

Lecture 07:

DL recap: optimization, regularization, vanishing gradient problem

MADE, Moscow

Radoslav Neychev

03.03.2021

1. Recap: backpropagation, activations, intuition.
2. Optimizers.
3. Data normalization.
4. Regularization.
5. Vanishing gradient in RNNs
6. Vanishing gradient in deep neural networks
7. Q & A.

Recap: Deep Learning basics

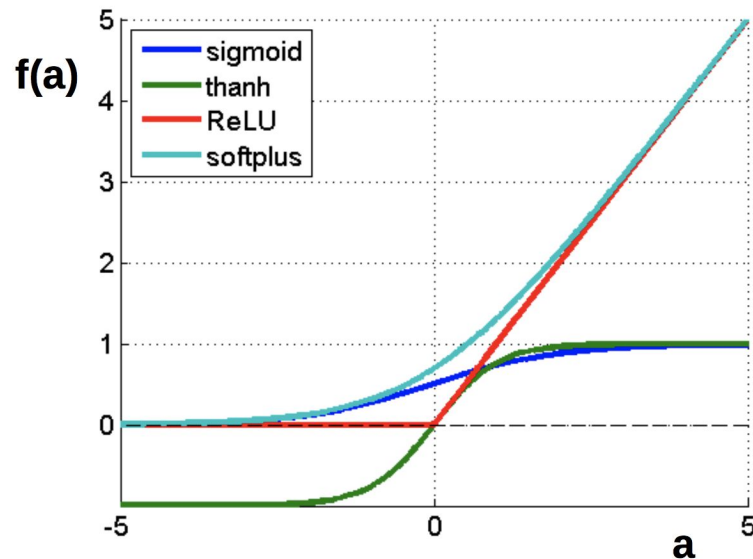
Once more: nonlinearities

$$f(a) = \frac{1}{1 + e^a}$$

$$f(a) = \tanh(a)$$

$$f(a) = \max(0, a)$$

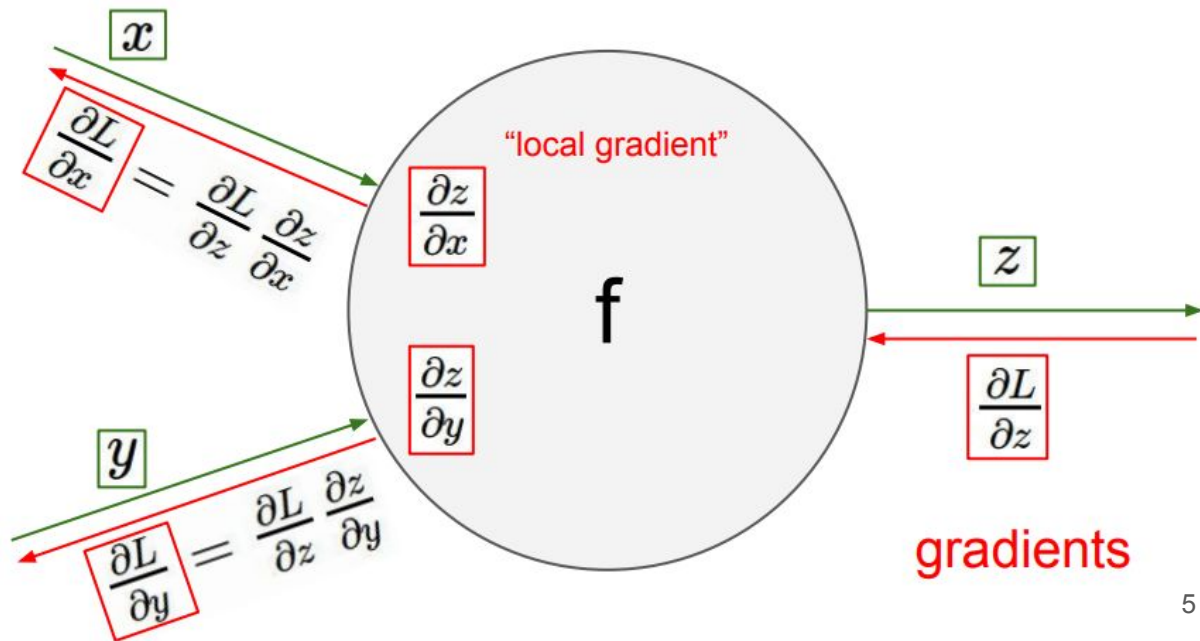
$$f(a) = \log(1 + e^a)$$



Backpropagation and chain rule

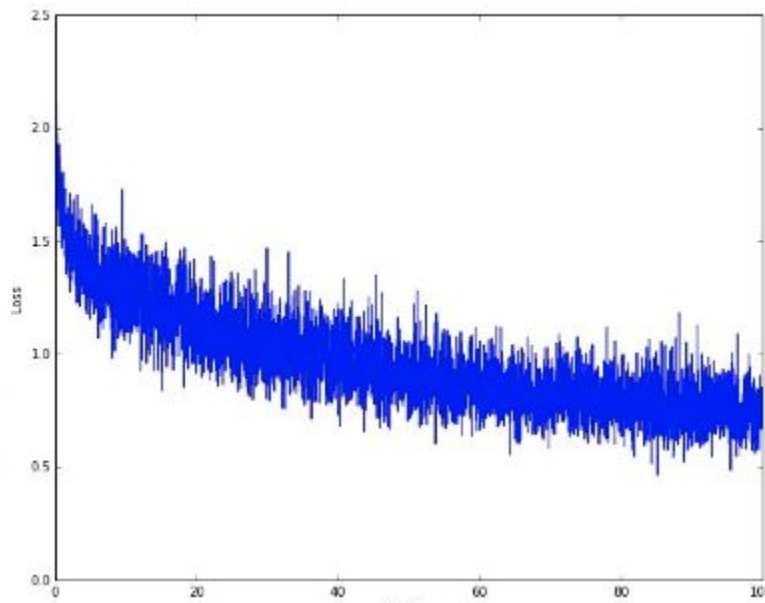
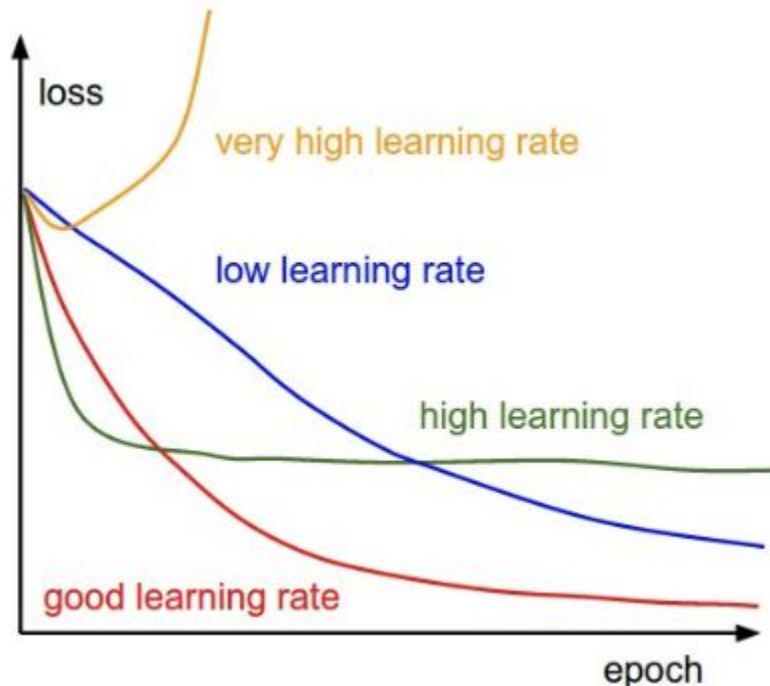
Chain rule is just simple math: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$

Backprop is just way to use it in NN training.



Stochastic gradient descent is used to optimize NN parameters.

$$x_{t+1} = x_t - \text{learning rate} \cdot dx$$

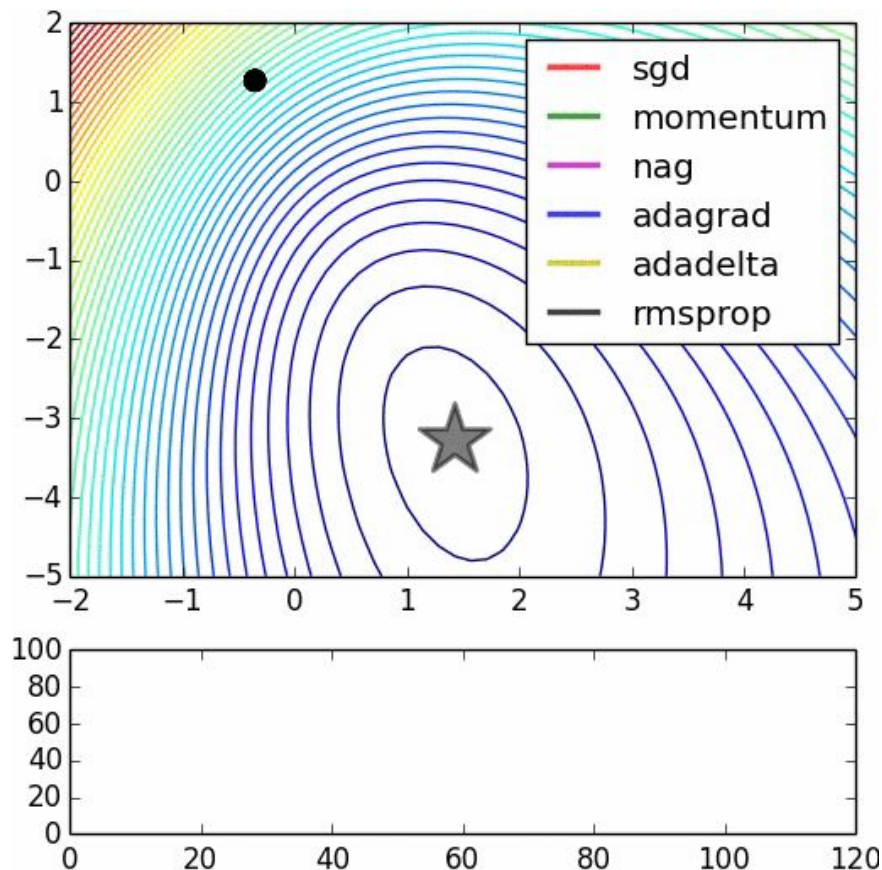


Optimization: SGD upgrades

Optimizers

There are much more optimizers:

- Momentum
- Adagrad
- Adadelat
- RMSprop
- Adam
- ...
- even other NNs

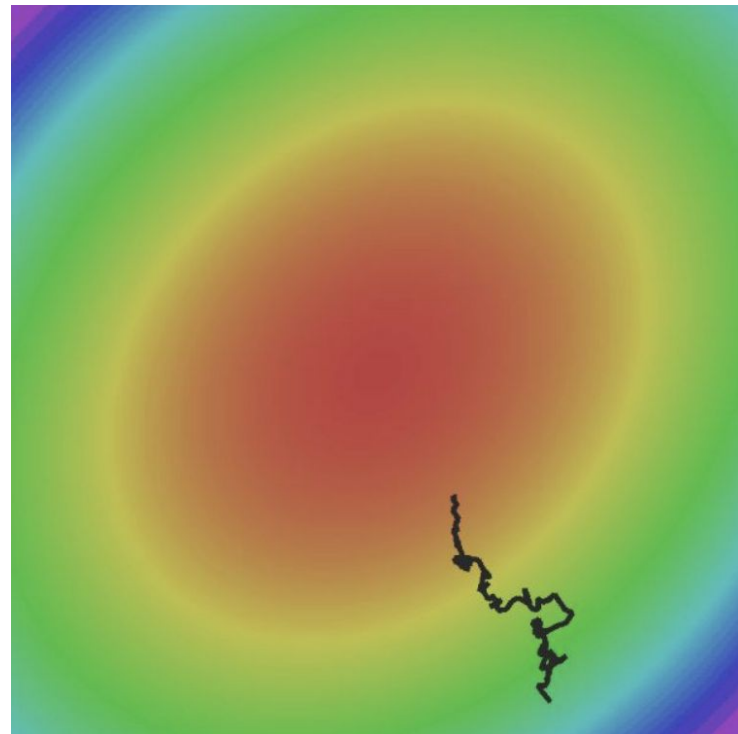


Optimization: SGD

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W)$$

Averaging over too small batches
leads to noisy gradient



First idea: momentum

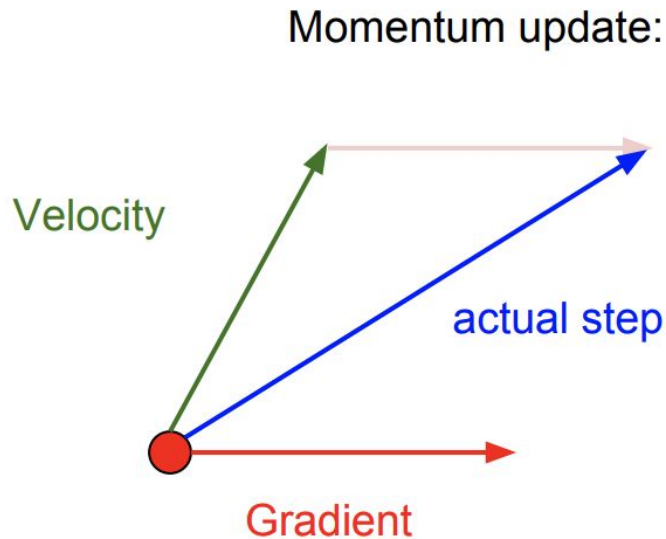
Simple SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

