BACKPROPAGATION ADVANCED

Постановка задачи

Дано:

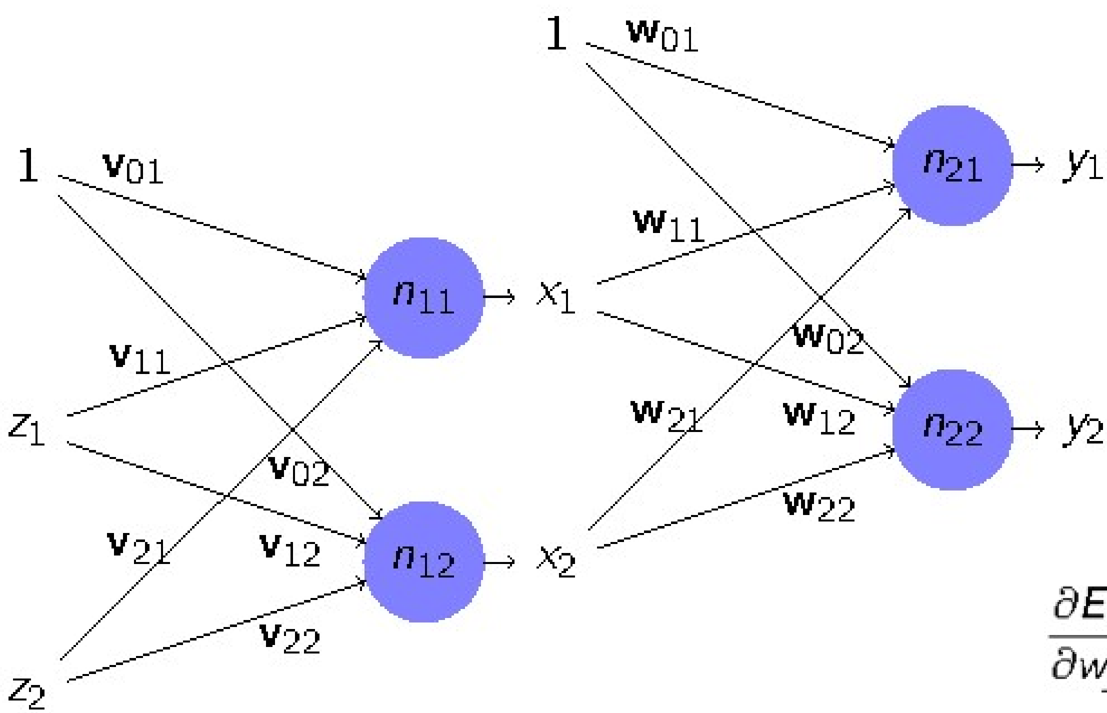
$\mathcal{X} = (X_1, \ldots, X_k)$	**входные вектора,** $X_i \in \mathbb{R}^n$
$\mathcal{A} = (A_1, \ldots, A_k)$	**правильные выходные вектора,** $A_i \in \mathbb{R}^m$
$(\mathcal{X}, \mathcal{A})$	**обучающая выборка**
W	**вектор весов нейронной сети**
$N(W, X)$	**функция, соответствующая нейронной сети**
$Y = N(W, X)$	**ответ нейронной сети,** $Y \in \mathbb{R}^m$
$D(Y, A) = \sum_{j=1}^{m} (Y[j] - A[j])^2$	**функция ошибки**
$D_i(Y) = D(Y, A_i)$	**функция ошибки на** i**-ом примере**
$E_i(W) = D_i(N(W, X_i))$	**ошибка сети на** i**-ом примере**
$E(W) = \sum_{i=1}^{k} E_i(W)$	**ошибка сети на всей обучающей выборке**

Найти:

вектор W **такой, что** $E(W) \to \min$ **(обучение на всей выборке)**
вектор W **такой, что** $E_i(W) \to \min$ **(обучение на одном примере)**

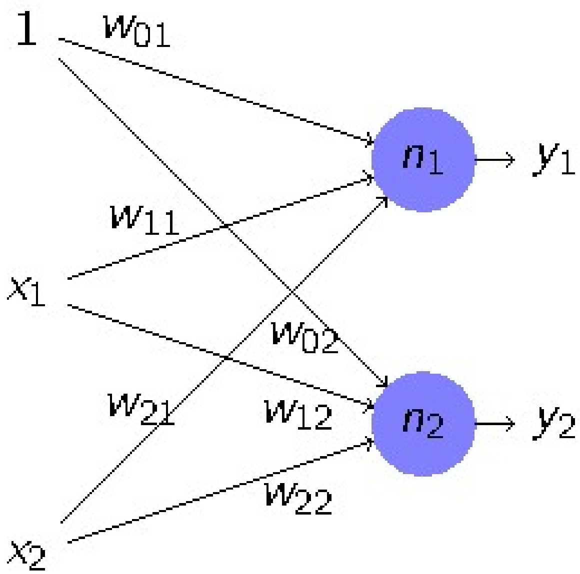
$$
\begin{aligned}
\frac{\partial E(\vec{w})}{\partial w_j} &= \frac{\partial}{\partial w_j} \sum_i (h_{\vec{w}}(\overrightarrow{x_i}) - y_i)^2 \\
&= \sum_i 2(h_{\vec{w}}(\overrightarrow{x_i}) - y_i) \frac{\partial}{\partial w_j} (h_{\vec{w}}(\overrightarrow{x_i}) - y_i) \\
&= \sum_i 2(h_{\vec{w}}(\overrightarrow{x_i}) - y_i) \frac{\partial}{\partial w_j} \sigma(\overrightarrow{x_i} \cdot \vec{w}) \\
&= \sum_i 2(h_{\vec{w}}(\overrightarrow{x_i}) - y_i)\ \sigma'(\overrightarrow{x_i} \cdot \vec{w}) \frac{d}{dw_j} \overrightarrow{x_i} \cdot \vec{w} \\
&= \sum_i 2(h_{\vec{w}}(\overrightarrow{x_i}) - y_i)\ \sigma'(\overrightarrow{x_i} \cdot \vec{w}) \frac{d}{dw_j} \sum_{k=1}^{n} x_{i,k} w_k \\
&= 2 \sum_i (h_{\vec{w}}(\overrightarrow{x_i}) - y_i)\ \sigma'(\overrightarrow{x_i} \cdot \vec{w}) x_{i,j}
\end{aligned}
$$

Обратное распространение ошибки



$$\frac{\partial E_k}{\partial w_{ji}} = \sum_{l=1}^{n} \frac{\partial D_k}{\partial y_l} \frac{\partial y_l}{\partial w_{ji}} = 2(y_i - a_i) f'(S_i) x_j$$

Обратное распространение ошибки



$$D_k(y_1, ..., y_n) = (y_i - a_i)^2 + ... + (y_n - a_n)^2$$

$$\frac{\partial D_k}{\partial y_i} = 2(y_i - a_i)$$

$$S_i = \sum_{j=0}^{m} x_j w_{ji} \qquad y_i = f(S_i) \qquad \frac{\partial y_i}{\partial x_j} = f'(S_i) w_{ji}$$

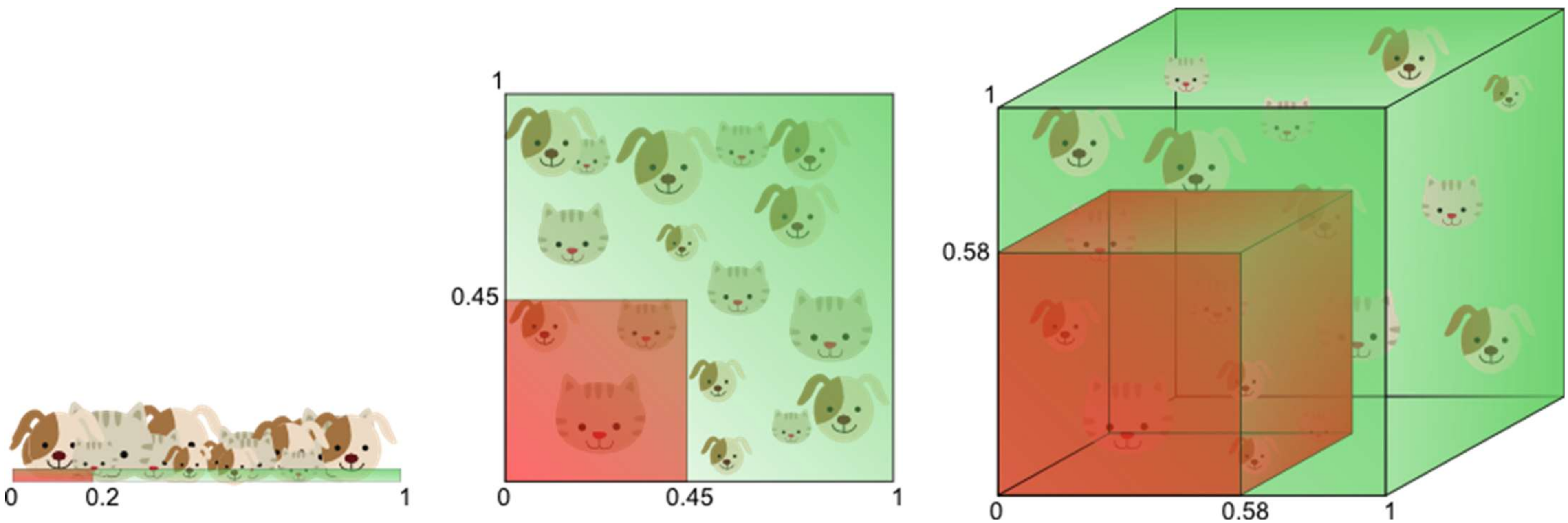
$$\frac{\partial D_k}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial D_k}{\partial y_i} \frac{\partial y_i}{\partial x_j} =$$

