Theoretical foundations of neural network architectures (MLP) and their optimisation

Ivan Nasonov
Telegram: @naconov
GitHub: NasonovIvan

5 November, 2023

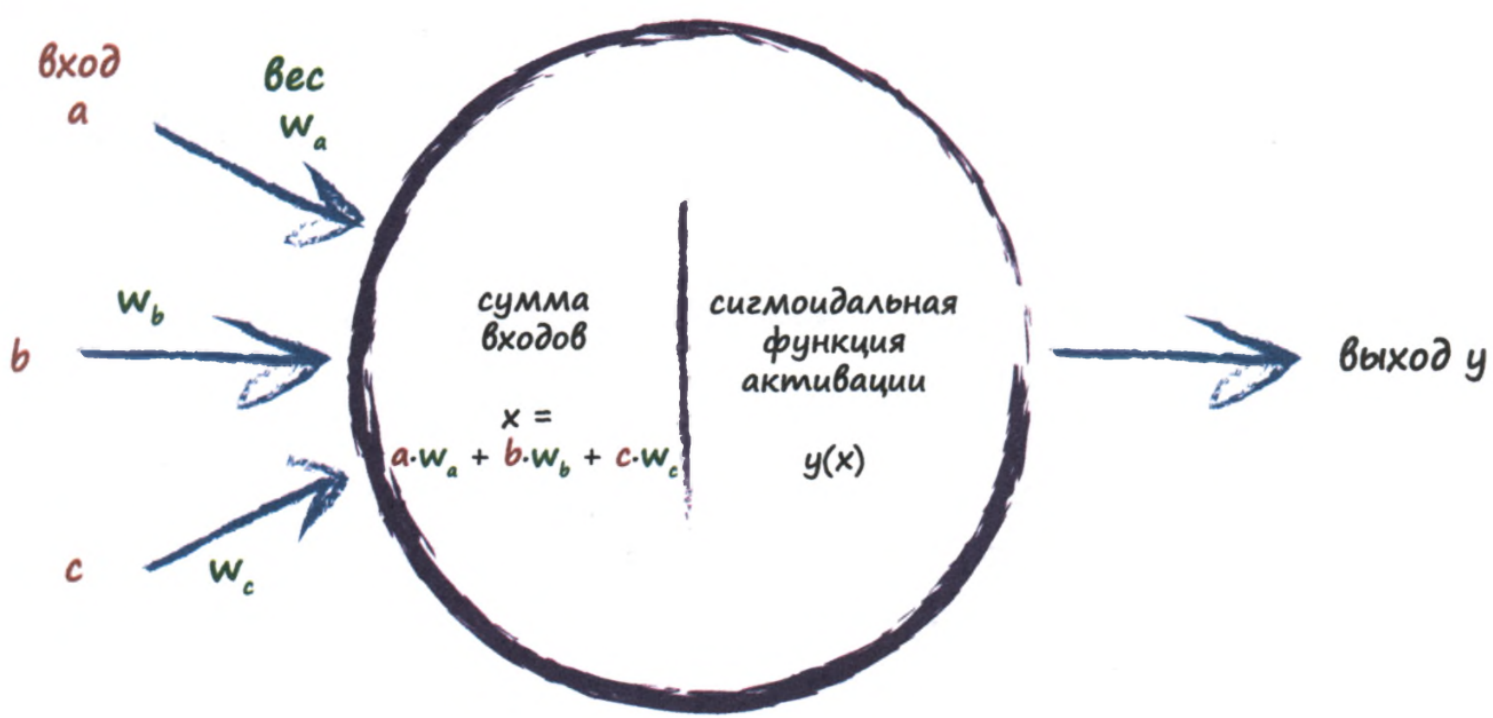
Постановка задачи

Мы будем работать с алгоритмом, который будет принимать сигнал и выдавать рассчитанный ответ.
При этом хотим, чтобы ответ совпадал с ожидающим истинным значением.

Source - Создаем нейронную сеть. Тарик Рашид.

Perceptron

Схема работы одного нейрона с 3 входными связями и 1 выходом.



Source - Создаем нейронную сеть. Тарик Рашид.

Переход к матричному представлению

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5)+(2*7) & (1*6)+(2*8) \\ (3*5)+(4*7) & (3*6)+(4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