SGD with momentum

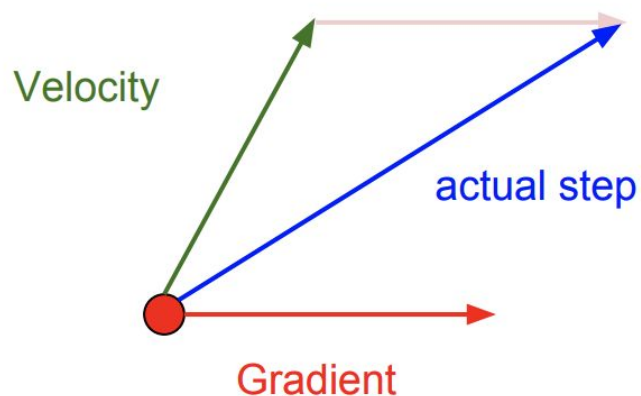
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$



Nesterov momentum

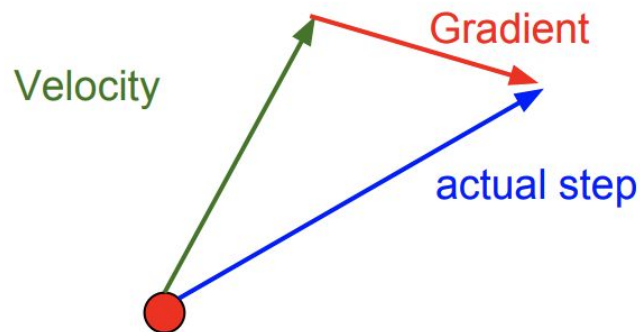
Momentum update:



$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

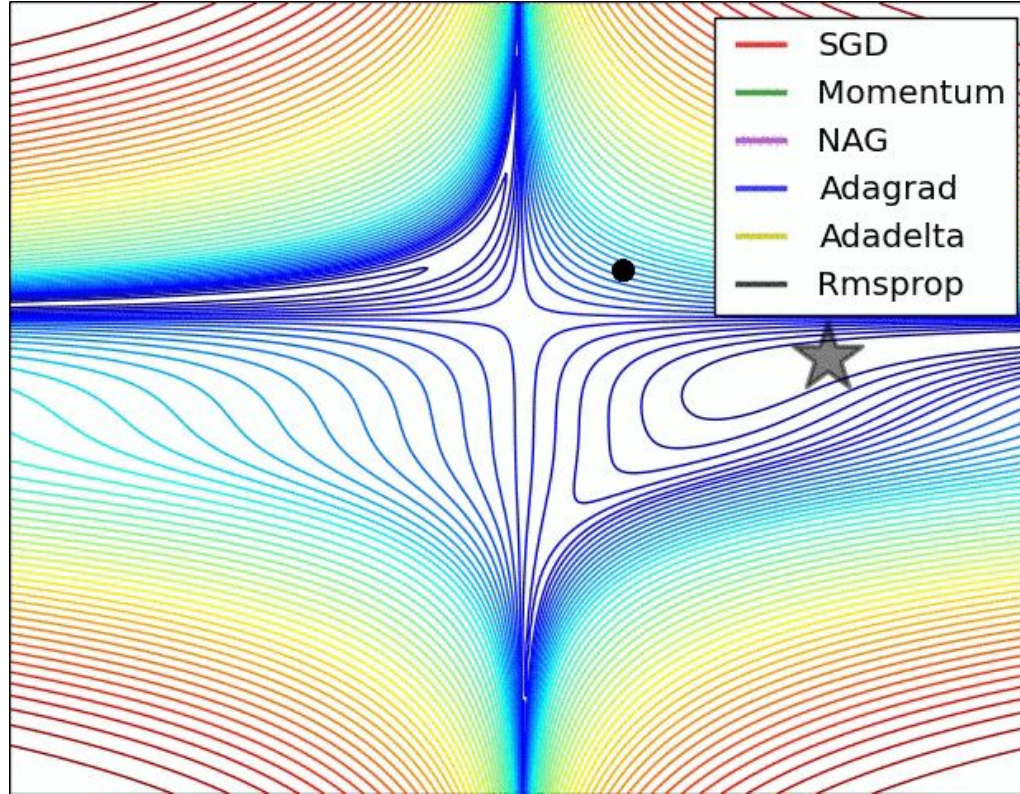
Nesterov Momentum

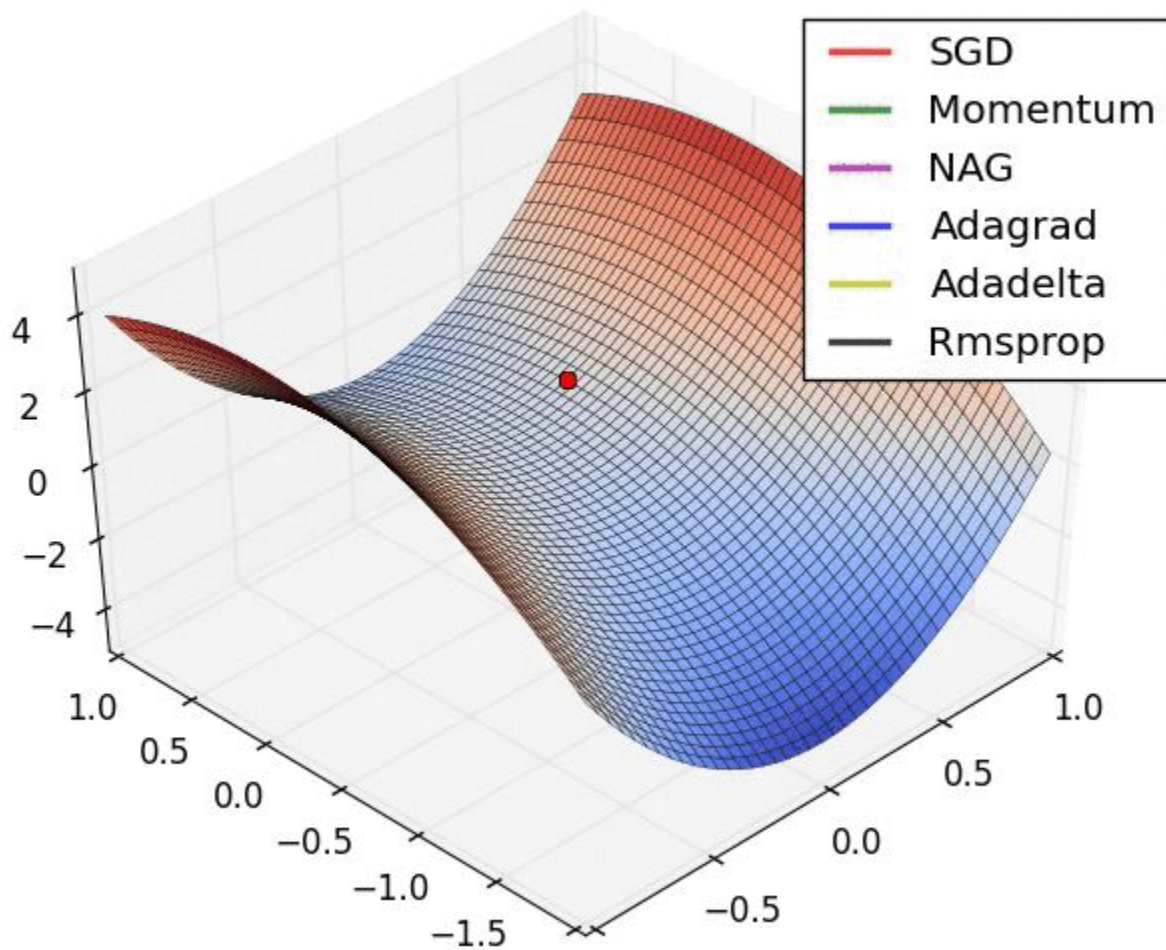


$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

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

Comparing momentums





Second idea: different dimensions are different

Adagrad: SGD with cache

$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

Second idea: different dimensions are different

Adagrad: SGD with cache

$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

Problem: gradient fades with time

Second idea: different dimensions are different

Adagrad: SGD with cache

$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

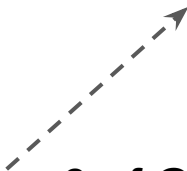
$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$



RMSProp: SGD with cache with exp. Smoothing

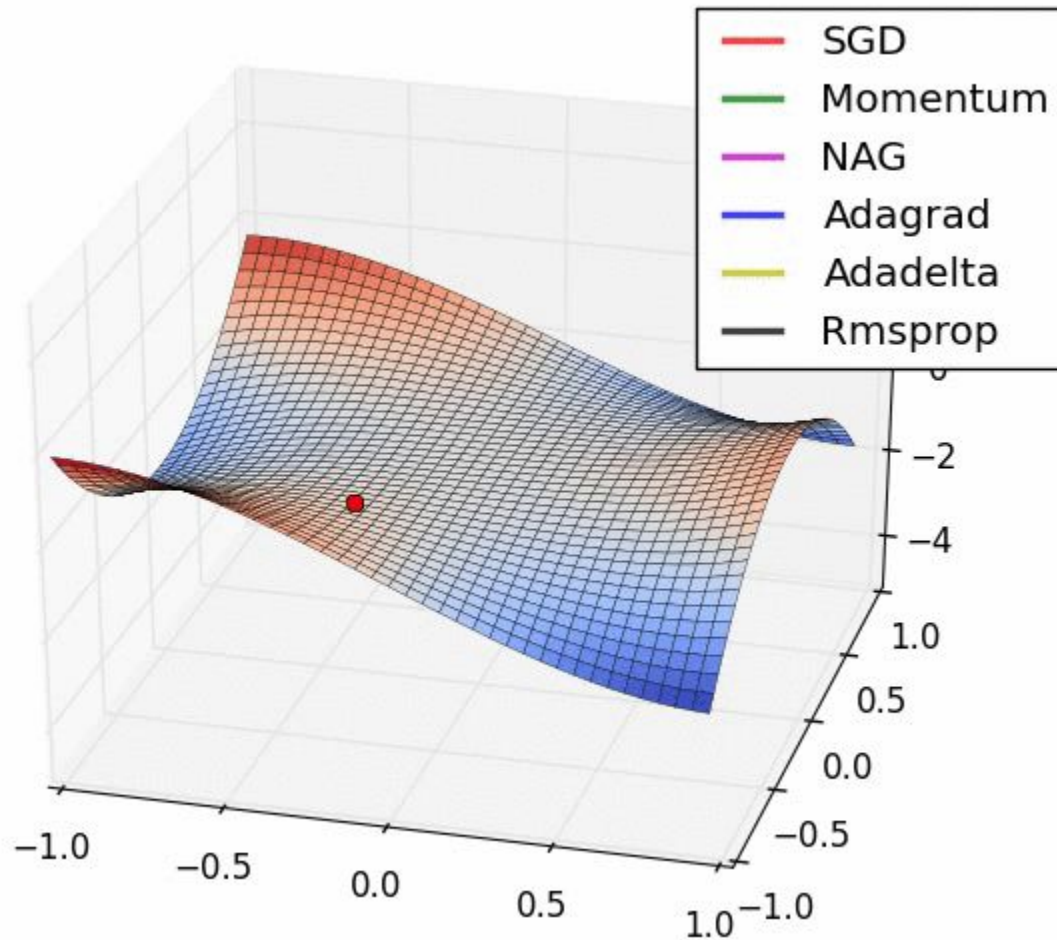
$$\text{cache}_{t+1} = \beta \text{cache}_t + (1 - \beta)(\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$