$$= 2 \sum_{i=1}^{n} (y_i - a_i) f'(S_i) w_{ji}$$

Algorythm of BP

```
initialize network weights (often small random values)
do
   forEach training example named ex
      prediction = neural-net-output(network, ex)  // forward pass
      actual = teacher-output(ex)
      compute error (prediction - actual) at the output units
      compute Δw_h for all weights from hidden layer to output layer  // backward pass
      compute Δw_i for all weights from input layer to hidden layer   // backward pass continued
      update network weights // input layer not modified by error estimate
until all examples classified correctly or another stopping criterion satisfied
return the network
```

The curse of dimensionality



Plan

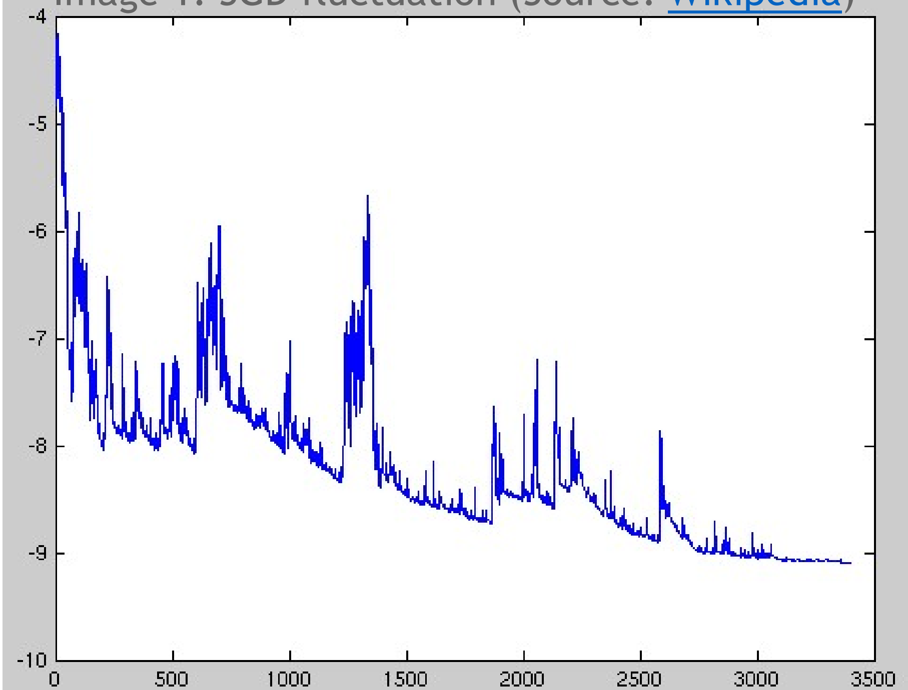
- Gradient descent variants
- Challenges
- Gradient descent optimization algorithms
- Parallelizing and distributing SGD
- Additional strategies for optimizing SGD

Batch gradient descent

Stochastic: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$.

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

Image 1: SGD fluctuation (Source: Wikipedia)



Batch vs Stochastic

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

Challenges: Vanilla mini-batch gradient descent

- Choosing a proper learning rate can be difficult
- Learning rate schedules try to adjust the learning rate during.
- Additionally, the same learning rate applies to all parameter updates.
- Suboptimal local minima.

