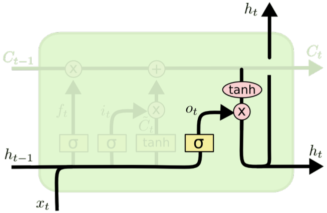


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Image Source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Output Gate

The output is a filtered version of the cell state. The sigmoid function decides which parts of the cell state we need to keep and the cell state is then put through a tanh layer (values squished between -1 and 1) and multiply it with the output of the sigmoid gate.



$$o_t = \sigma(W_o x_t + U_o h_{(t-1)} + b_o)$$
$$h_t = o_t * tanh(C_t)$$

Image Source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

LSTM variant: Gated Recurrent Unit (GRU)

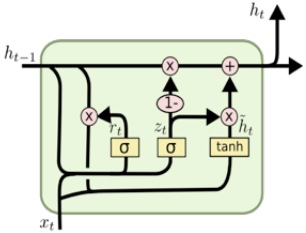
Variant of LSTM unit that combines the forget and the input gates into a single update gate and merges the cell state and the hidden state.

- update gate: determines how much of the past information from previous steps needs to be passed along
- forget gate: determines how much of the past information to forget
- those two gates are two vectors that decide which information should be passed to the output of the unit
- they can store information from “a long time ago”.

Seminal paper: Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

A Single GRU

Single Gated Recurrent Unit (GRU) where z_t is the update gate, r_t is the forget gate, $\tilde{h}_t$ is the current memory content, and h_t is the final memory at current time step => update gate decides what to keep from previous and current time steps



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

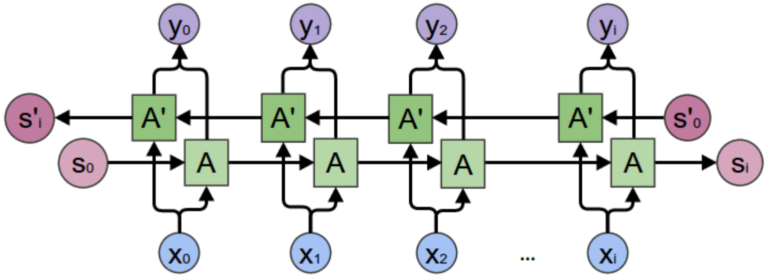
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Image Source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Bidirectional RNNs

This could be RNNs, LSTMs, or GRUs. An input sequence is red from left to right (A) and from right to left (A').



Output: $\hat{y}^t = g(W_y[A, A'] + b_y)$

Image source:
`https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66`

Deep RNNs

This could be RNNs, LSTMs, or GRUs. Several hidden layers are stacked on top of each other (here: three hidden layers).

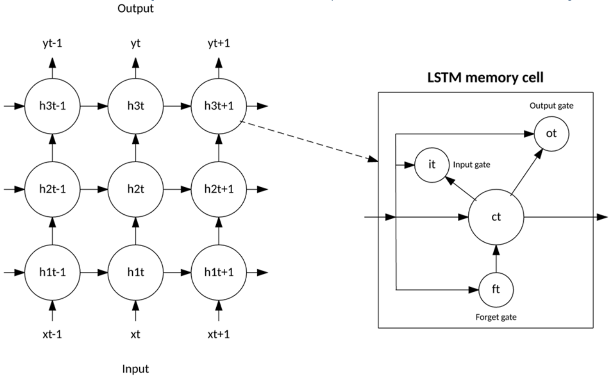


Image source: `https://www.ibm.com/developerworks/library/cc-machine-learning-deep-learning-architectures/index.html`

Review of Lecture 10

- How can RNNs be optimized?
- What is an adaptive learning rate and how can it be implemented?
- What is the difference between an RNN and an LSTM?
- In what way do GRUs differ from LSTMs?
- What does it mean for an RNN to be bidirectional? improved?