Slide 29 Lecture 6 of Geoff Hinton's Coursera class

http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

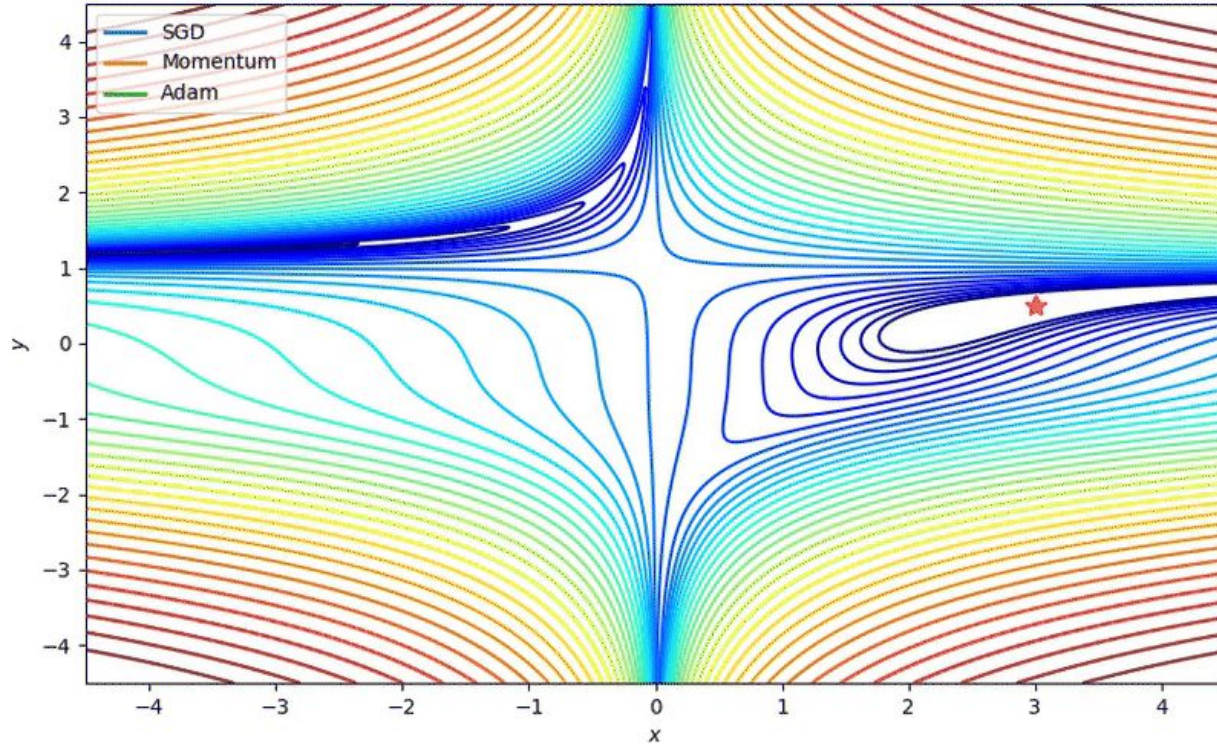


Let's combine the momentum idea and RMSProp normalization:

$$\begin{aligned}v_{t+1} &= \gamma v_t + (1 - \gamma) \nabla f(x_t) \\ \text{cache}_{t+1} &= \beta \text{cache}_t + (1 - \beta) (\nabla f(x_t))^2 \\ x_{t+1} &= x_t - \alpha \frac{v_{t+1}}{\text{cache}_{t+1} + \varepsilon}\end{aligned}$$

Actually, that's not quite Adam.

Comparing optimizers





Andrej Karpathy ✓

@karpathy



3e-4 is the best learning rate for Adam, hands down.

6:01 AM · Nov 24, 2016 · [Twitter Web Client](#)

108 Retweets **461** Likes



Andrej Karpathy ✓ @karpathy · Nov 24, 2016



Replying to [@karpathy](#)

(i just wanted to make sure that people understand that this is a joke...)



9



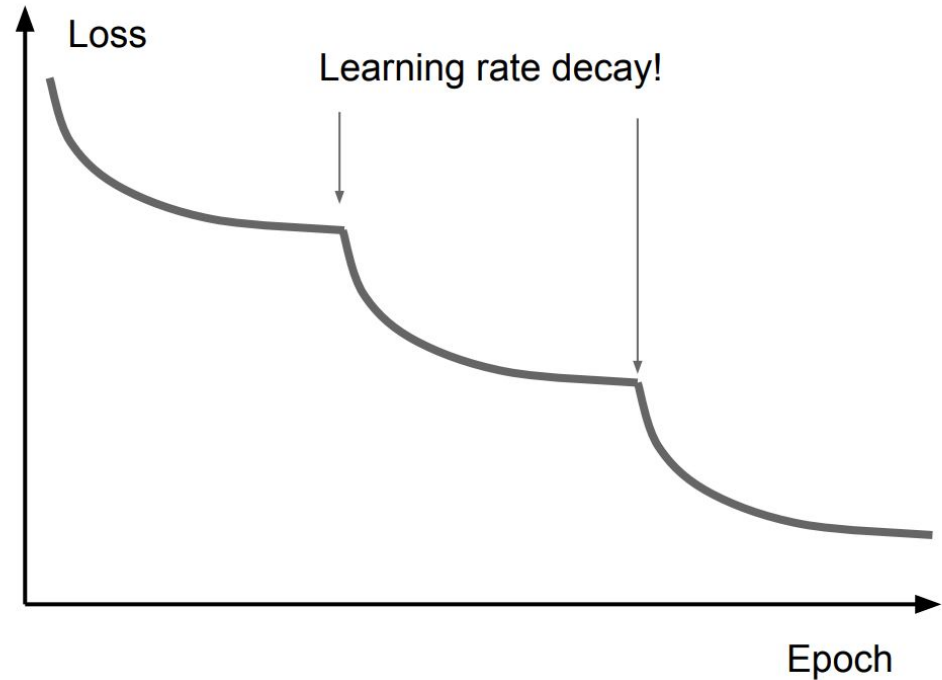
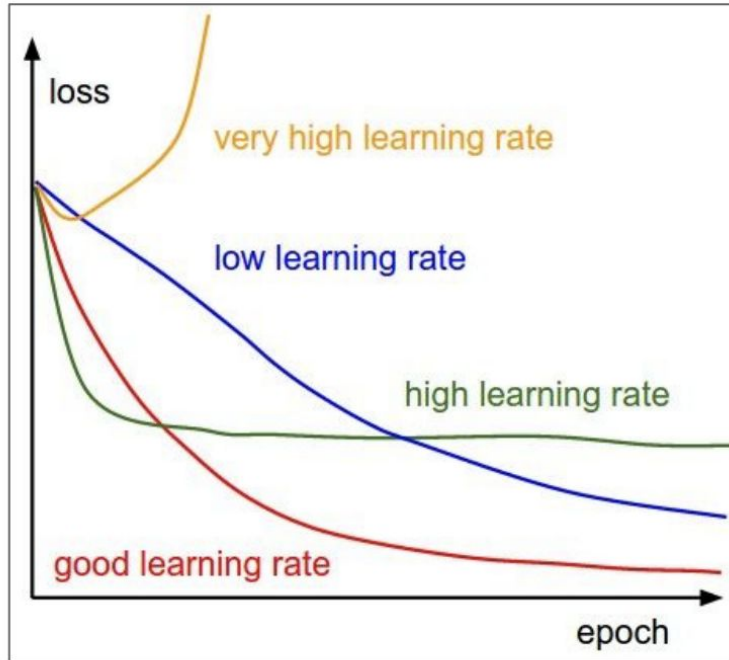
3



119



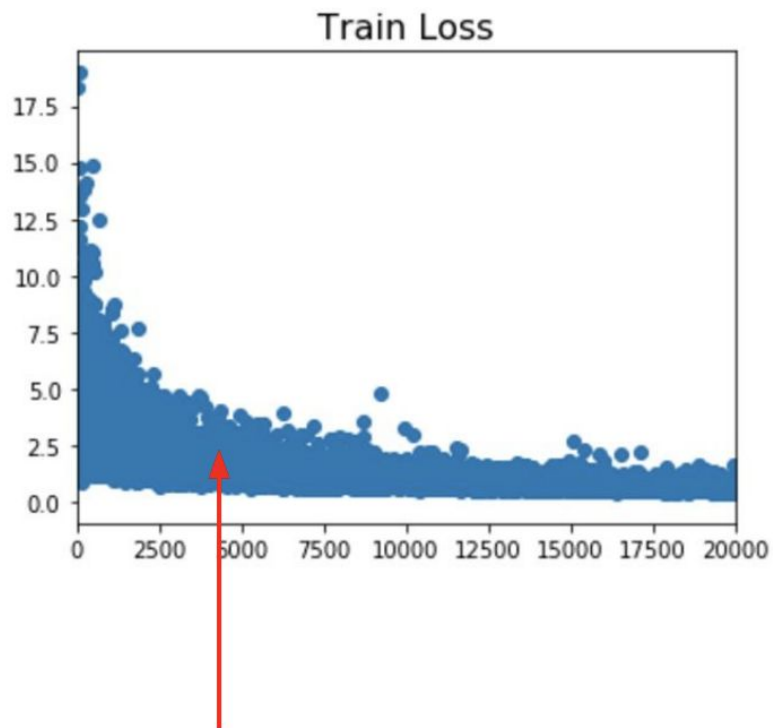
Once more: learning rate



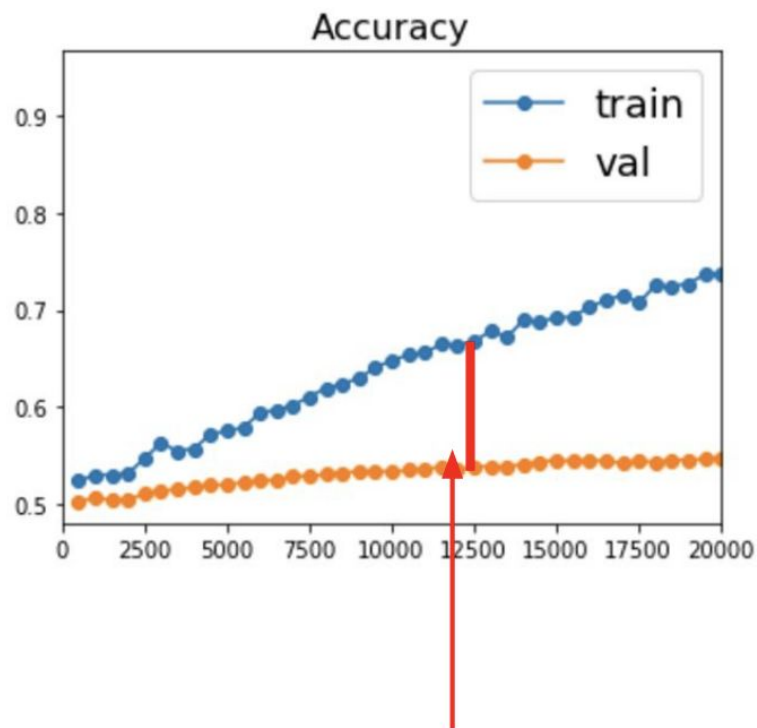
Sum up: optimization

- Adam is great basic choice
- Even for Adam/RMSProp learning rate matters
- Use learning rate decay
- Monitor your model quality