Mini-batch gradient descent

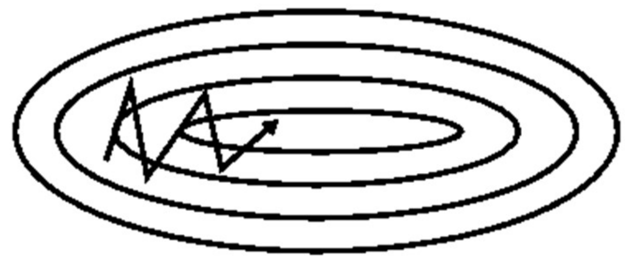
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad
```

Gradient descent optimization algorithms

Momentum
Nesterov accelerated gradient
Adagrad
Adadelta
RMSprop
Adam
AdaMax
Nadam
Visualization of algorithms
Which optimizer to choose?

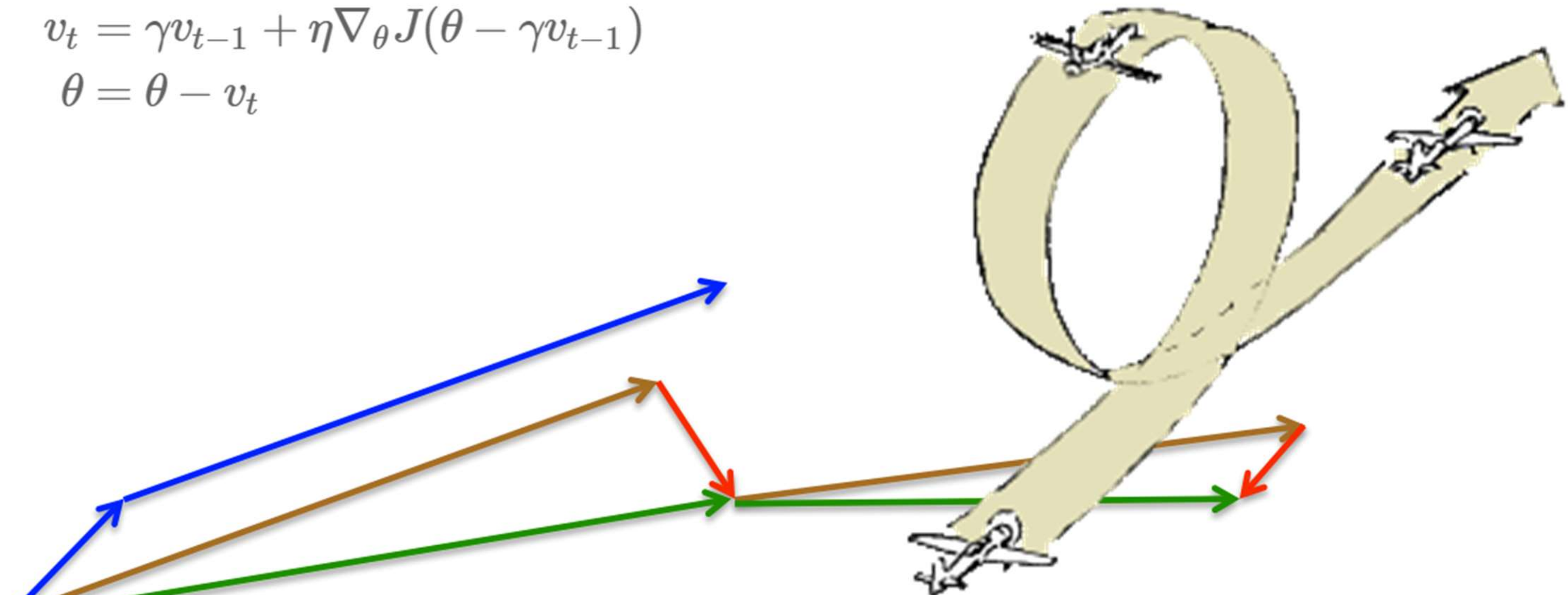
Momentum



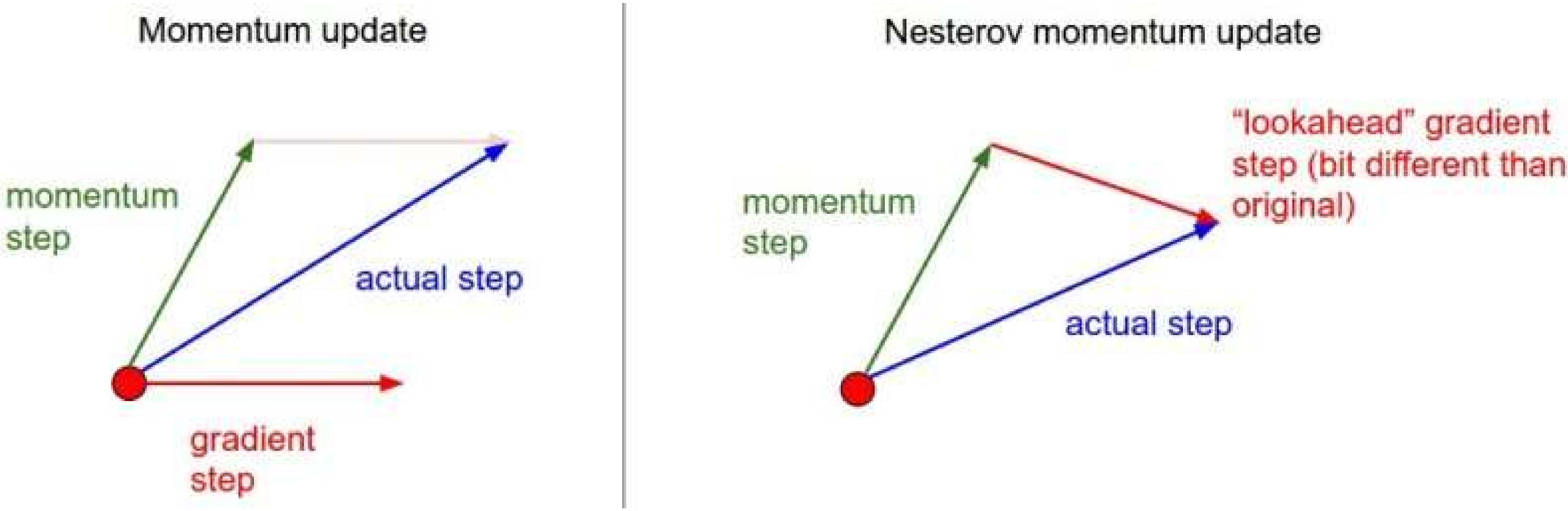
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Nesterov accelerated gradient

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

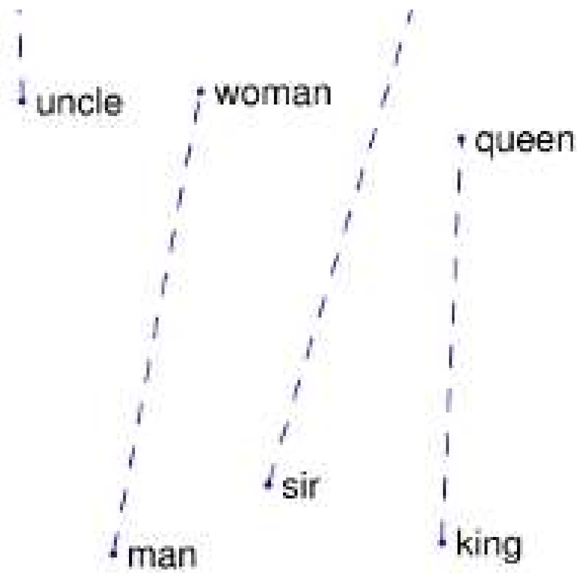


Momentum vs Nesterov Acc. Grad.



Adagrad

It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.



Adagrad
Adaptive Gradients

$g_{t,i} = \nabla_\theta J(\theta_i).$ the gradient of the objective function w.r.t. to the parameter θ at time step t

$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$ The SGD update for every parameter θ at each time step t then becomes

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

Adagrad

$G_t \in \mathbb{R}^{d \times d}$ diagonal matrix the sum of the squares of the gradients