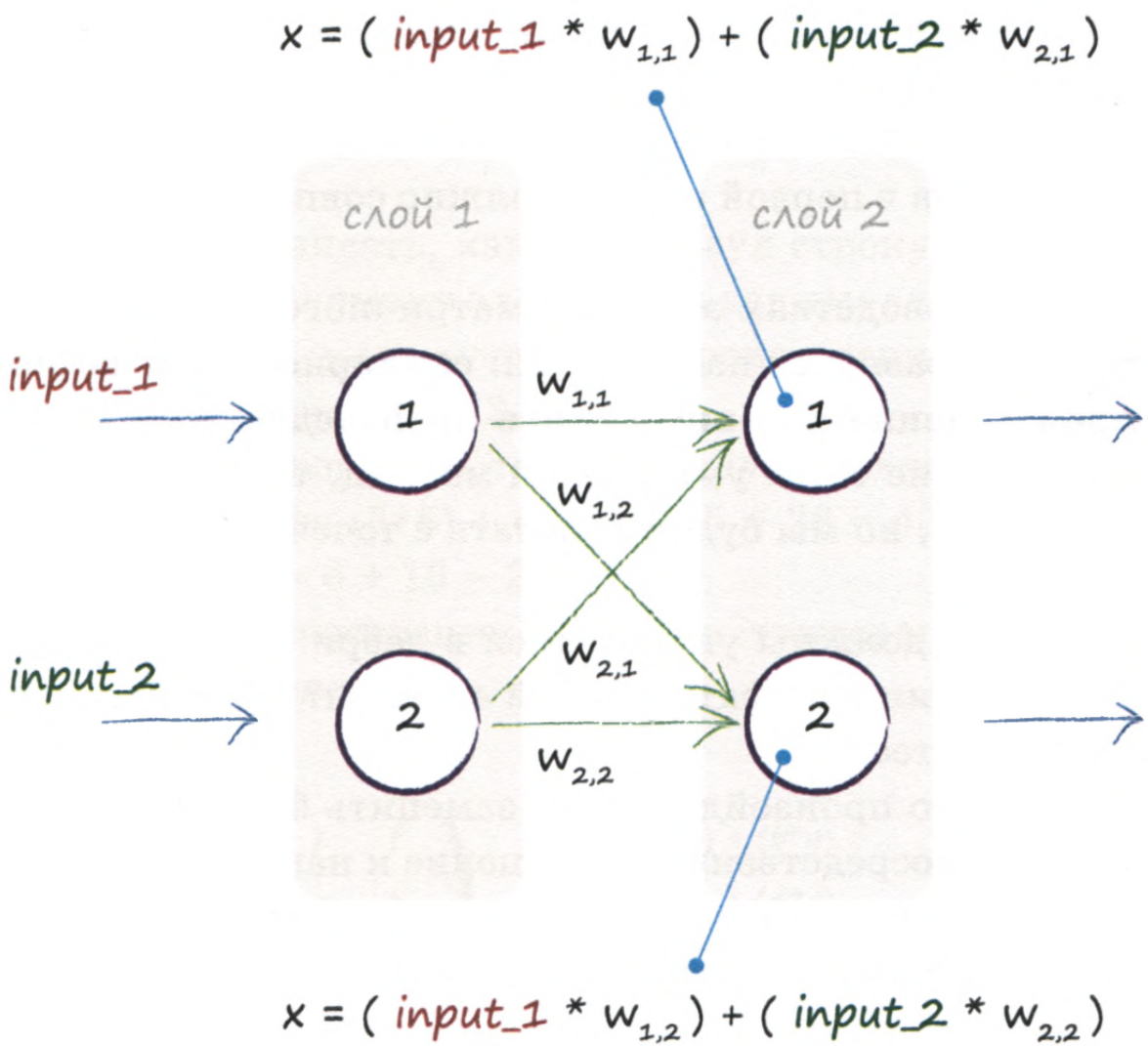
$$\begin{pmatrix} a & b & .. \\ c & d & .. \end{pmatrix} \begin{pmatrix} e & f \\ g & h \\ .. & .. \end{pmatrix} = \begin{pmatrix} (a*e)+(b*g)+... & (a*f)+(b*h)+... \\ (c*e)+(d*g)+... & (c*f)+(d*h)+... \end{pmatrix}$$

$$= \begin{pmatrix} ae+bg+... & af+bh+... \\ ce+dg+... & cf+dh+... \end{pmatrix}$$

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input\_1 \\ input\_2 \end{pmatrix} = \begin{pmatrix} (input\_1 * w_{1,1}) + (input\_2 * w_{2,1}) \\ (input\_1 * w_{1,2}) + (input\_2 * w_{2,2}) \end{pmatrix}$$

Source - Создаем нейронную сеть. Тарик Рашид.

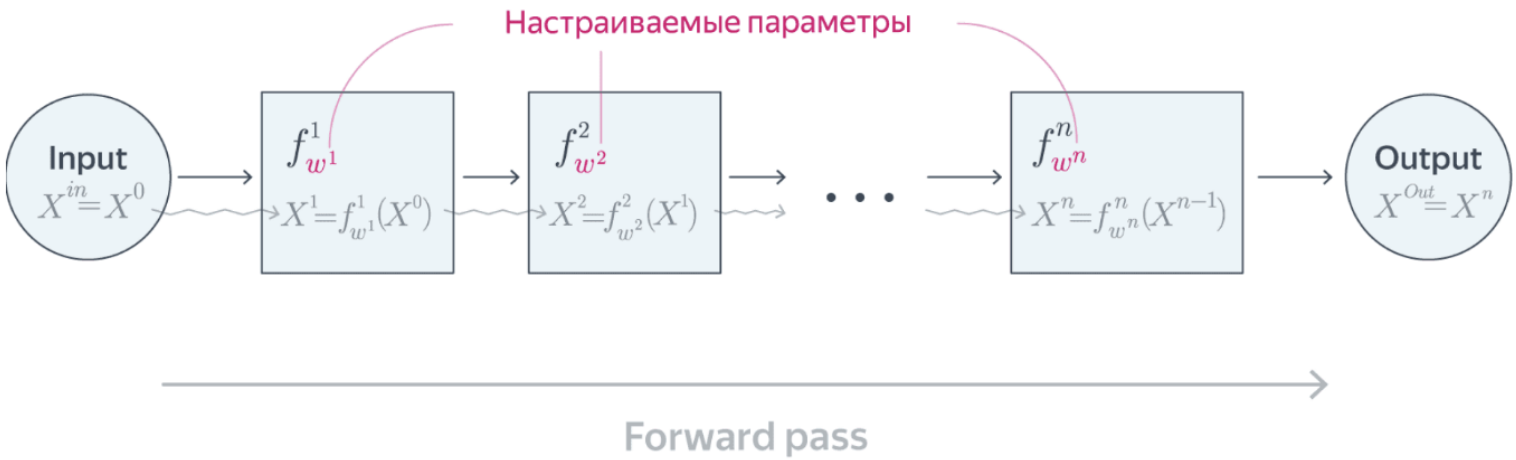
Пример расчета матрицы в сети



Source - Создаем нейронную сеть. Тарик Рашид.