Regularization in DL

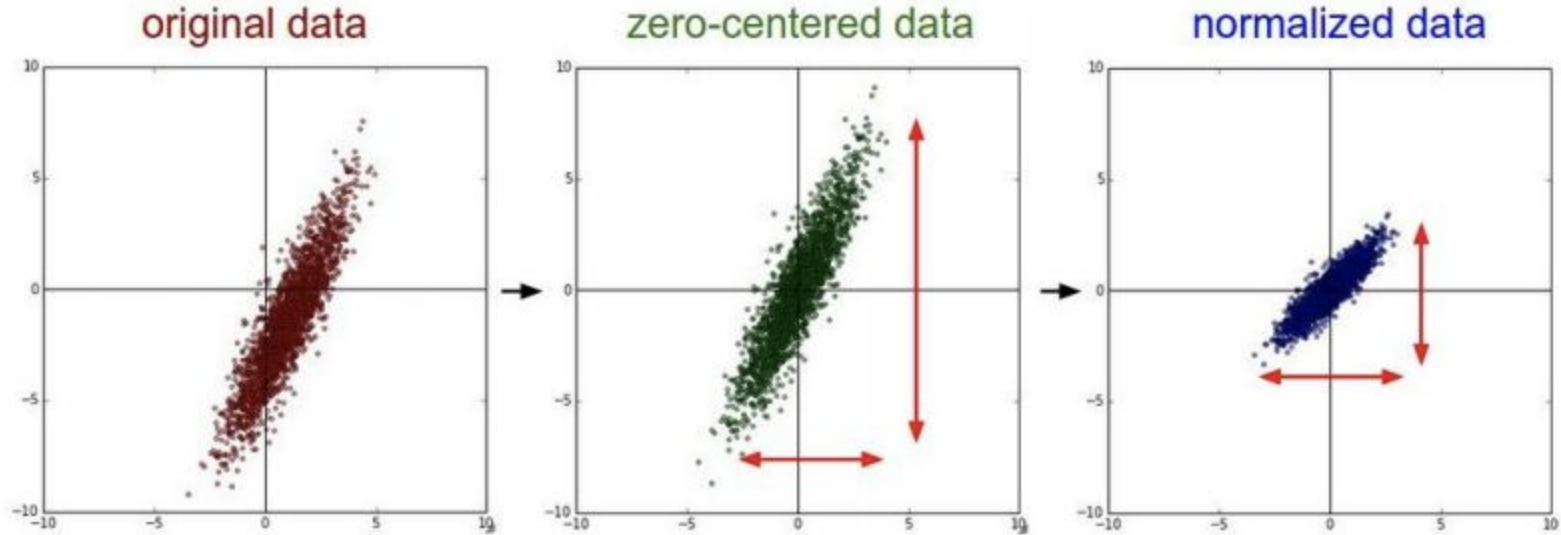


Better optimization algorithms
help reduce training loss



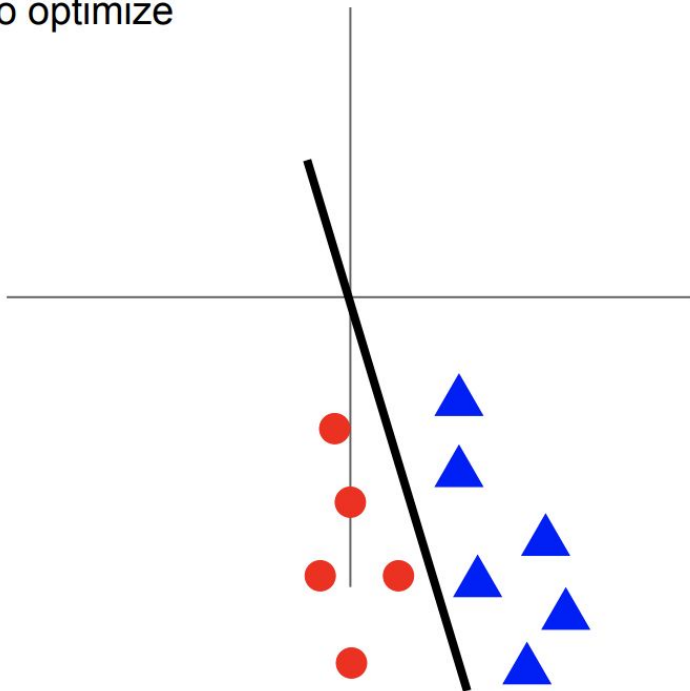
But we really care about error on new
data - how to reduce the gap?

Data normalization

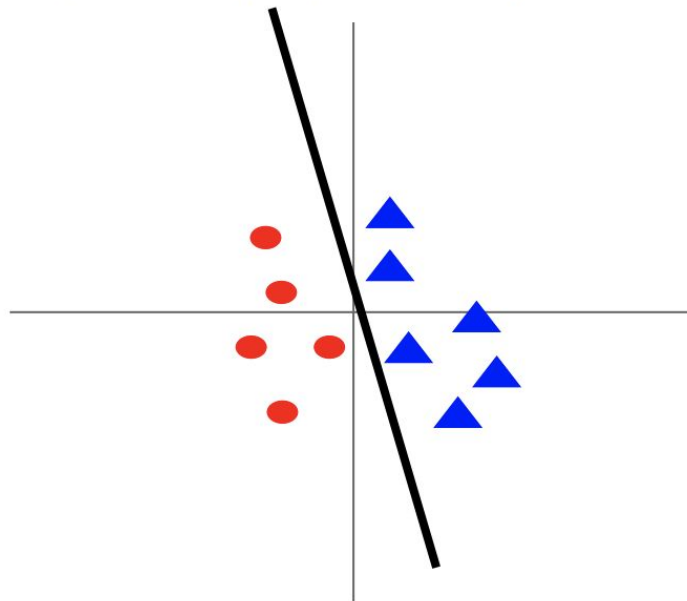


Data normalization

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



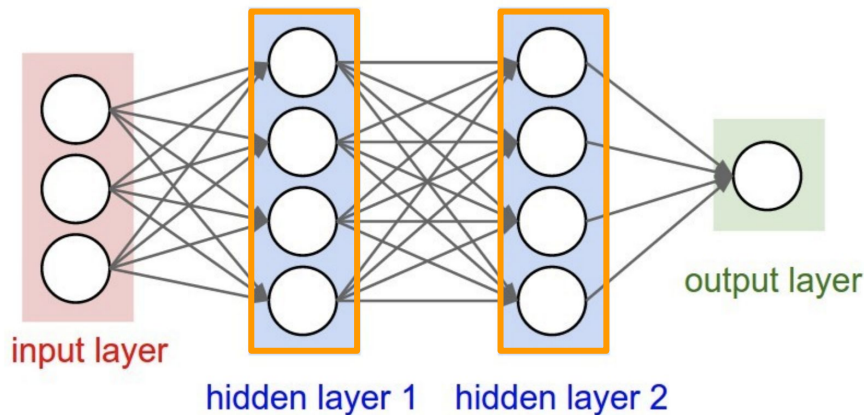
After normalization: less sensitive to small changes in weights; easier to optimize



Problem:

Batch normalization

- Consider a neuron in any layer beyond first
- At each iteration its weights are tuned to reduce loss
- Its inputs are tuned as well. Some of them become larger, some – smaller
- Now the neuron needs to be re-tuned for it's new inputs



TL; DR:

- *It's usually a good idea to normalize linear model inputs*
- (c) Every machine learning lecturer, ever

Batch normalization

- Normalize activation of a hidden layer
(zero mean unit variance)

$$h_i = \frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}$$

- Update μ_i, σ_i^2 with moving average while training

$$\mu_i := \alpha \cdot \text{mean}_{\text{batch}} + (1 - \alpha) \cdot \mu_i$$

$$\sigma_i^2 := \alpha \cdot \text{variance}_{\text{batch}} + (1 - \alpha) \cdot \sigma_i^2$$

Batch normalization

Original
algorithm (2015)

What is this?

This
transformation
should be able to
represent the
identity transform.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;
Parameters to be learned: γ, β

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

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

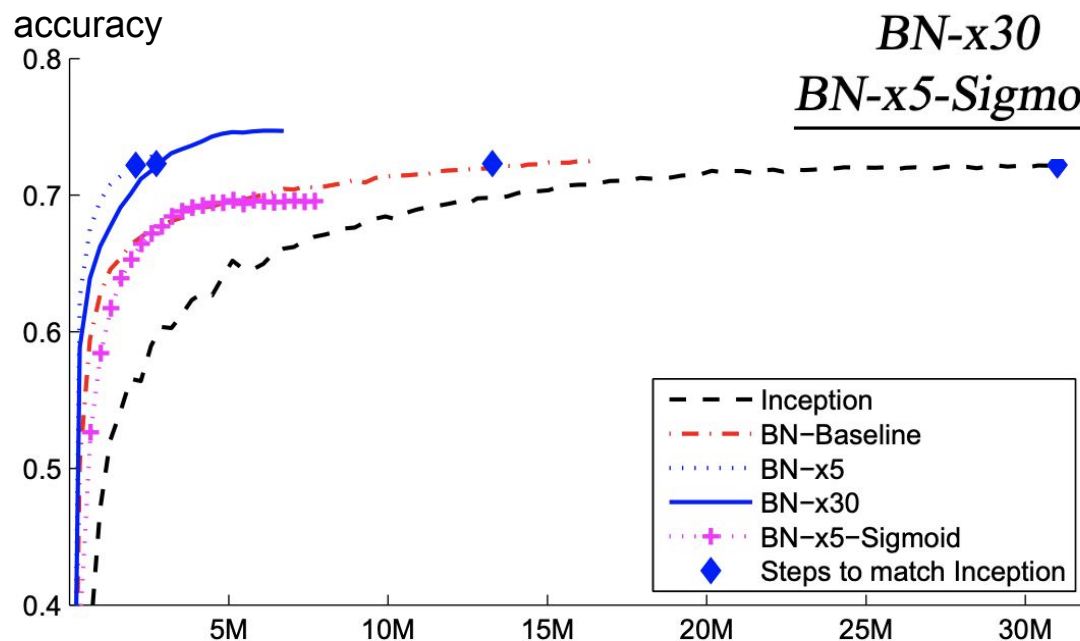
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

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

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

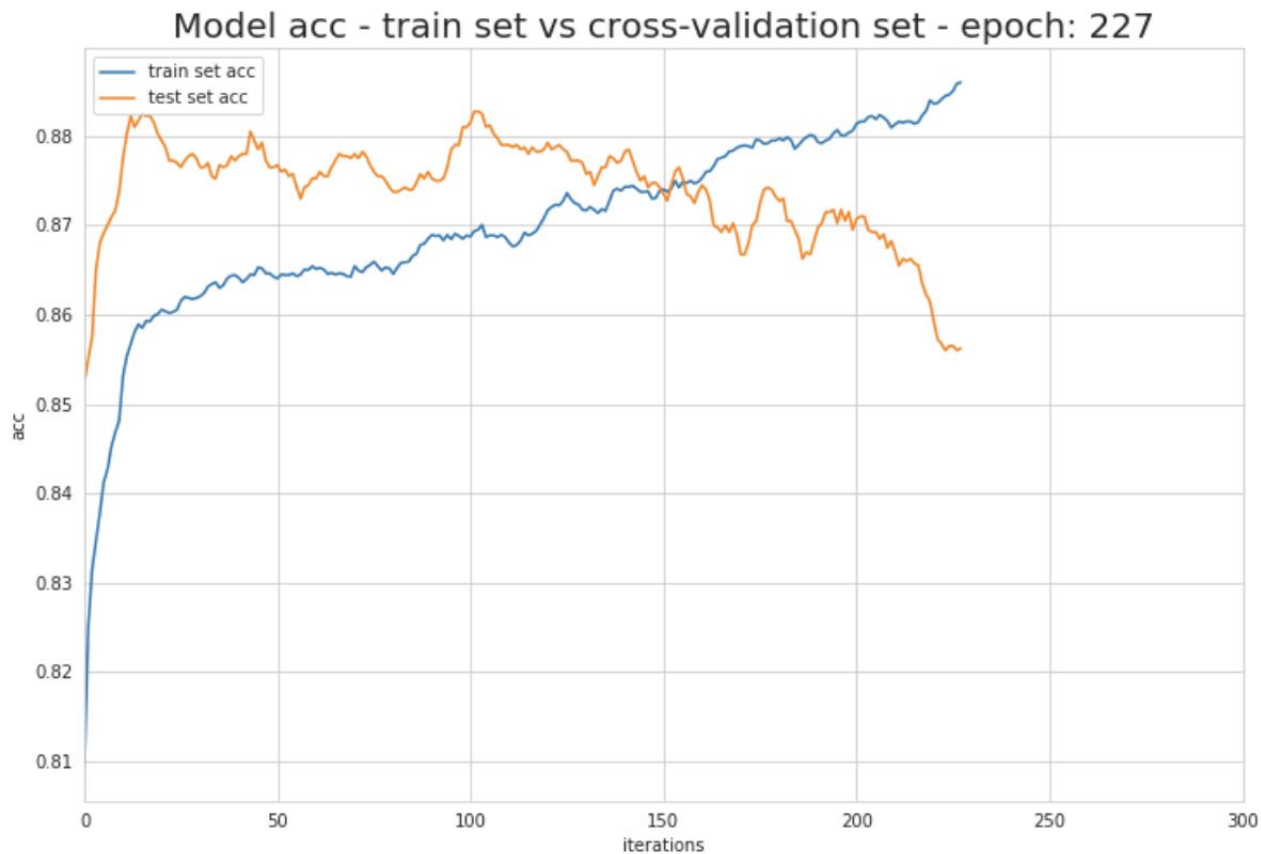
Batch normalization

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
<i>BN-Baseline</i>	$13.3 \cdot 10^6$	72.7%
<i>BN-x5</i>	$2.1 \cdot 10^6$	73.0%
<i>BN-x30</i>	$2.7 \cdot 10^6$	74.8%
<i>BN-x5-Sigmoid</i>		69.8%



number of training steps

Problem: overfitting



$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

Adding some extra term to the loss function.