$$G = \sum_{\tau=1}^{t} g_\tau g_\tau^\top$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}. \quad \Rightarrow \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

Adadelta

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Adadelta

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

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t. \quad \Longrightarrow \quad \Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

Adadelta

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2. \quad (*)$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}.$$

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

RMSprop

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

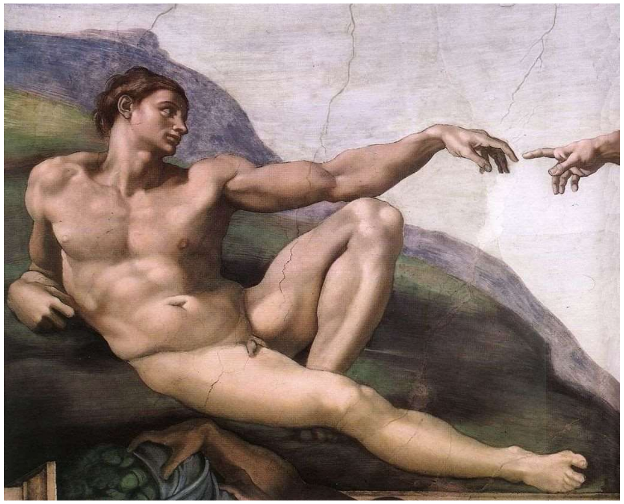
Geoff Hinton

Adam
Adaptive Moment Estimation

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

The Analogue between Adam and (RMSProp, Ada*)

Init m and v: 0



$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

Nadam
Nesterov-accelerated Adaptive Moment Estimation

$$g_t = \nabla_{\theta_t} J(\theta_t)$$
$$m_t = \gamma m_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - m_t$$