Forward Propagation

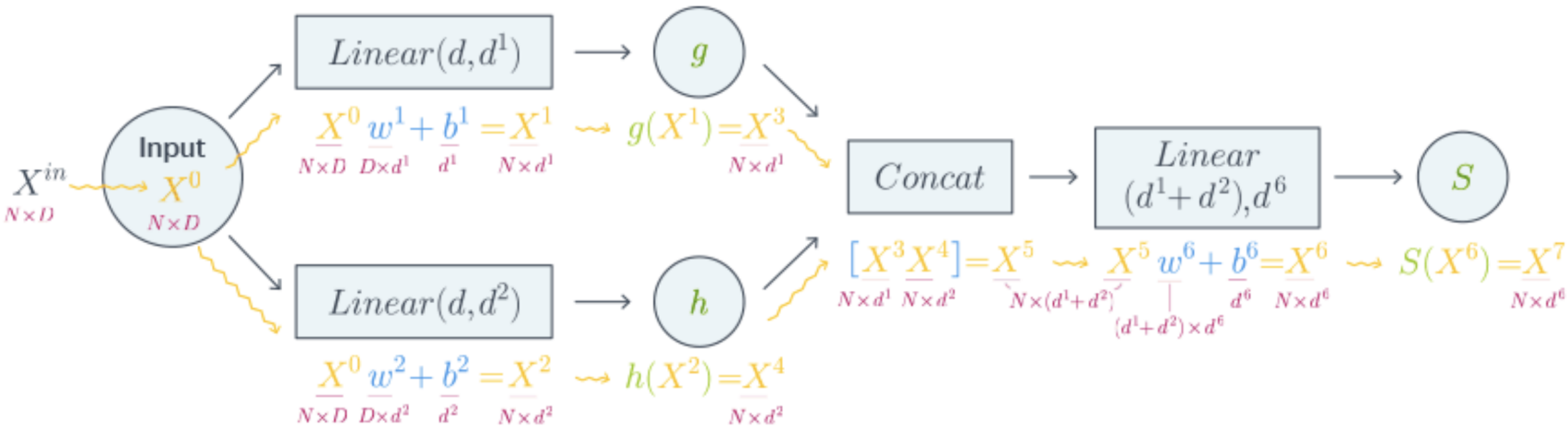
Применение нейронной сети к данным (вычисление выхода по заданному входу) часто называют прямым проходом, или же forward propagation (forward pass). На этом этапе происходит преобразование исходного представления данных в целевое и последовательно строятся промежуточные (внутренние) представления данных — результаты применения слоёв к предыдущим представлениям. Именно поэтому проход называют прямым.



Source - Яндекс учебник ML

Forward Propagation Example

Посмотрим, что происходит с размерностями, если на вход подаётся матрица $N \times D$:

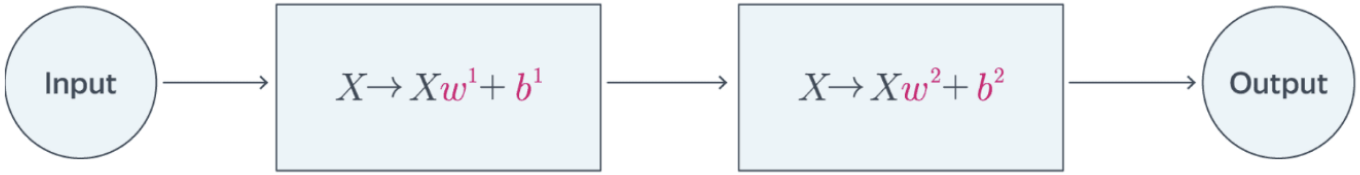


Source - Яндекс учебник ML

Why do we have activation functions?

Казалось бы, можно последовательно выстраивать лишь линейные слои, но так не делают: после каждого линейного слоя обязательно вставляют функцию активации. Но зачем? Рассмотрим нейронную сеть из двух линейных слоёв.

Линейная комбинация линейных отображений есть линейное отображение, то есть два последовательных линейных слоя эквивалентны одному линейному слою. Добавление функций активации после линейного слоя позволяет получить нелинейное преобразование, и подобной проблемы уже не возникает.