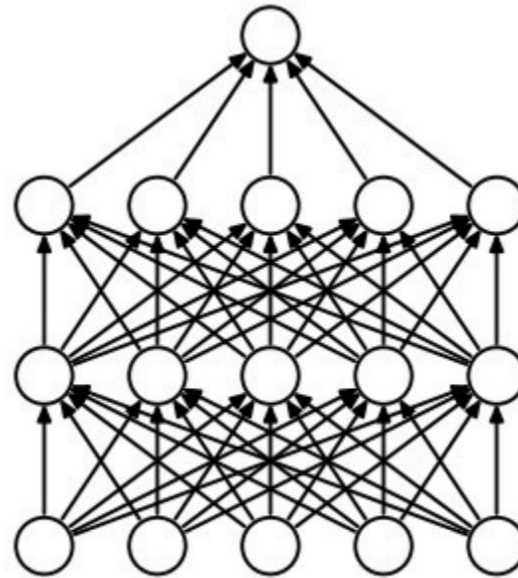
Common cases:

- L2 regularization: $R(W) = \|W\|_2^2$
- L1 regularization: $R(W) = \|W\|_1$
- Elastic Net (L1 + L2): $R(W) = \beta \|W\|_2^2 + \|W\|_1$

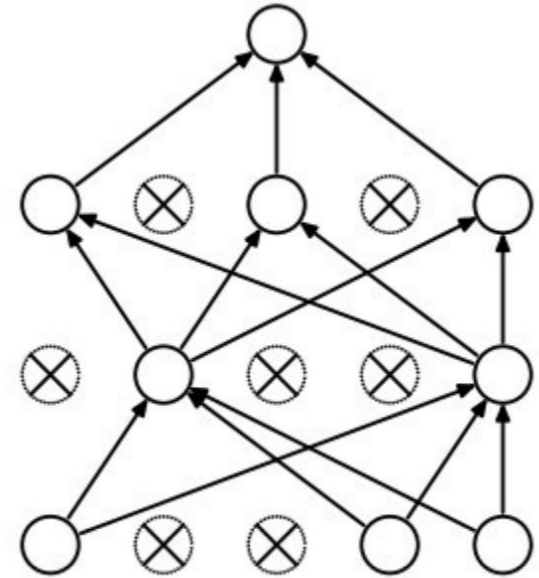
Regularization: Dropout

Some neurons are
“dropped” during
training.

Prevents overfitting.



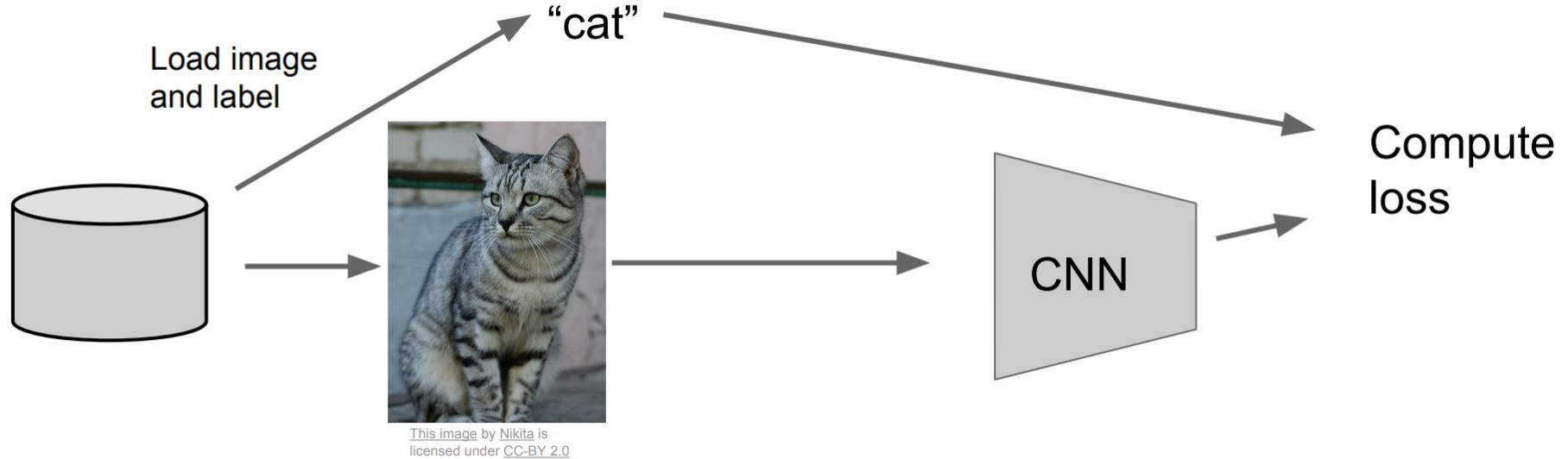
(a) Standard Neural Net



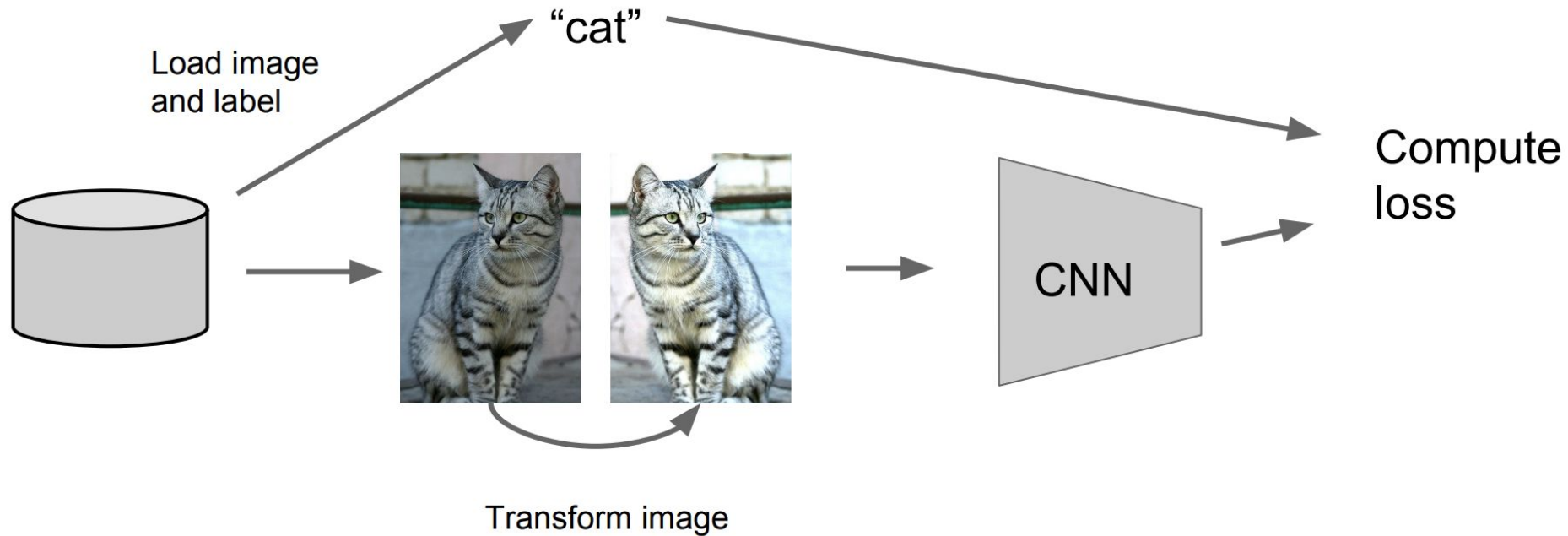
(b) After applying dropout.

Actually, on test case output should be
normalized. See sources for more info.

Regularization: data augmentation



Regularization: data augmentation



Optimization:

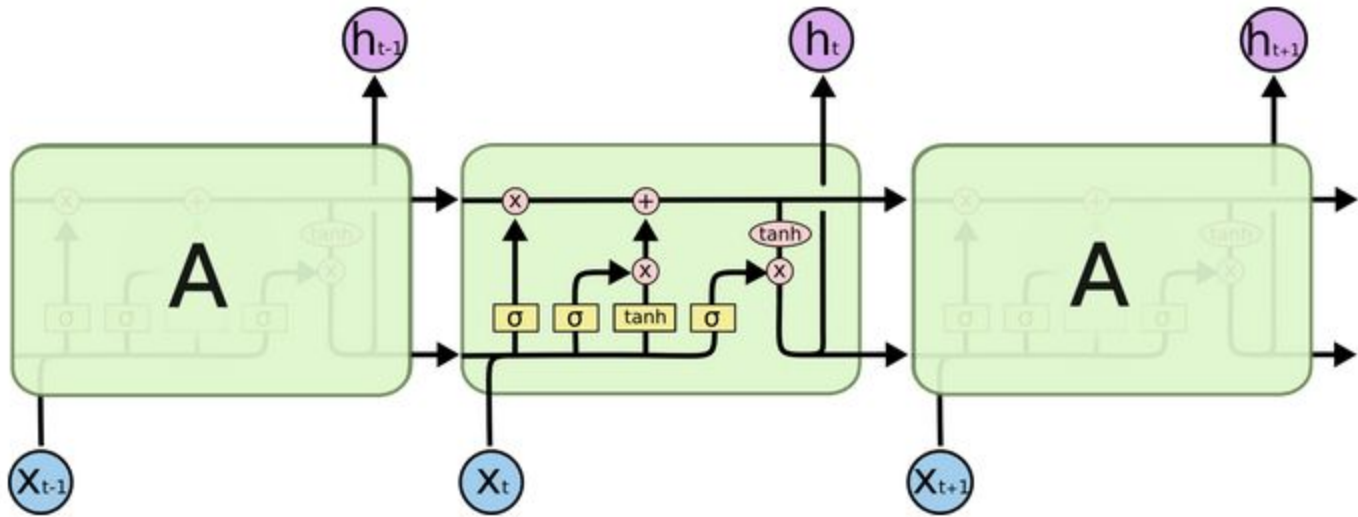
- Adam is great basic choice
- Even for Adam/RMSProp learning rate matters
- Use learning rate decay
- Monitor your model quality

Regularization:

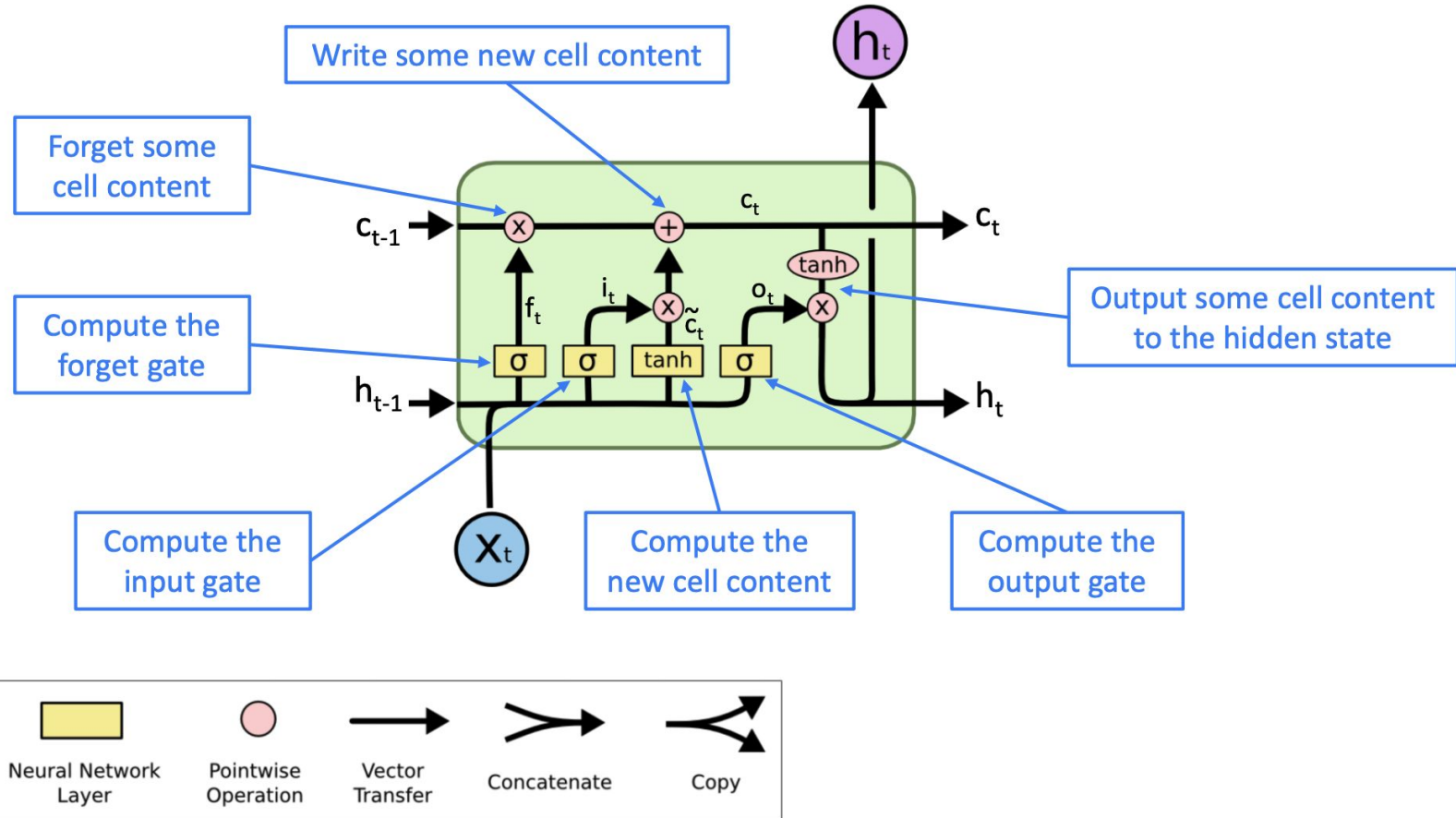
- Add some weight constraints
- Add some random noise during train and marginalize it during test
- Add some prior information in appropriate form