Recall of momentum

$$\theta_{t+1} = \theta_t - (\gamma m_{t-1} + \eta g_t)$$

Recall of NAG

$$g_t = \nabla_{\theta_t} J(\theta_t - \gamma m_{t-1})$$
$$m_t = \gamma m_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - m_t$$

Nadam

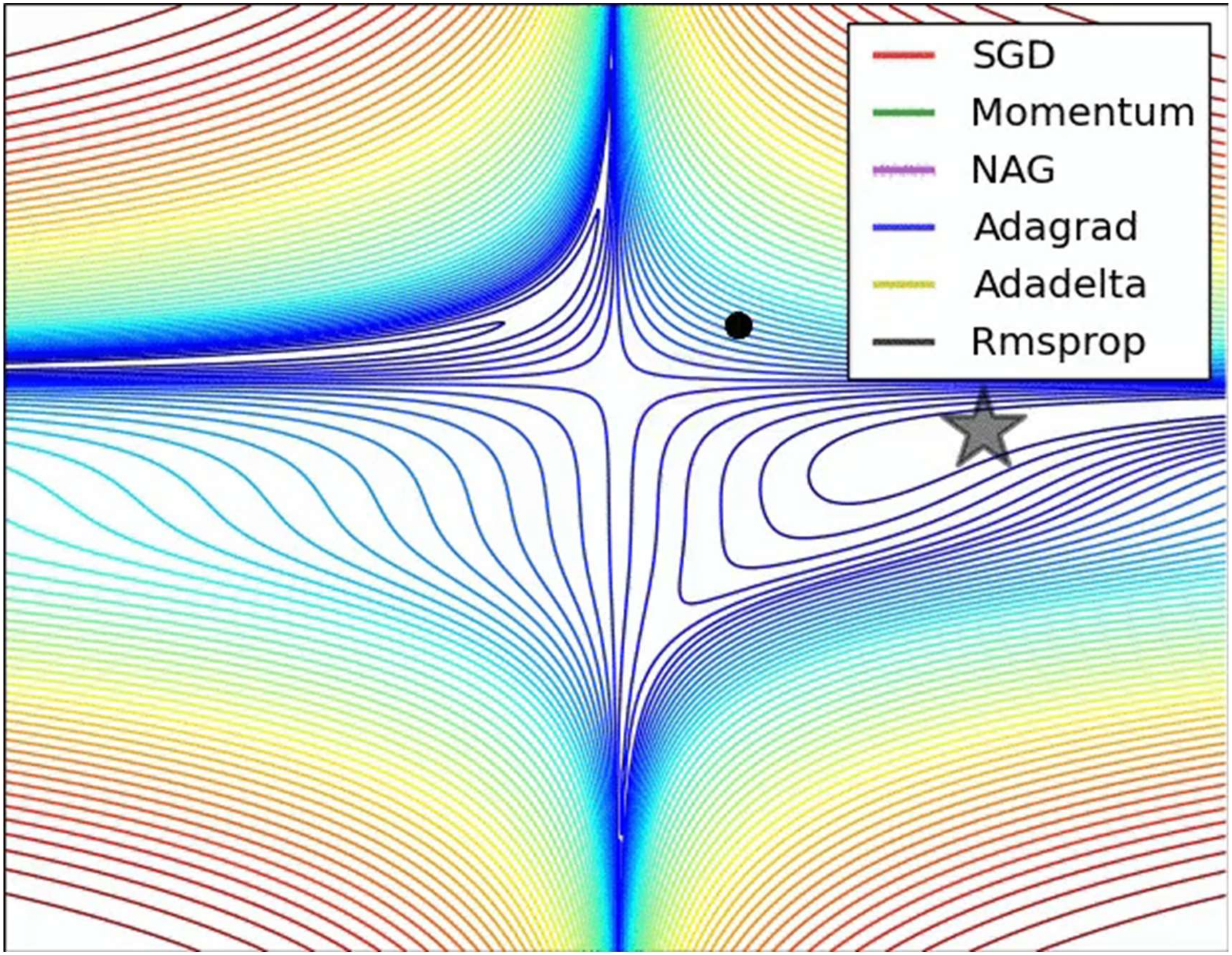
$$g_t = \nabla_{\theta_t} J(\theta_t)$$
$$m_t = \gamma m_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - (\gamma m_t + \eta g_t)$$

Another way of NAG

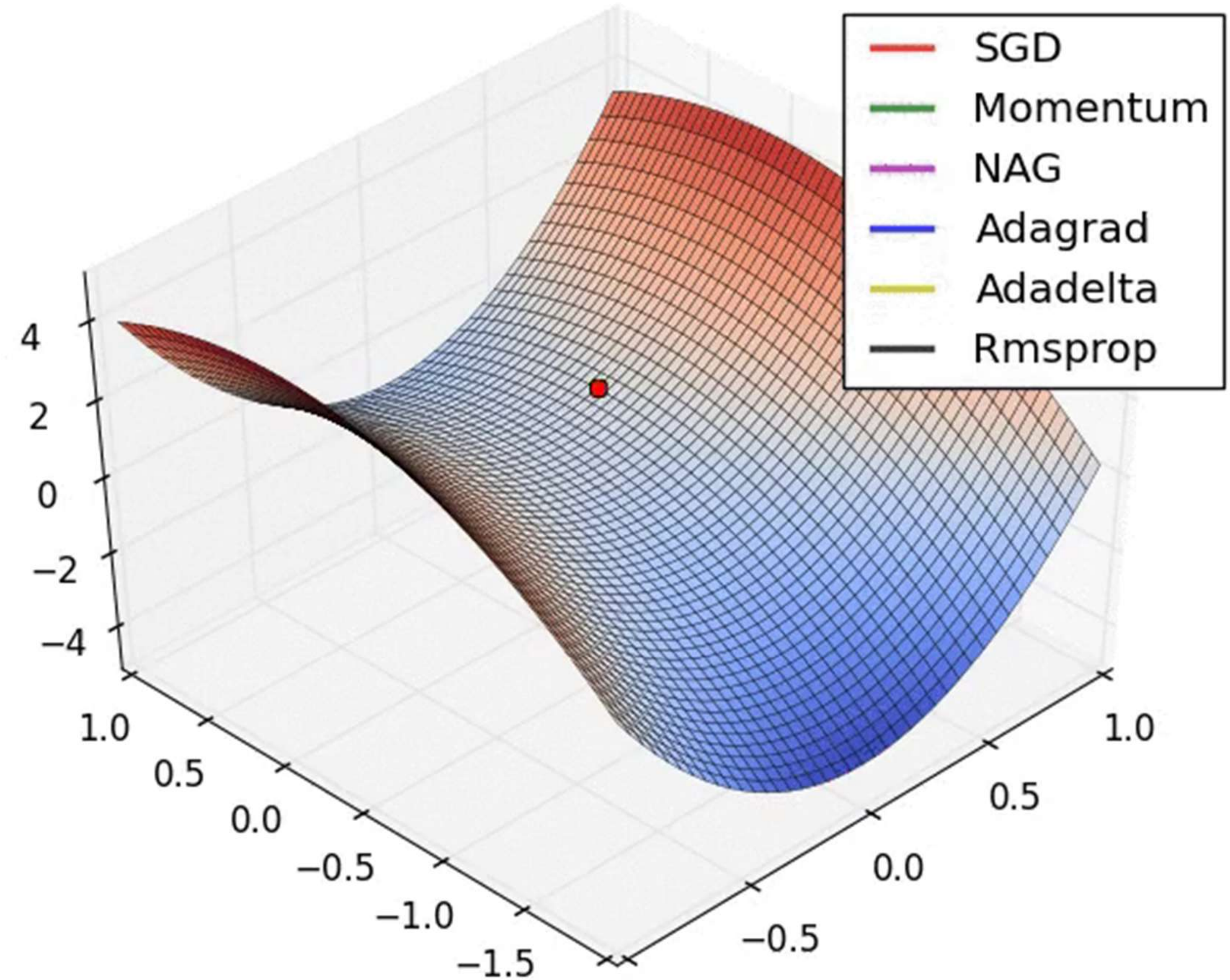
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Nadam

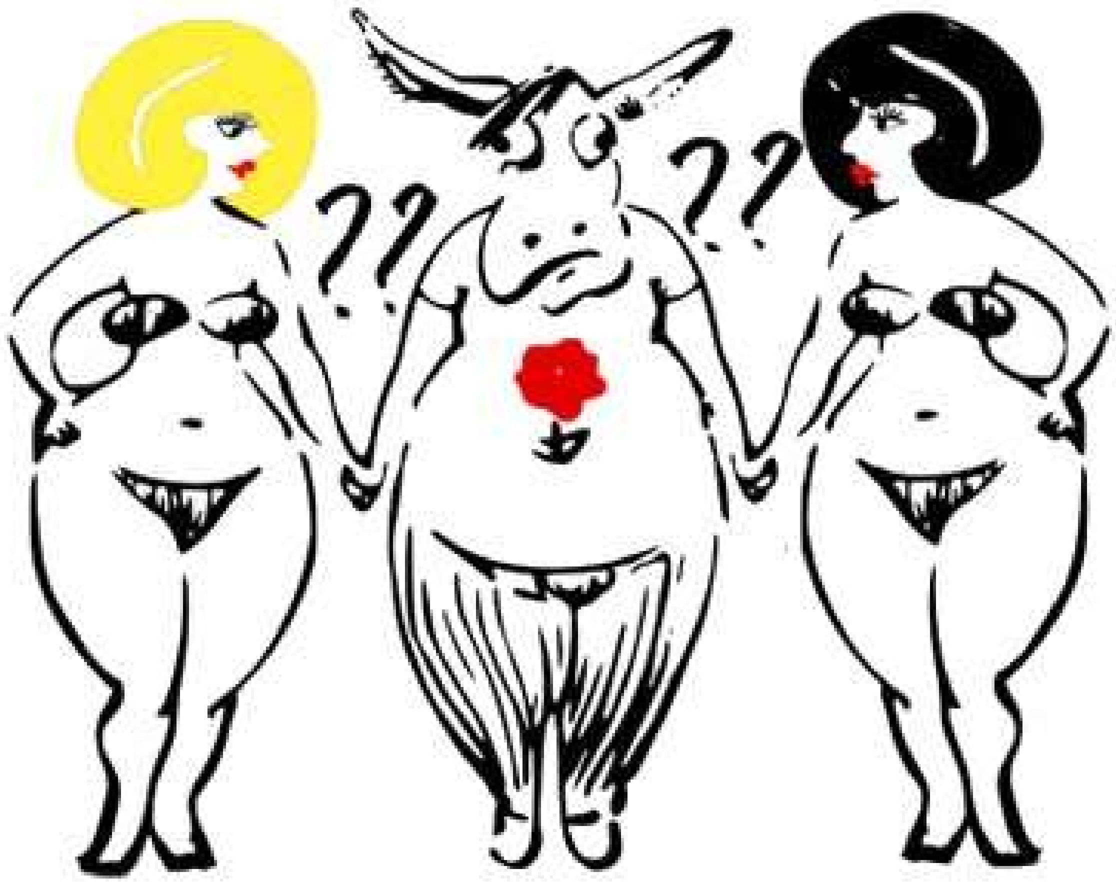
Alec Radford



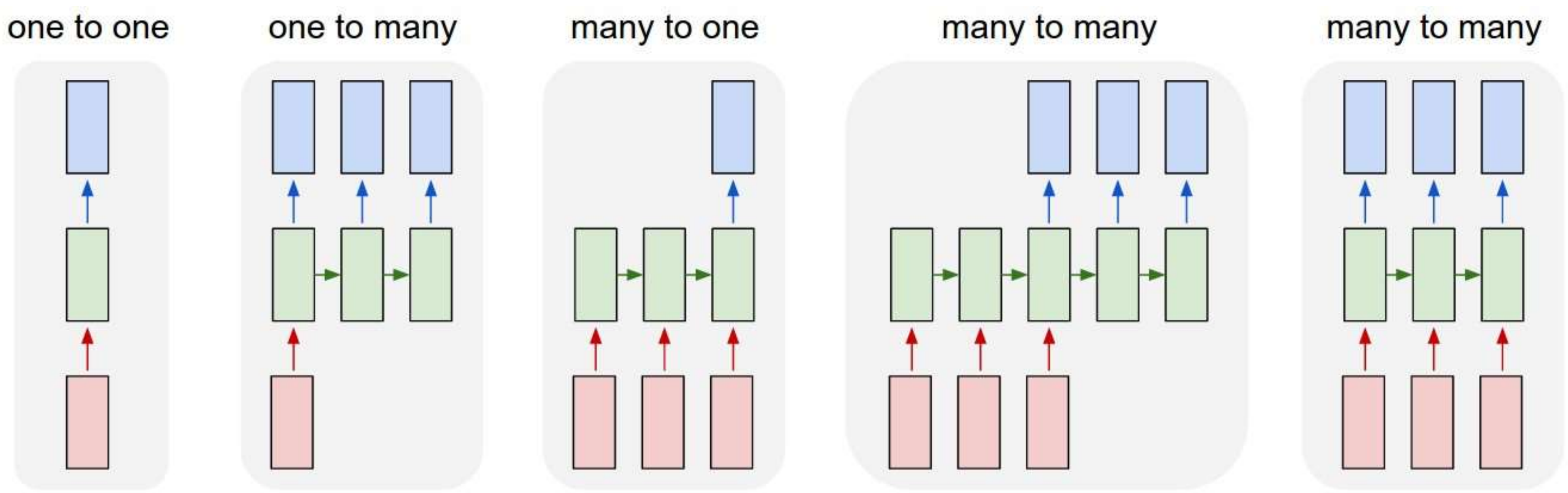
Alec Radford



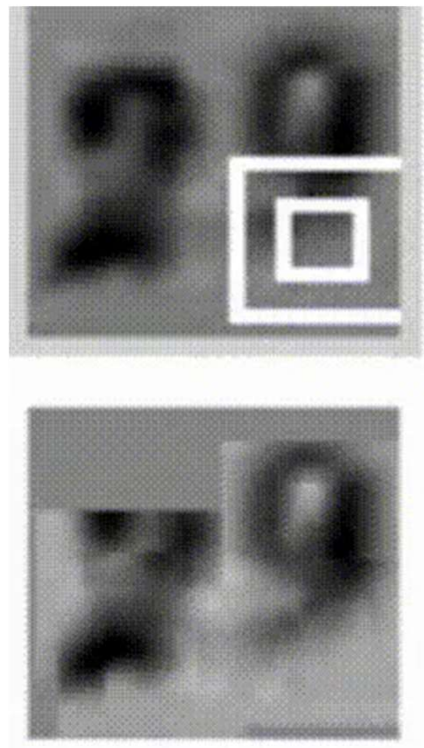
Which one to use?



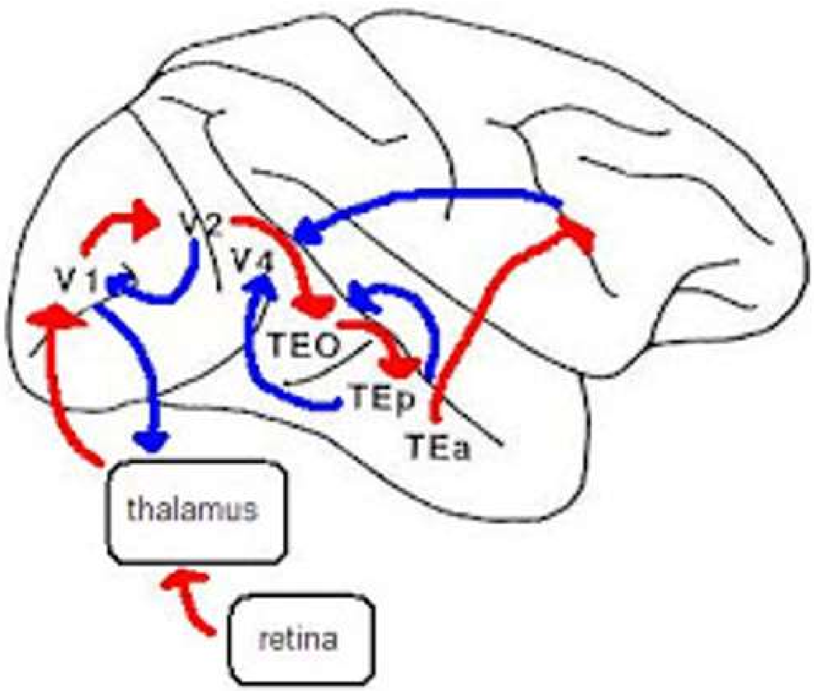
RNN



Example



Временные ряды: Рабочая память



Recurrent Processing - concious?

Long Short-Term Memory (LSTM)

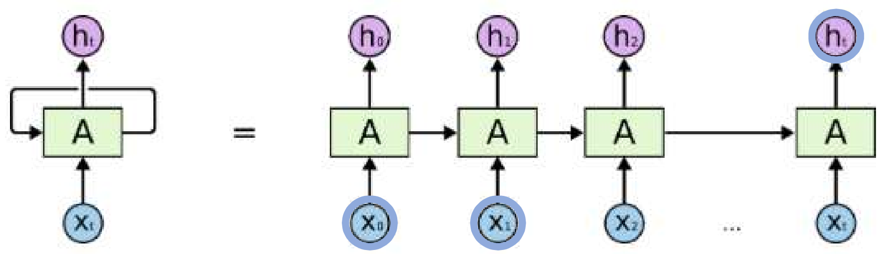
Recurrent Neural Net

RNN Code Example

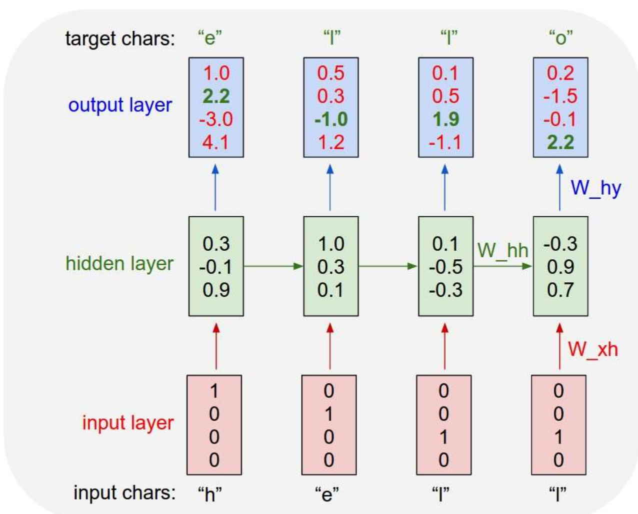
```
rnn = RNN()
y = rnn.step(x) # x is an input vector, y is the RNN's output vector
```