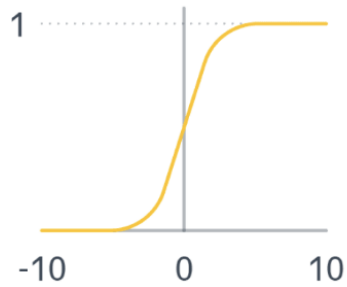


$$\hat{y} = X^{out} = X^1 W^2 + b^2 = (X^0 W^1 + b^1) W^2 + b^2 =$$

$$= X^0 W^1 W^2 + b^1 W^2 + b^2 = X^0 \widetilde{W} + \tilde{b}$$

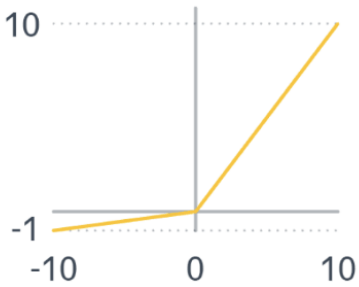
Source - Яндекс учебник ML

Activation Functions



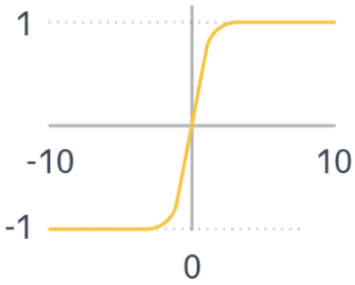
Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Leaky ReLU

$$max(0.1x, x)$$

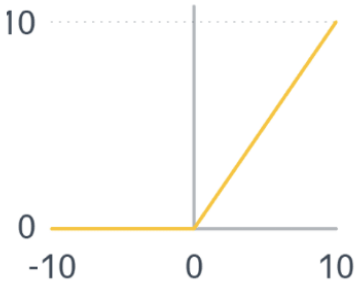


tanh

$$tanh(x)$$

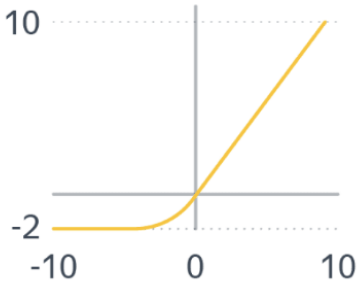
Maxout

$$max(\omega_1^T x + b_1, \omega_2^T x + b_2)$$



ReLU

$$max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Source - Яндекс учебник ML

Backpropagation Formulas

$$Z = X \cdot W = \sum_i^N (x_i \cdot w_i),$$

где $W = [w_1, w_2, ..., w_N]$ и $X = [1, x_1, x_2, ..., x_N]$

$$W = W - \eta \cdot \frac{1}{N} \cdot \frac{\partial L}{\partial W}$$

Activation Functions

- $SoftMax(z) = \frac{exp(z_i)}{\sum_i^N exp(z_i)}$
- $\sigma(z) = \frac{1}{1+exp(-z)}$
- $ReLU(z) = max(0, z)$

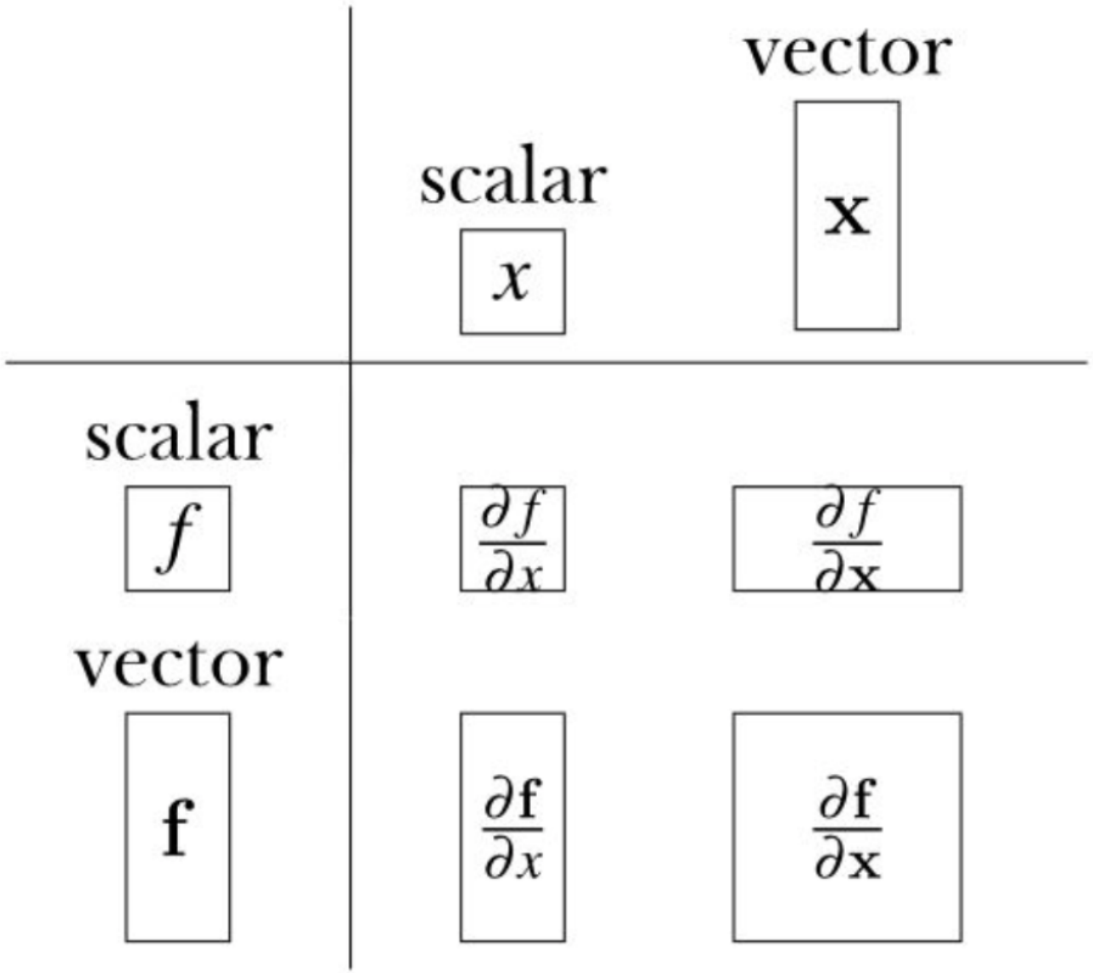
Loss functions

- $CrossEntropy = -\sum_i^N (y_i \cdot \ln(p_i))$
- $LogLoss = -(1-y)\ln(1-p) - y\ln(p)$
- $MSE = \frac{1}{N}\sum_i^N (y_i - p_i)^2$