Further readings available [here](#)

LSTM



LSTM: quick overview



LSTM: quick overview

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$\begin{aligned}f^{(t)} &= \sigma \left(W_f h^{(t-1)} + U_f x^{(t)} + b_f \right) \\i^{(t)} &= \sigma \left(W_i h^{(t-1)} + U_i x^{(t)} + b_i \right) \\o^{(t)} &= \sigma \left(W_o h^{(t-1)} + U_o x^{(t)} + b_o \right)\end{aligned}$$

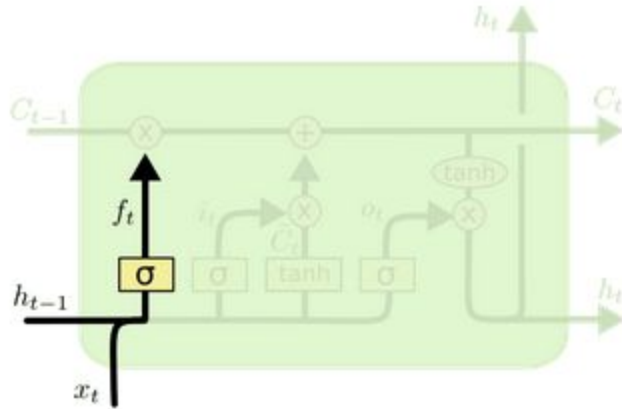
$$\begin{aligned}\tilde{c}^{(t)} &= \tanh \left(W_c h^{(t-1)} + U_c x^{(t)} + b_c \right) \\c^{(t)} &= f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}\end{aligned}$$

$$h^{(t)} = o^{(t)} \circ \tanh c^{(t)}$$

All these are vectors of same length n

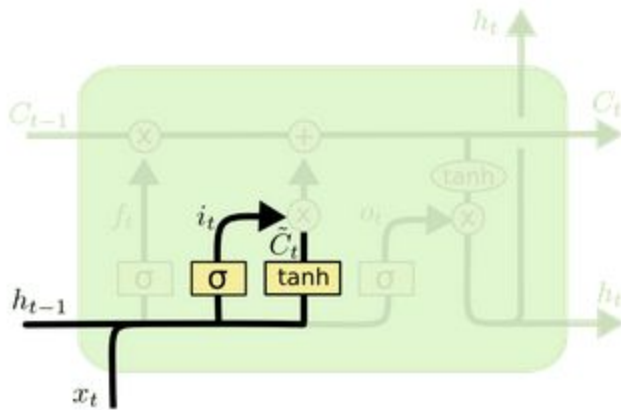
Gates are applied using element-wise product

LSTM: quick overview



$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

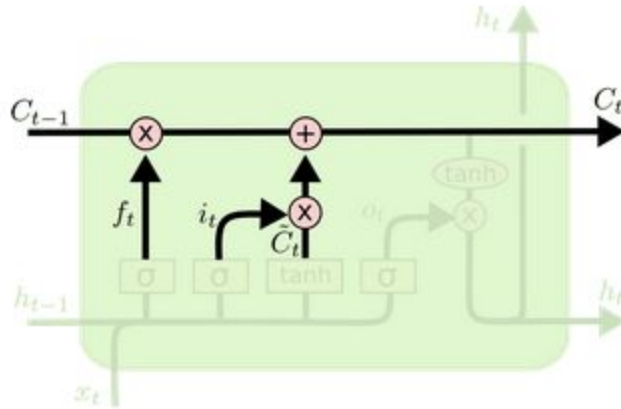
LSTM: quick overview



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

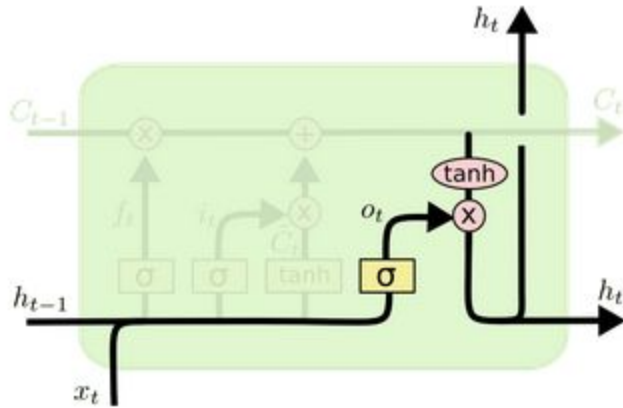
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

LSTM: quick overview



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

LSTM: quick overview



$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

LSTM: with formulas

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$\begin{aligned}f^{(t)} &= \sigma \left(W_f h^{(t-1)} + U_f x^{(t)} + b_f \right) \\i^{(t)} &= \sigma \left(W_i h^{(t-1)} + U_i x^{(t)} + b_i \right) \\o^{(t)} &= \sigma \left(W_o h^{(t-1)} + U_o x^{(t)} + b_o \right)\end{aligned}$$

$$\tilde{c}^{(t)} = \tanh \left(W_c h^{(t-1)} + U_c x^{(t)} + b_c \right)$$

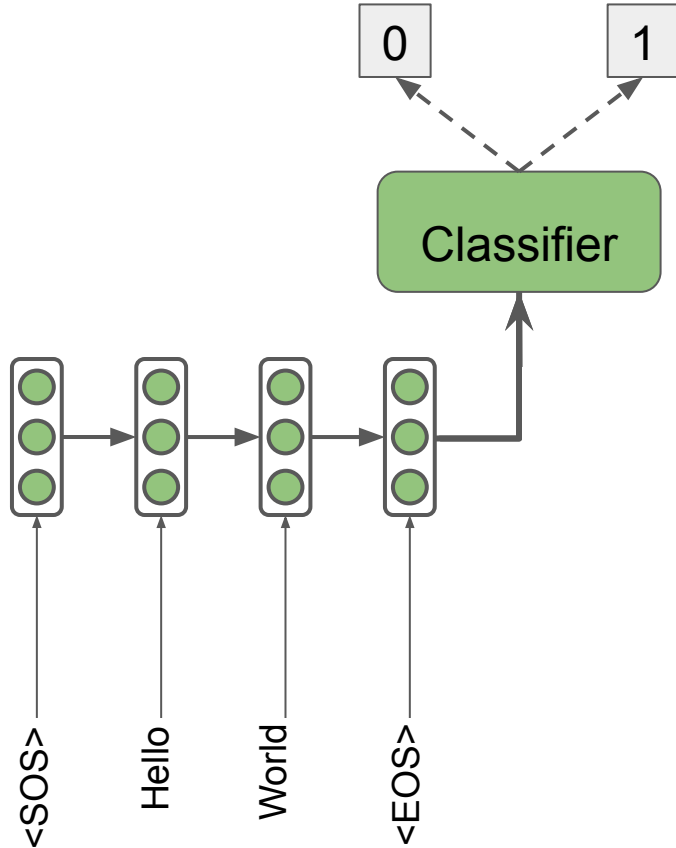
$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}$$

$$h^{(t)} = o^{(t)} \circ \tanh c^{(t)}$$

All these are vectors of same length n

Gates are applied using element-wise product

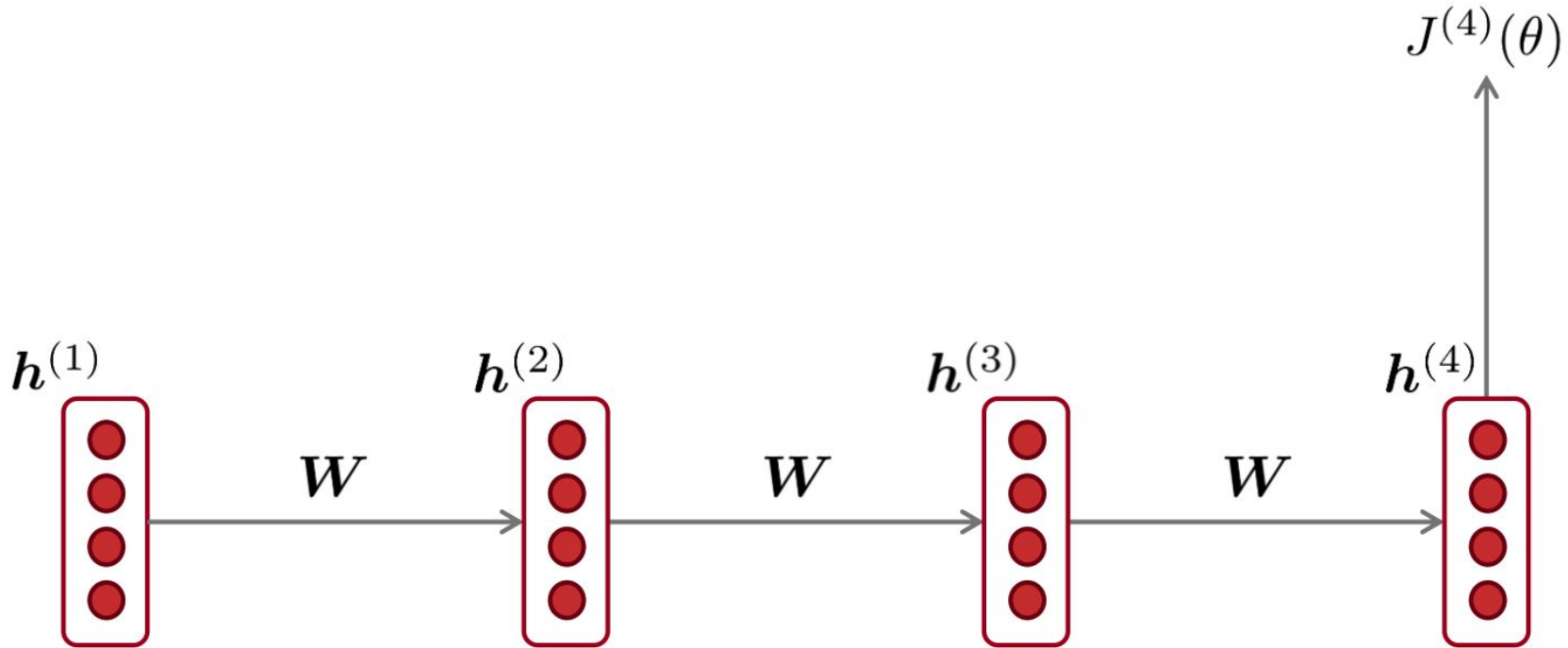
RNN as encoder for sequential data



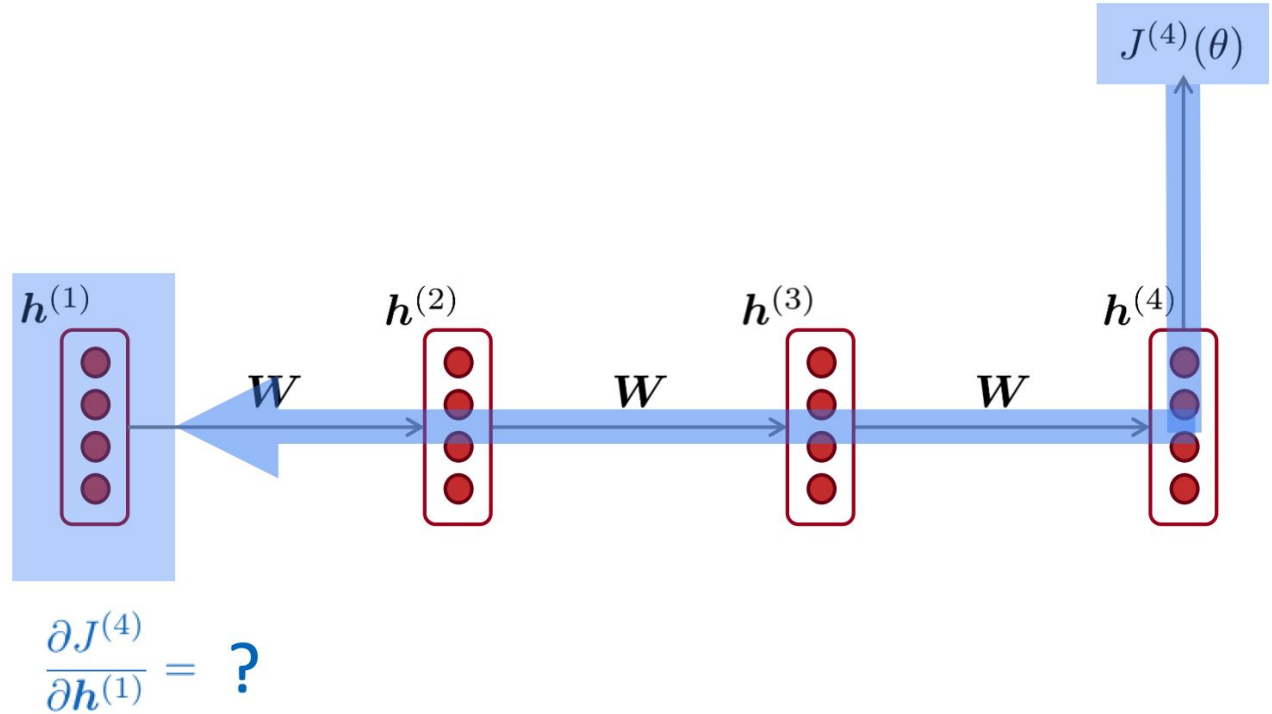
RNNs can be used to encode an input sequence in a fixed size vector.