```
class RNN: # ...
    def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

Рекуррентные сети (Long Short-Term Memory)

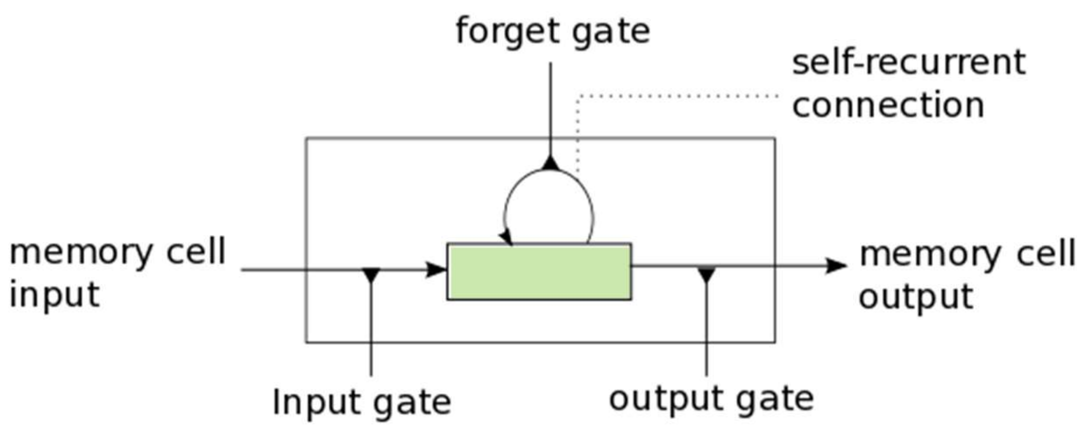


Временная инвариантность



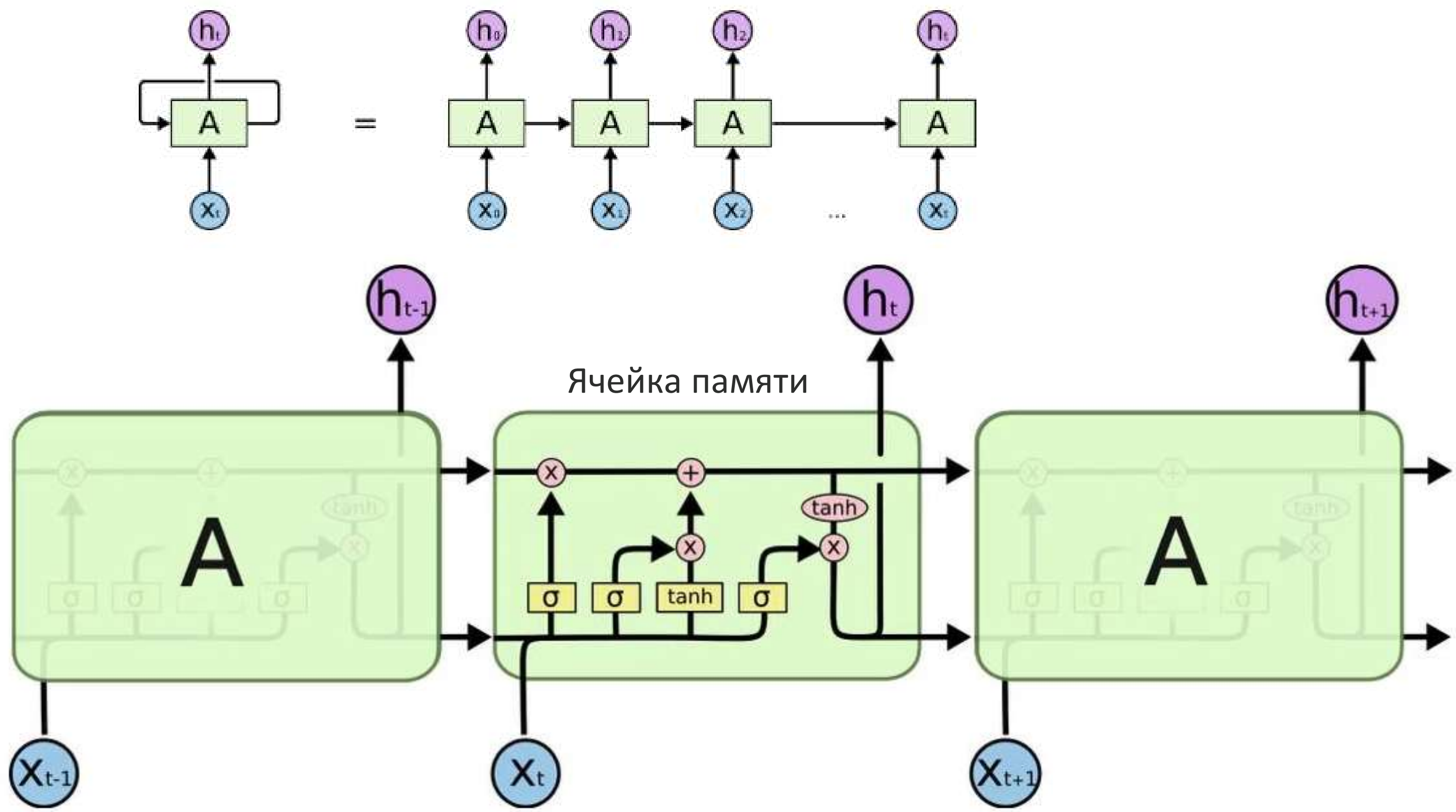
Hochreiter (1997)
Long Short-Term Memory

LSTM Architecture



Ячейка памяти

Рекуррентные сети (Long Short-Term Memory)



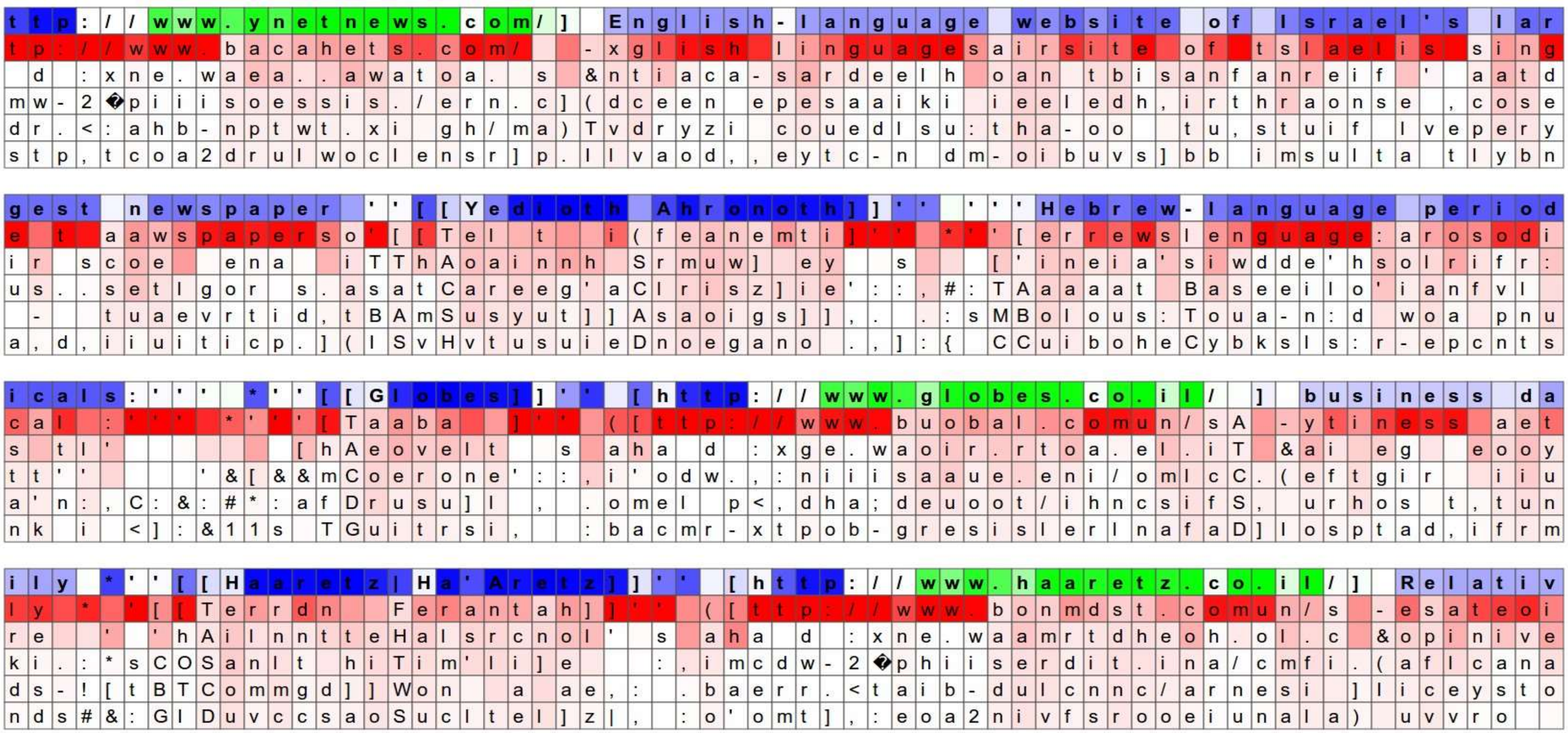
Shakespeare\Wiki\Latex toy models

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e plia tklrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

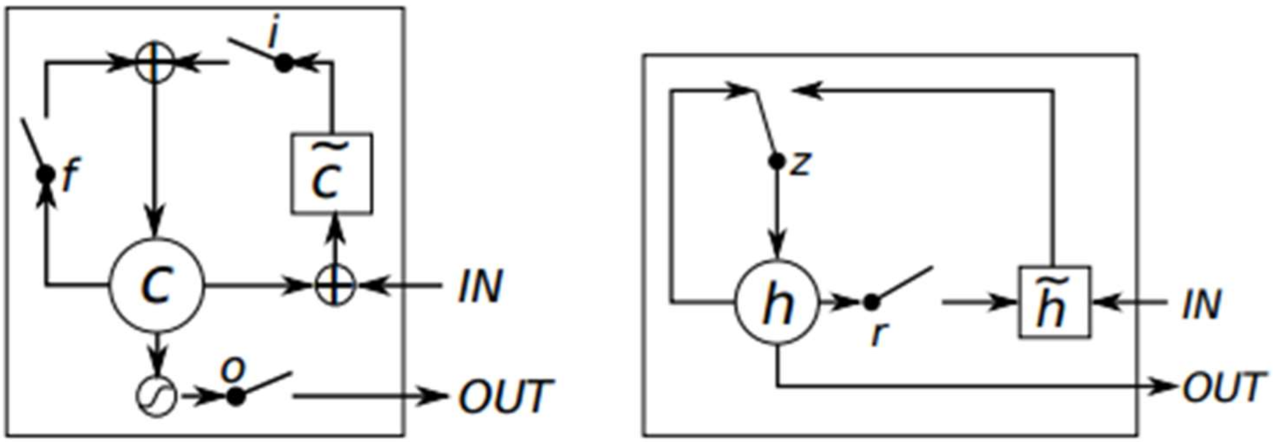
"Tmont thithey" fomesscerliund Keushey. Thom here sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."

we counter. He stutn co des. His stanted out one ofler that concossions and was to gearang reay Jotrets and with fre colt otf paitt thin wall. Which das stimn

LSTM Vizualization



LSTM vs GRU



(a) Long Short-Term Memory

(b) Gated Recurrent Unit

Figure 1: Illustration of (a) LSTM and (b) gated recurrent units. (a) i, f and o are the input, forget and output gates, respectively. c and $\tilde{c}$ denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and $\tilde{h}$ are the activation and the candidate activation.