Necessary Formulas

- $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial z}\frac{\partial z}{\partial w}$
- $f(x) = \frac{g(x)}{h(x)}$, then $f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h^2(x)}$

Backpropagation Derivatives Shapes



Source - girafe.ai

Backpropagation. SoftMax + Cross-Entropy

$$Z = XW$$

$$s_i(z_i) = \frac{exp(z_i)}{\sum_i^N exp(z_i)}$$

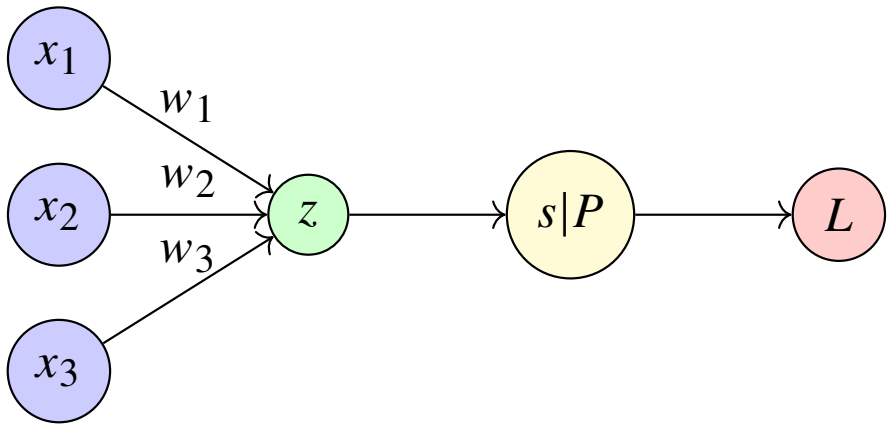
$$L(y_i, s_i) = -\sum_i^N (y_i \ln(s_i))$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} \rightarrow \begin{pmatrix} \frac{exp(z_1)}{\sum_i^3 exp(z_i)} \\ \frac{exp(z_2)}{\sum_i^3 exp(z_i)} \\ \frac{exp(z_3)}{\sum_i^3 exp(z_i)} \end{pmatrix} \rightarrow \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

Need: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial s} \frac{\partial s}{\partial z} \frac{\partial z}{\partial w}$

Derivatives:

- *Scalar / Vector* $\frac{\partial L}{\partial s} = -\begin{pmatrix} y_1/s_1 \\ y_2/s_2 \\ \vdots \\ y_N/s_N \end{pmatrix}$
- $\frac{\partial Z}{\partial W} = X^T$



Shapes:

- X [N, features]
- W [features, classes]
- Z [N, classes]

Backpropagation. SoftMax + Cross-Entropy

$$\frac{\partial s}{\partial z} \rightarrow \textit{Vector / Vector}$$

$$\frac{\partial s_1}{\partial z_1} = \frac{e^{z1}\,(e^{z1}+e^{z2}+e^{z3})-e^{z1}\,e^{z1}}{(e^{z1}+e^{z2}+e^{z3})\,(e^{z1}+e^{z2}+e^{z3})} = \frac{e^{z1}}{(e^{z1}+e^{z2}+e^{z3})} \cdot \frac{e^{z1}+e^{z2}+e^{z3}-e^{z1}}{(e^{z1}+e^{z2}+e^{z3})} = s_1(1-s_1)$$

$$\frac{\partial s_1}{\partial z_2} = \frac{0\cdot(e^{z1}+e^{z2}+e^{z3})-e^{z1}\,e^{z2}}{(e^{z1}+e^{z2}+e^{z3})\,(e^{z1}+e^{z2}+e^{z3})} = -\frac{e^{z1}}{(e^{z1}+e^{z2}+e^{z3})} \cdot \frac{e^{z2}}{(e^{z1}+e^{z2}+e^{z3})} = -s_1 s_2$$

$$\frac{\partial s}{\partial z} = J = \begin{pmatrix} s_1(1-s_1) & -s_1 s_2 & -s_1 s_3 & \dots & -s_1 s_N \\ -s_1 s_2 & s_2(1-s_2) & -s_2 s_3 & \dots & -s_2 s_N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -s_1 s_N & -s_2 s_N & -s_3 s_N & \dots & s_N(1-s_N) \end{pmatrix}$$

Backpropagation. SoftMax + Cross-Entropy

But let's see the magic...

$$\frac{\partial L}{\partial s} \cdot \frac{\partial s}{\partial z} =$$

$$-\begin{pmatrix} y_1/s_1 \\ y_2/s_2 \\ \vdots \\ y_N/s_N \end{pmatrix} \cdot \begin{pmatrix} s_1(1-s_1) & -s_1 s_2 & -s_1 s_3 & \dots & -s_1 s_N \\ -s_1 s_2 & s_2(1-s_2) & -s_2 s_3 & \dots & -s_2 s_N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -s_1 s_N & -s_2 s_N & -s_3 s_N & \dots & s_N(1-s_N) \end{pmatrix}$$

$$= -\begin{pmatrix} y_1 - s_1(y_1 + y_2 + \cdots + y_N) \\ y_2 - s_2(y_1 + y_2 + \cdots + y_N) \\ \vdots \\ y_N - s_N(y_1 + y_2 + \cdots + y_N) \end{pmatrix} = s - y$$

$$\frac{\partial L}{\partial w} = X^T(s - y)$$

Shapes: [features, N] $\times$ [N, classes]

Memes moments...