This vector can be treated as a representation of input sequence.

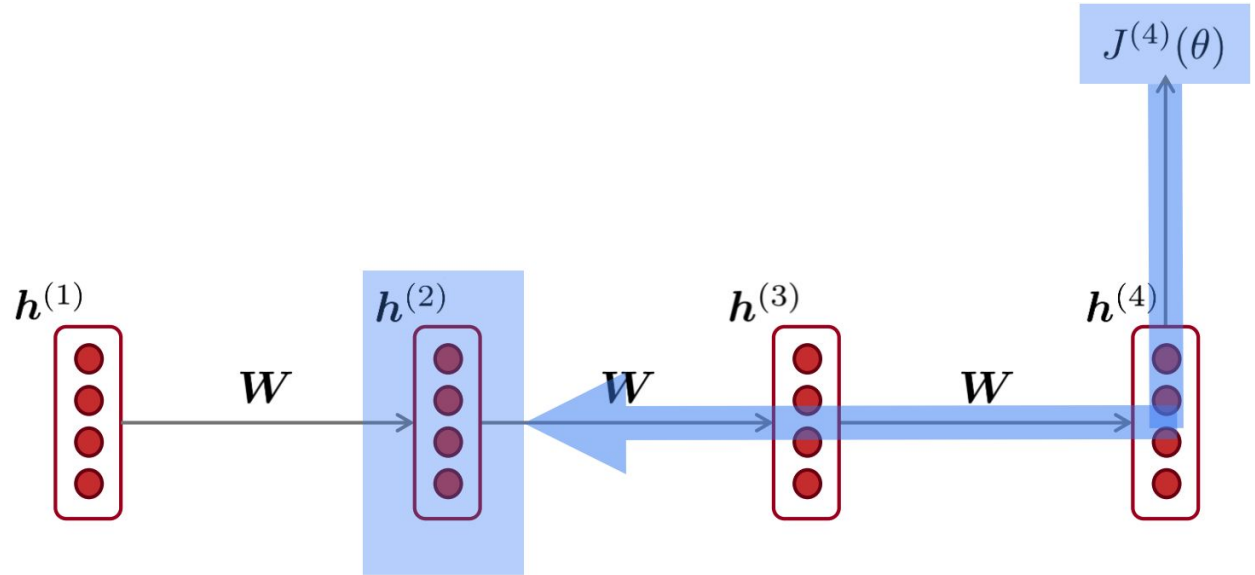
Vanishing gradient problem



Vanishing gradient problem



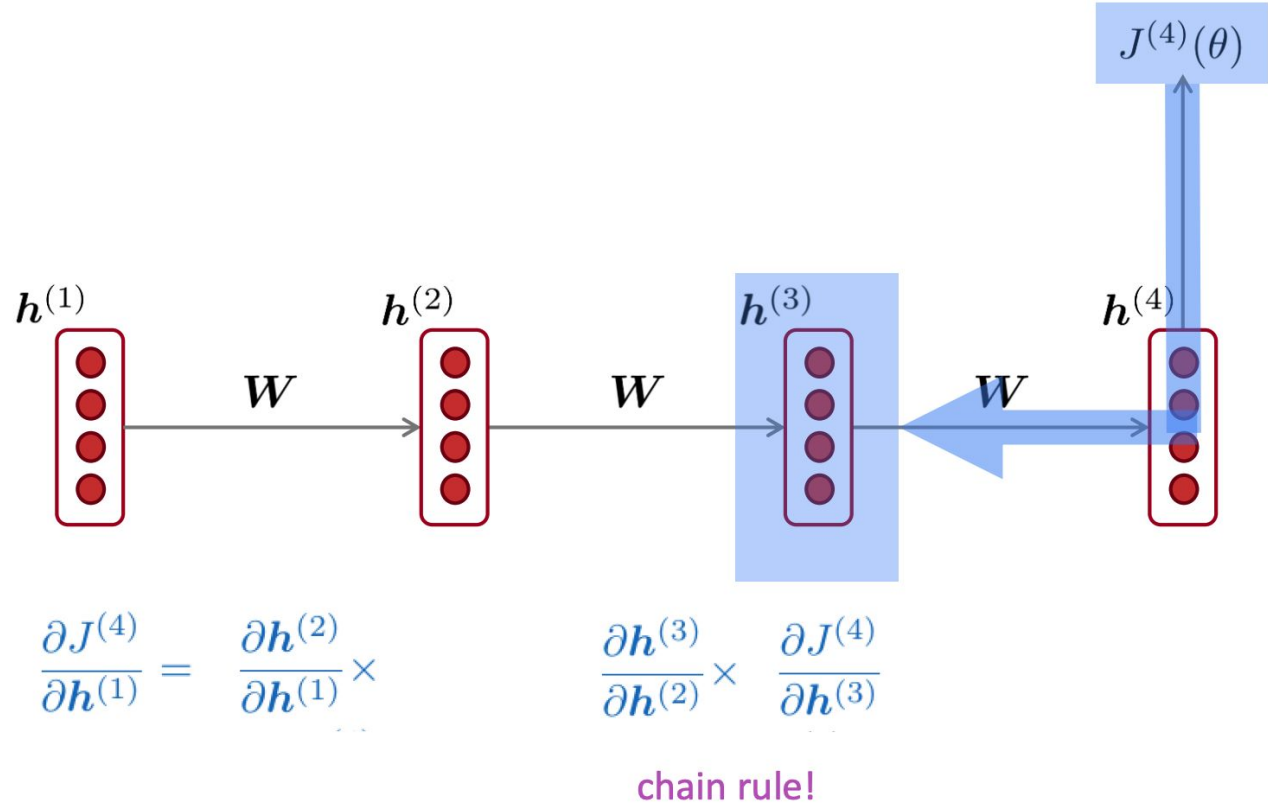
Vanishing gradient problem



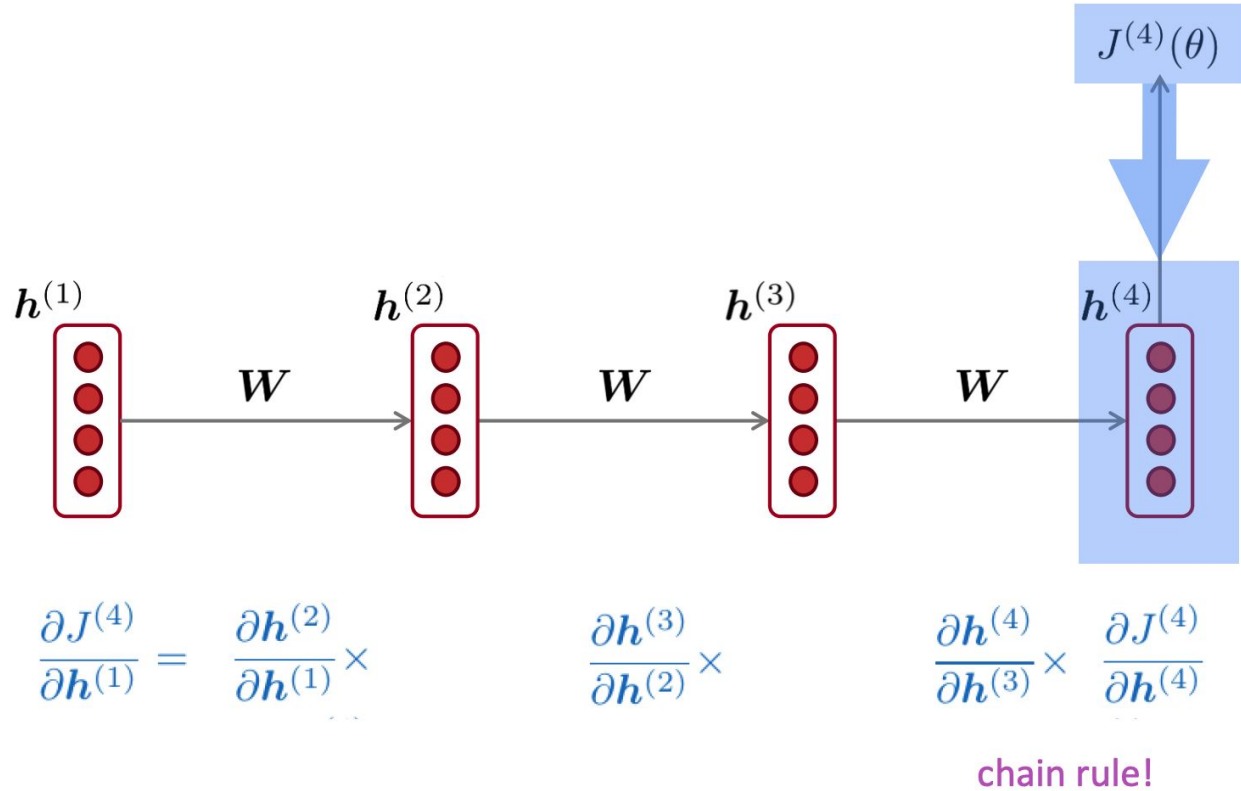
$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial J^{(4)}}{\partial h^{(2)}}$$

chain rule!

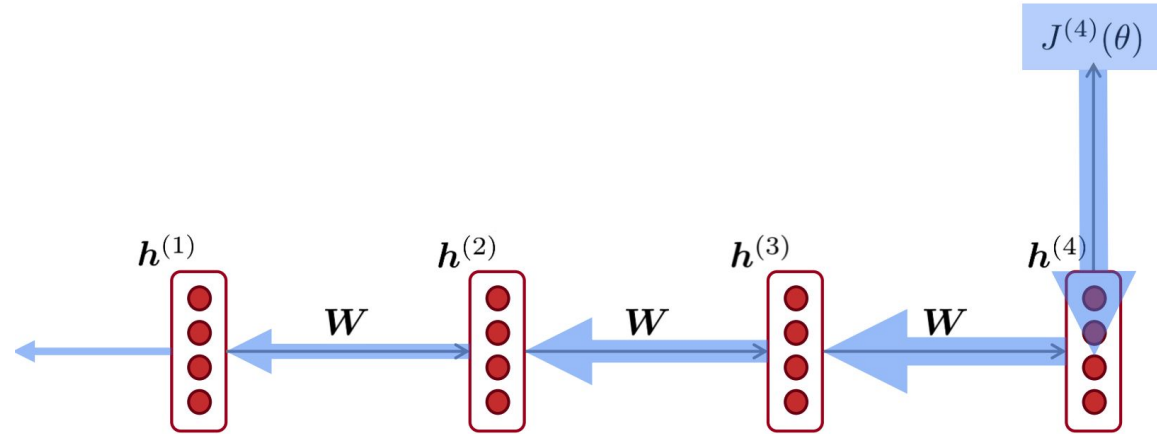
Vanishing gradient problem



Vanishing gradient problem



Vanishing gradient problem



$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

What happens if these are small?