LSTM output gate

Unlike to the recurrent unit which simply computes a weighted sum of the input signal and applies a nonlinear function, each j-th LSTM unit maintains a memory c_t^j at time t. The output h_t^j, or the activation, of the LSTM unit is then

$$h_t^j = o_t^j \tanh\left(c_t^j\right),$$

where o_t^j is an *output gate* that modulates the amount of memory content exposure. The output gate is computed by

$$o_t^j = \sigma\left(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c}_t\right)^j,$$

where σ is a logistic sigmoid function. V_o is a diagonal matrix.

LSTM memory gate

The memory cell c_t^j is updated by partially forgetting the existing memory and adding a new memory content $\tilde{c}_t^j$:

$$c_t^j = f_t^j c_{t-1}^j + i_t^j \tilde{c}_t^j, \tag{4}$$

where the new memory content is

$$\tilde{c}_t^j = \tanh\left(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1}\right)^j .$$

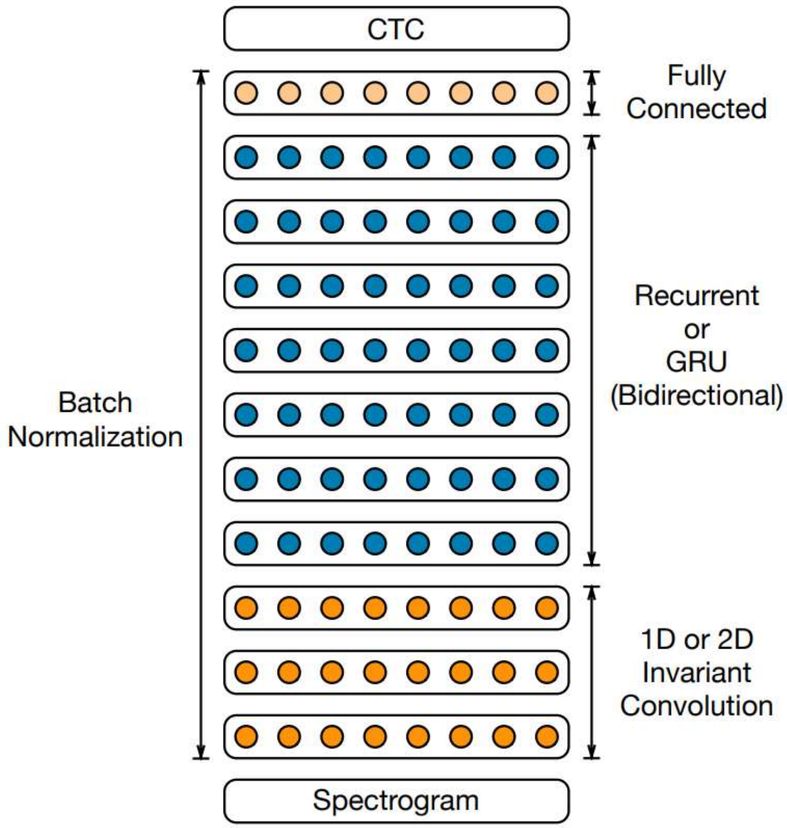
LSTM Inpute gate

The extent to which the existing memory is forgotten is modulated by a *forget gate* f_t^j, and the degree to which the new memory content is added to the memory cell is modulated by an *input gate* i_t^j. Gates are computed by

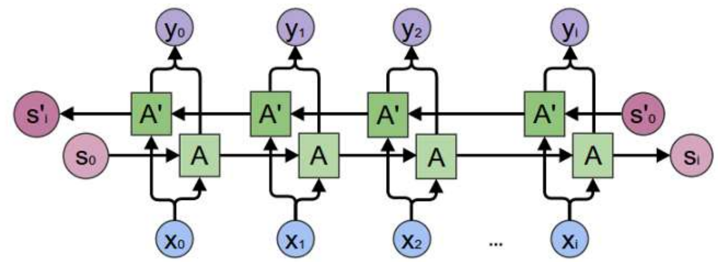
$$
\begin{aligned}
f_t^j &= \sigma\left(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_f \mathbf{c}_{t-1}\right)^j, \\
i_t^j &= \sigma\left(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c}_{t-1}\right)^j.
\end{aligned}
$$

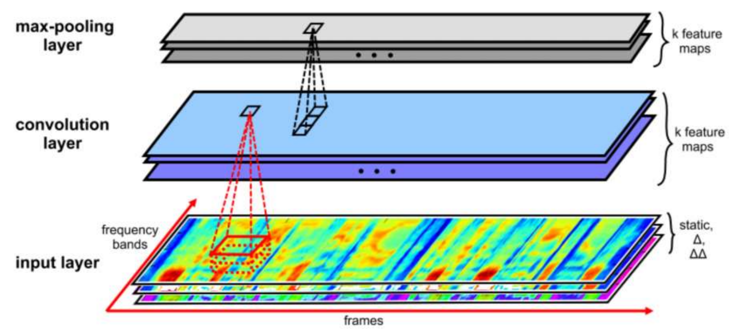
Note that V_f and V_i are diagonal matrices.

Глубокие рекуррентные сети (DLSTM)



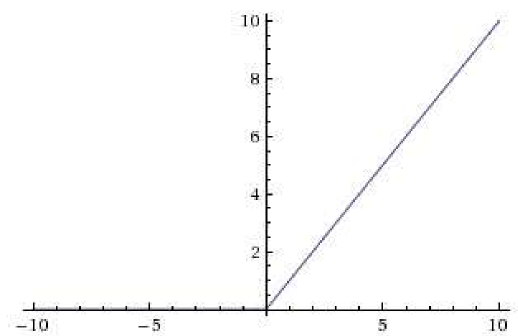
Amodei (2015) *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*



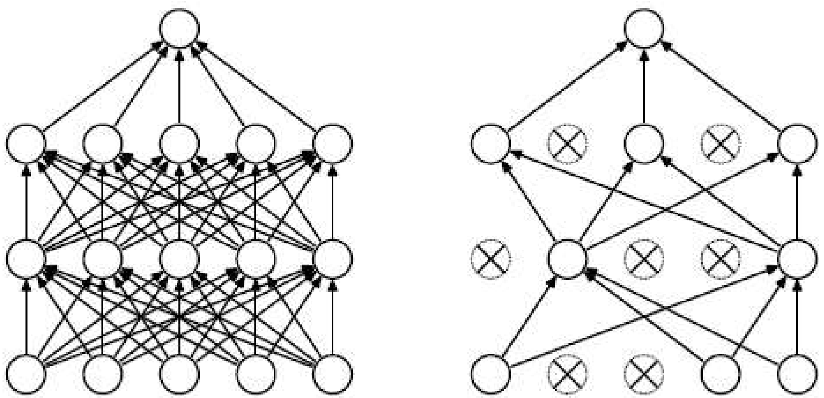


Регуляризация обучения

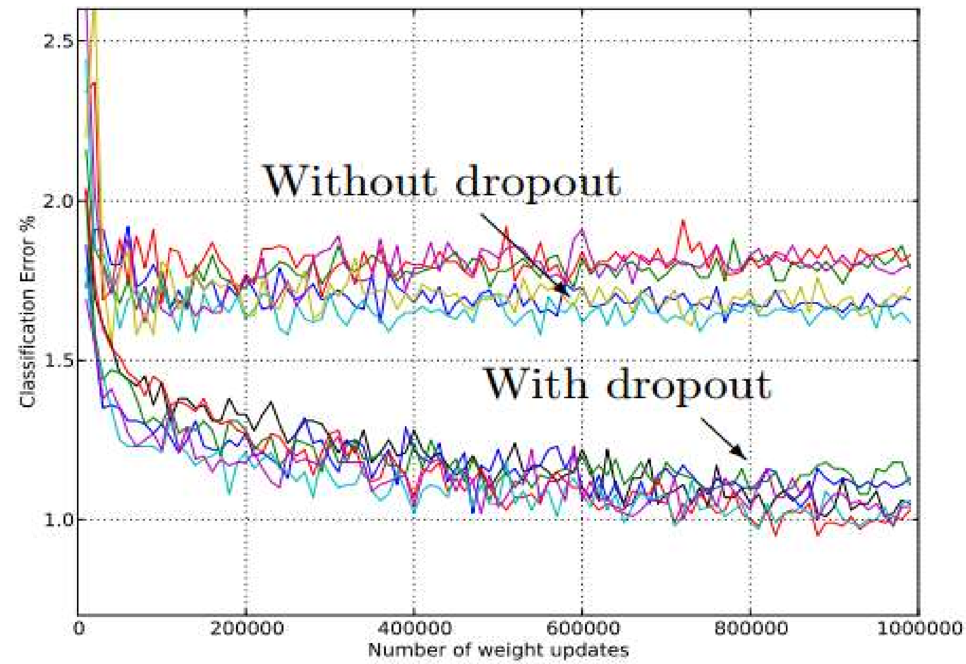
- **ReLU** (Nair, 2010)
- **Dropout** (Hinton, 2012)
- **Batch normalization** (Ioffe, 2015)



Dropout



(a) Standard Neural Net (b) After applying dropout.