Это я показываю своему другу как его 1000+ строчек на C++ будут выглядеть в 10 строчках на Python

Backpropagation. Sigmoid + LogLoss

$$Z = XW$$
$$\sigma(Z) = \frac{1}{1+e^{-Z}}$$
$$L(y, \sigma) = -(1-y)\ln(1-\sigma) - y\ln(\sigma)$$

Need: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial w}$

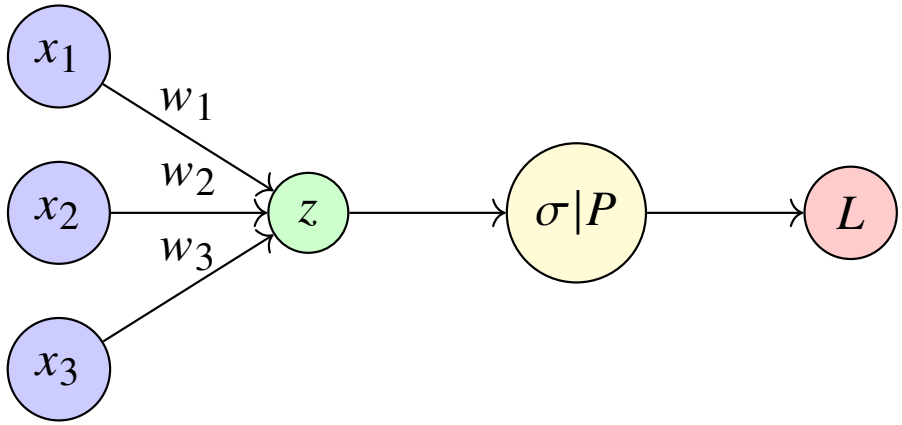
Derivatives:

- $\frac{\partial L}{\partial \sigma} = \frac{1-y}{1-\sigma} - \frac{y}{\sigma}$
- $\frac{\partial \sigma}{\partial Z} = \frac{e^{-Z}}{(1+e^{-Z})^2} = \frac{1}{1+e^{-Z}}\frac{e^{-Z}}{1+e^{-Z}} = \frac{1}{1+e^{-Z}}\frac{(1+e^{-Z})-1}{1+e^{-Z}} = \frac{1}{1+e^{-Z}}(1 - \frac{1}{1+e^{-Z}}) = \sigma(1-\sigma)$
- $\frac{\partial Z}{\partial W} = X^T$

Shapes:

- X [N, features]
- W [features, classes=1]
- Z [N, classes=1]

$$W = W - \eta \cdot \frac{1}{N} \cdot \frac{\partial L}{\partial W}$$



Answer:

$$\frac{\partial L}{\partial W} = (\frac{1-y}{1-\sigma} - \frac{y}{\sigma})\sigma(1-\sigma)X^T$$