Vanishing gradient problem:

When the derivatives are small, the gradient signal gets smaller and smaller as it backpropagates further

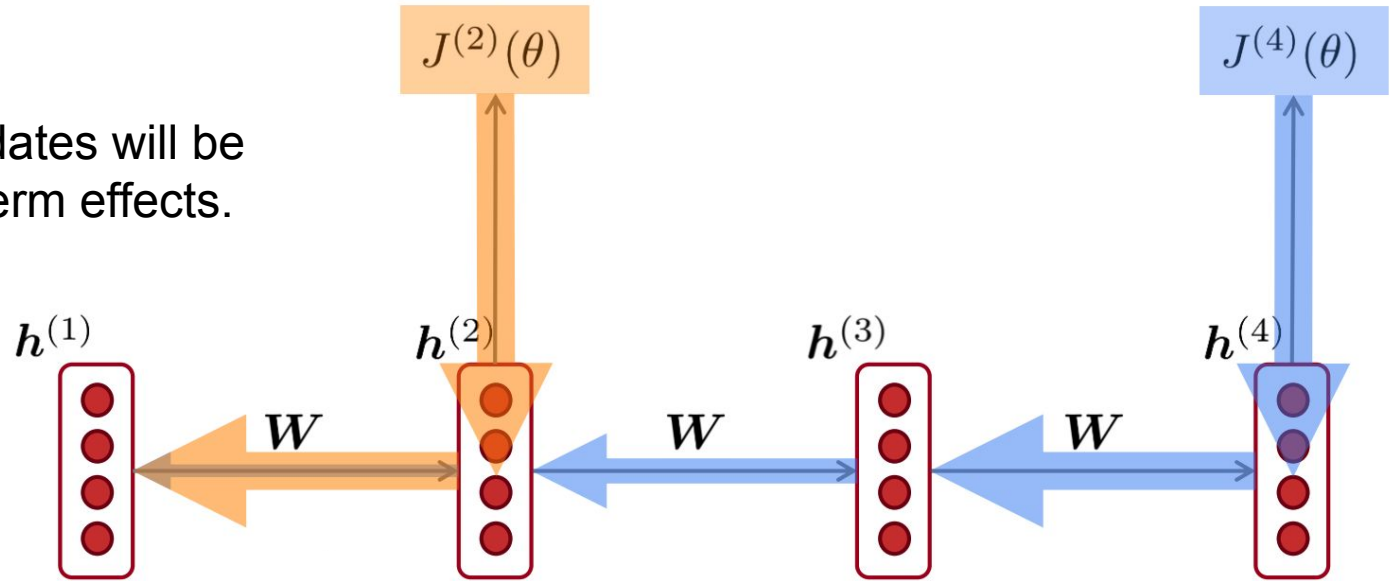
More info: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013

<http://proceedings.mlr.press/v28/pascanu13.pdf>

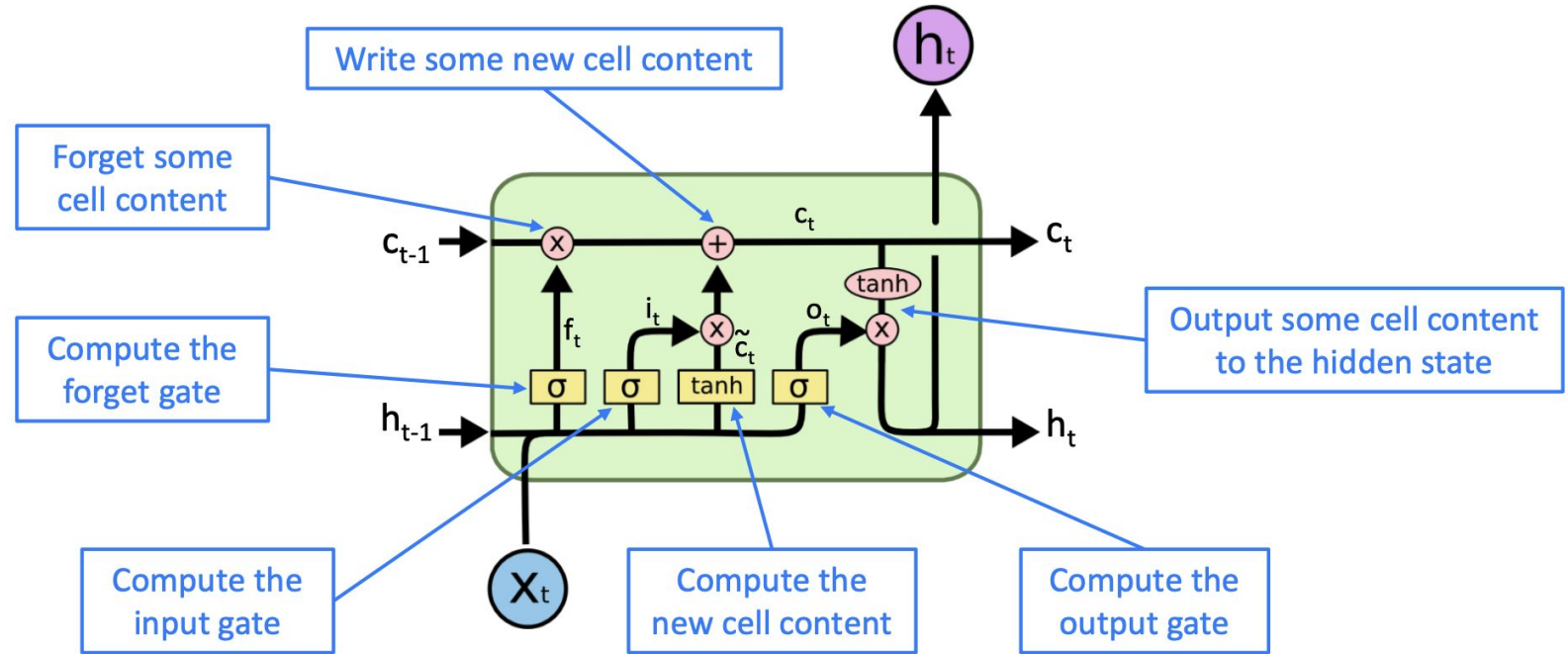
Vanishing gradient problem

Gradient signal from **far away** is lost because it's much smaller than from **close-by**.

So model weights updates will be based only on short-term effects.



Vanishing gradient: LSTM



Vanishing gradient: LSTM

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$f^{(t)} = \sigma \left(W_f h^{(t-1)} + U_f x^{(t)} + b_f \right)$$

$$i^{(t)} = \sigma \left(W_i h^{(t-1)} + U_i x^{(t)} + b_i \right)$$

$$o^{(t)} = \sigma \left(W_o h^{(t-1)} + U_o x^{(t)} + b_o \right)$$

$$\tilde{c}^{(t)} = \tanh \left(W_c h^{(t-1)} + U_c x^{(t)} + b_c \right)$$

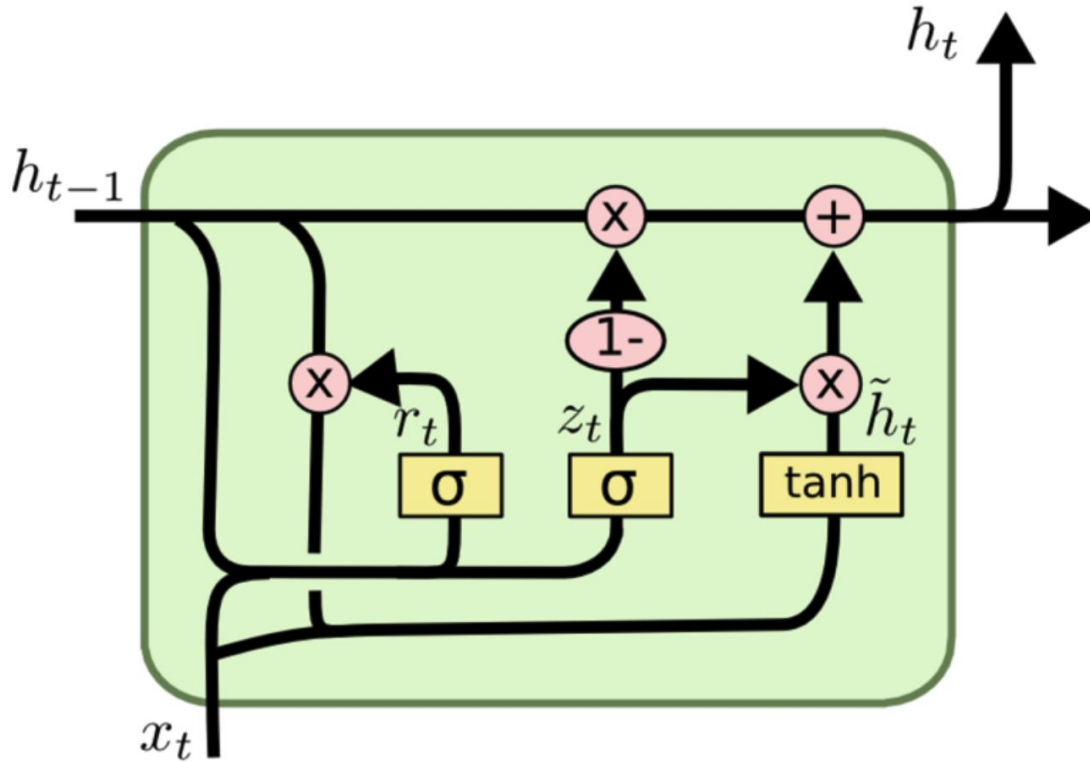
$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}$$

$$h^{(t)} = o^{(t)} \circ \tanh c^{(t)}$$

All these are vectors of same length n

Gates are applied using element-wise product

Vanishing gradient: GRU



Vanishing gradient: GRU

Update gate: controls what parts of hidden state are updated vs preserved

$$\mathbf{u}^{(t)} = \sigma \left(\mathbf{W}_u \mathbf{h}^{(t-1)} + \mathbf{U}_u \mathbf{x}^{(t)} + \mathbf{b}_u \right)$$

Reset gate: controls what parts of previous hidden state are used to compute new content

$$\mathbf{r}^{(t)} = \sigma \left(\mathbf{W}_r \mathbf{h}^{(t-1)} + \mathbf{U}_r \mathbf{x}^{(t)} + \mathbf{b}_r \right)$$

New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

$$\tilde{\mathbf{h}}^{(t)} = \tanh \left(\mathbf{W}_h (\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}) + \mathbf{U}_h \mathbf{x}^{(t)} + \mathbf{b}_h \right)$$

$$\mathbf{h}^{(t)} = (1 - \mathbf{u}^{(t)}) \circ \mathbf{h}^{(t-1)} + \mathbf{u}^{(t)} \circ \tilde{\mathbf{h}}^{(t)}$$

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

How does this solve vanishing gradient?

Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

Vanishing gradient: LSTM vs GRU

- LSTM and GRU are both great
 - GRU is quicker to compute and has fewer parameters than LSTM
 - There is no conclusive evidence that one consistently performs better than the other
 - LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data)

Vanishing gradient in non-RNN

Vanishing gradient is present in **all** deep neural network architectures.

- Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small during backpropagation
- Lower levels are hard to train and are trained slower
- **Potential solution:** direct (or skip-) connections (just like in ResNet)

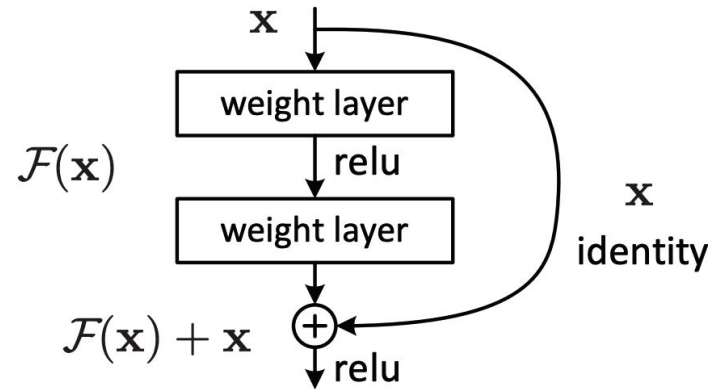
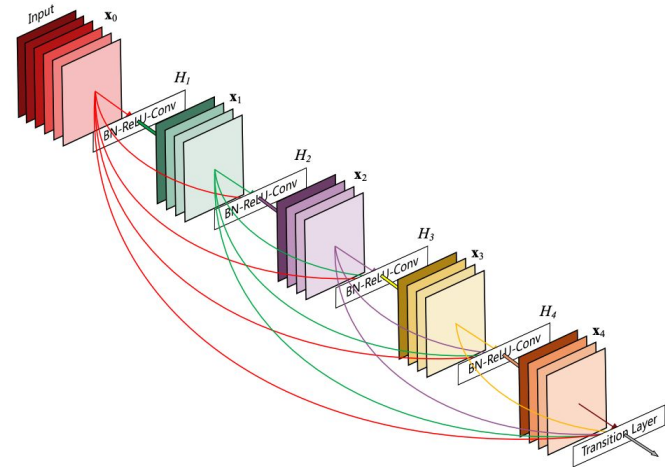


Figure 2. Residual learning: a building block.

Vanishing gradient in non-RNN

Vanishing gradient is present in **all** deep neural network architectures.

- Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small during backpropagation
- Lower levels are hard to train and are trained slower
- **Potential solution:** dense connections (just like in DenseNet)



- RNN is a great choice for data with sequential structure
- Multi-layer RNN can also be of great use
- **Rule of thumb:** start with LSTM, but switch to GRU if you want something more efficient