Hinton (2012) *Improving neural networks by preventing co-adaptation of feature detectors*

Batch normalization

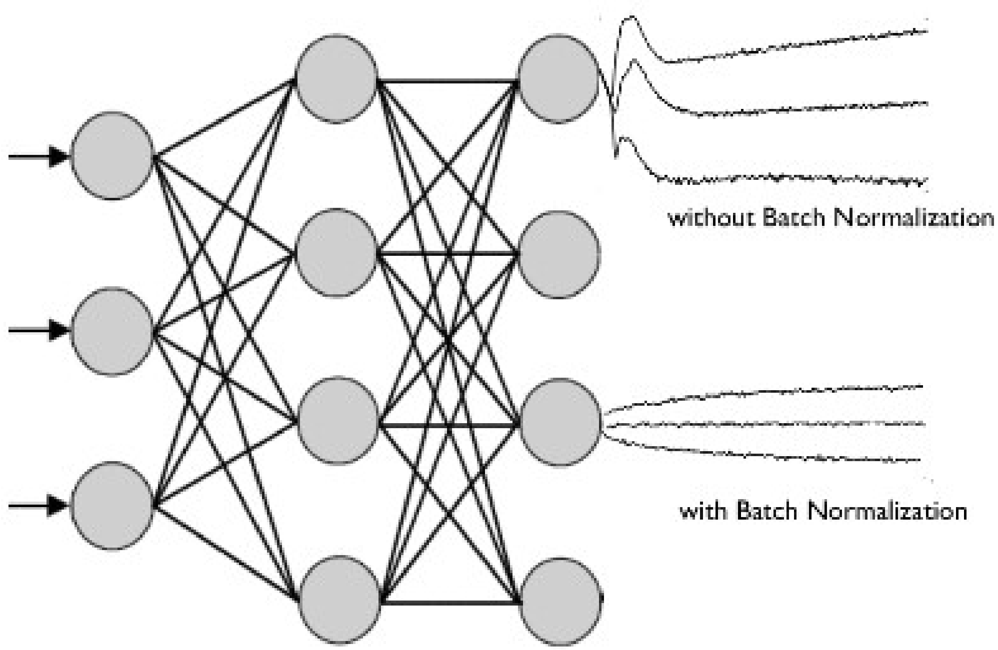
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

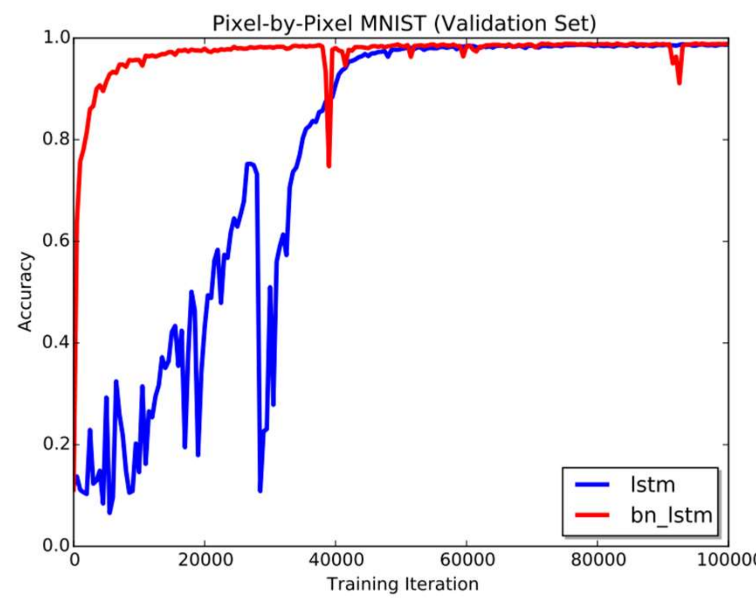
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$$

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

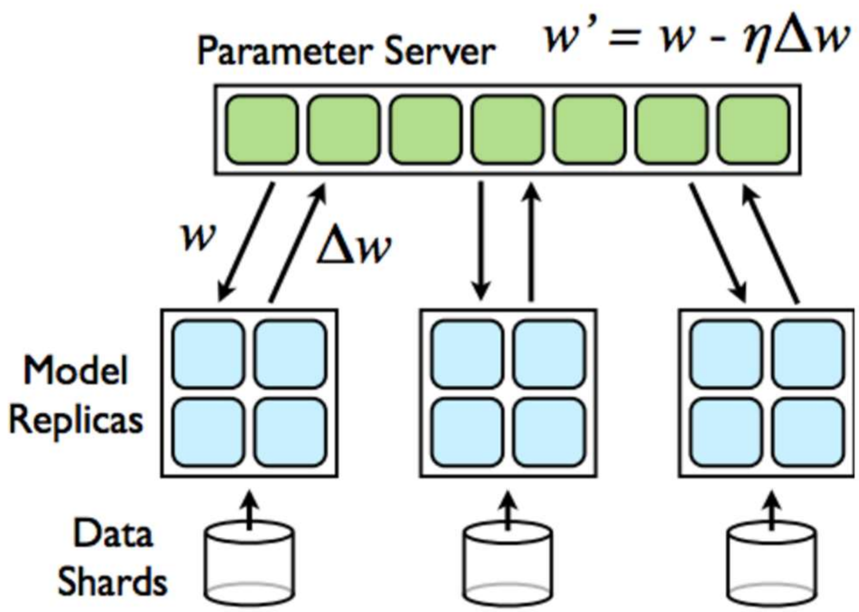
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

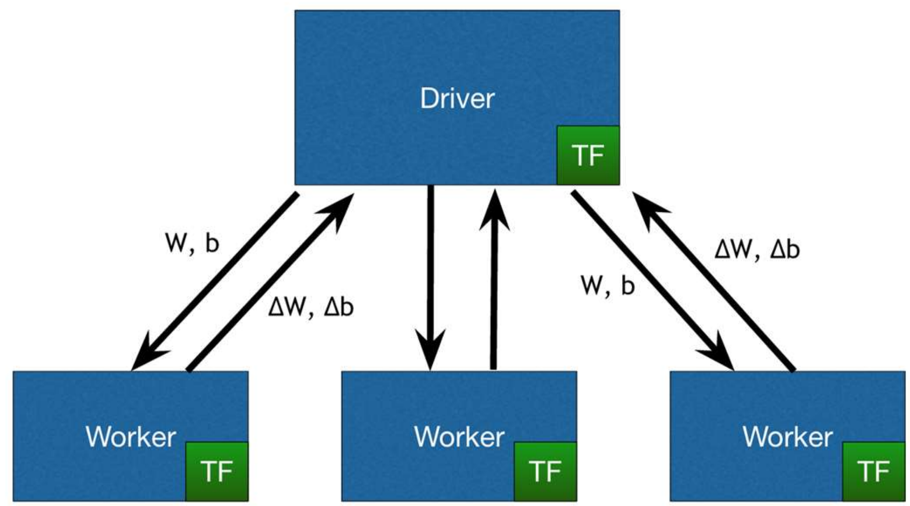
Ioffe (2015) *Batch normalization: Accelerating deep network training by reducing internal covariate shift*





Parallelizing and distributing SGD
Hogwild!





Downpour SGD

Downpour SGD is an asynchronous variant of SGD that was used by Dean et al. [4] in their DistBelief framework (predecessor to TensorFlow) at Google. It runs multiple replicas of a model in parallel on subsets of the training data. These models send their updates to a parameter server, which is split across many machines. Each machine is responsible for storing and updating a fraction of the model's parameters. However, as replicas don't communicate with each other e.g. by sharing weights or updates, their parameters are continuously at risk of diverging, hindering convergence.