$$\frac{\partial L}{\partial W} = (\sigma - y)X^T$$

Derivatives in NN. Graph

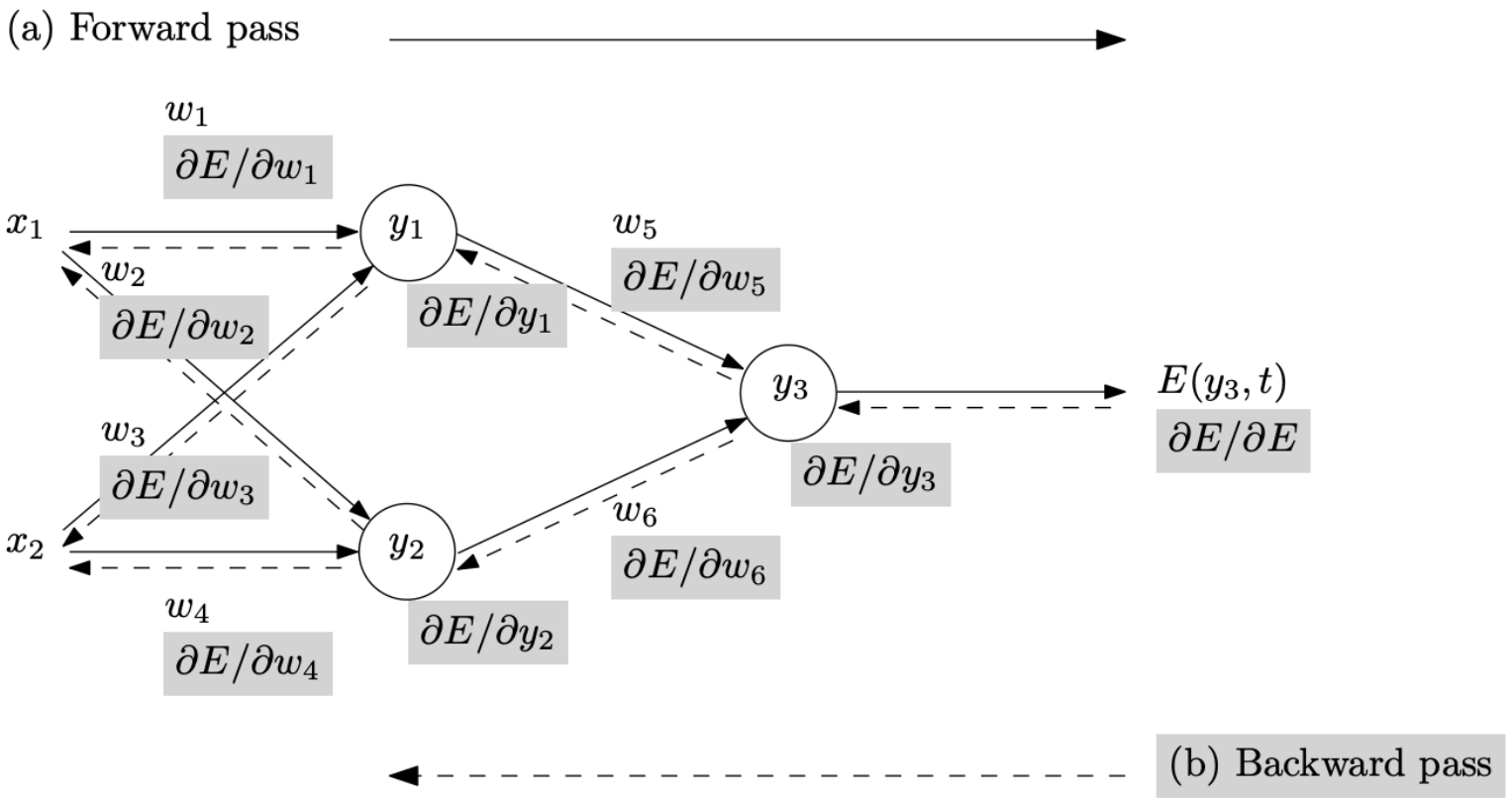


Рис. 1: Compute derivatives in NN

Source - Automatic Differentiation in Machine Learning: a Survey. Alexey Andreyevich Radul et al.

Metrics

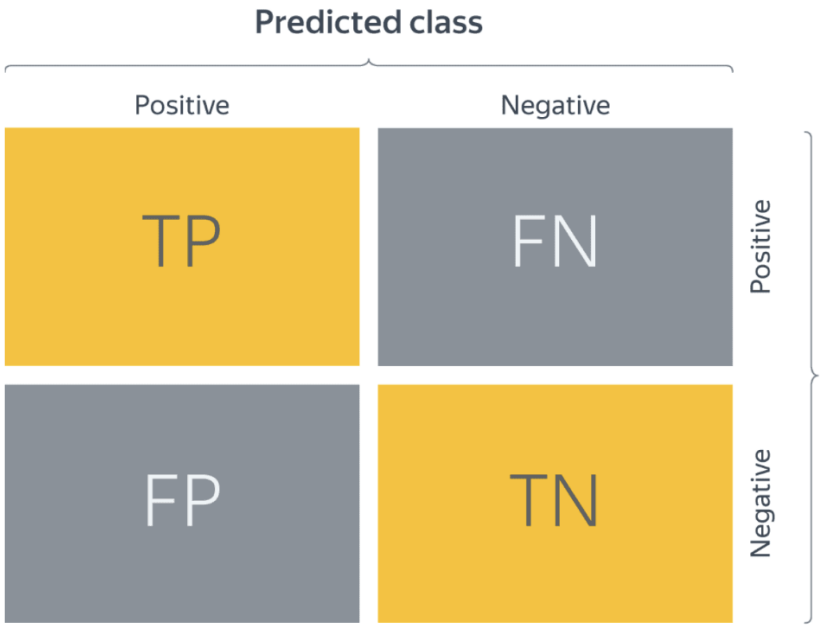
Classification

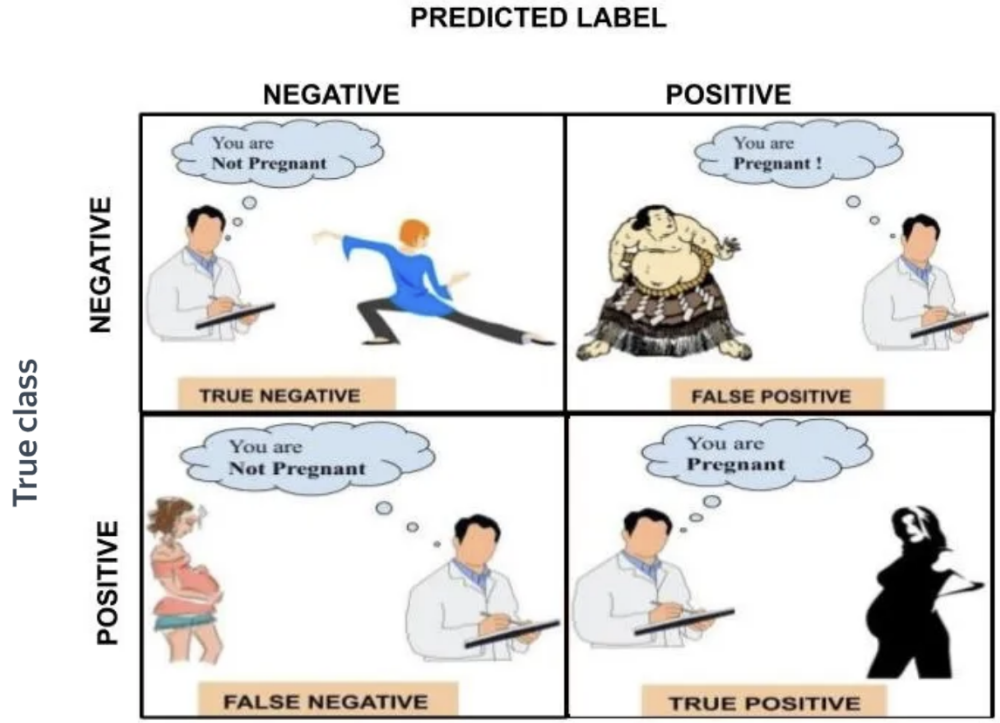
- Confusion Matrix
- Accuracy
- Precision
- Recall
- F1-score
- AUC-ROC

Regression

- MSE
- RMSE
- R^2
- MAE
- Huber Loss

Metrics. Classification





Source - Яндекс учебник ML, Medium

Metrics. Classification

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

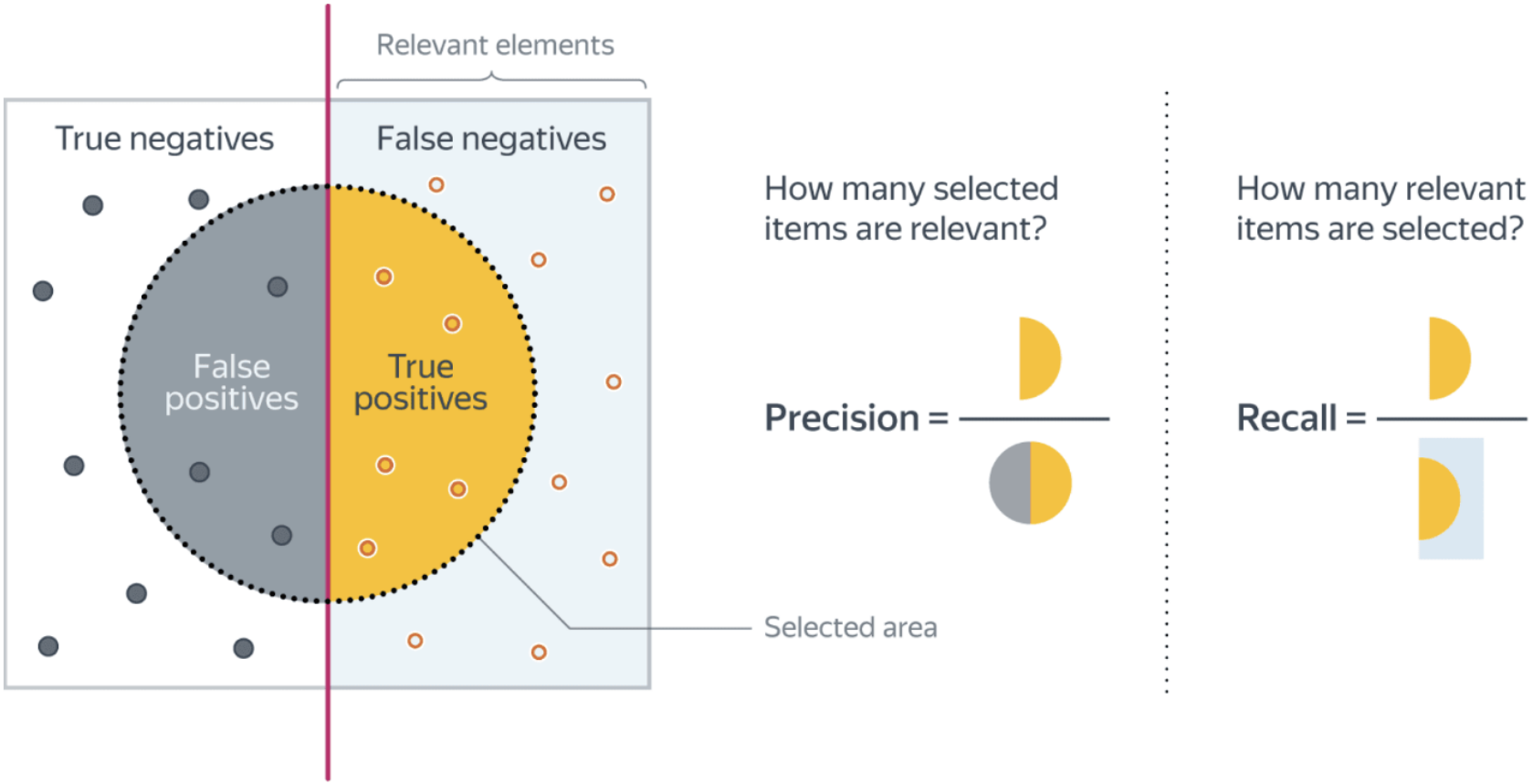
$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

- Precision – доля правильно предсказанных положительных объектов среди всех объектов, предсказанных положительным классом. Метрика показывает долю релевантных документов среди всех найденных классификатором.
- Recall – доля правильно найденных положительных объектов среди всех объектов положительного класса. Метрика показывает долю найденных документов из всех релевантных.

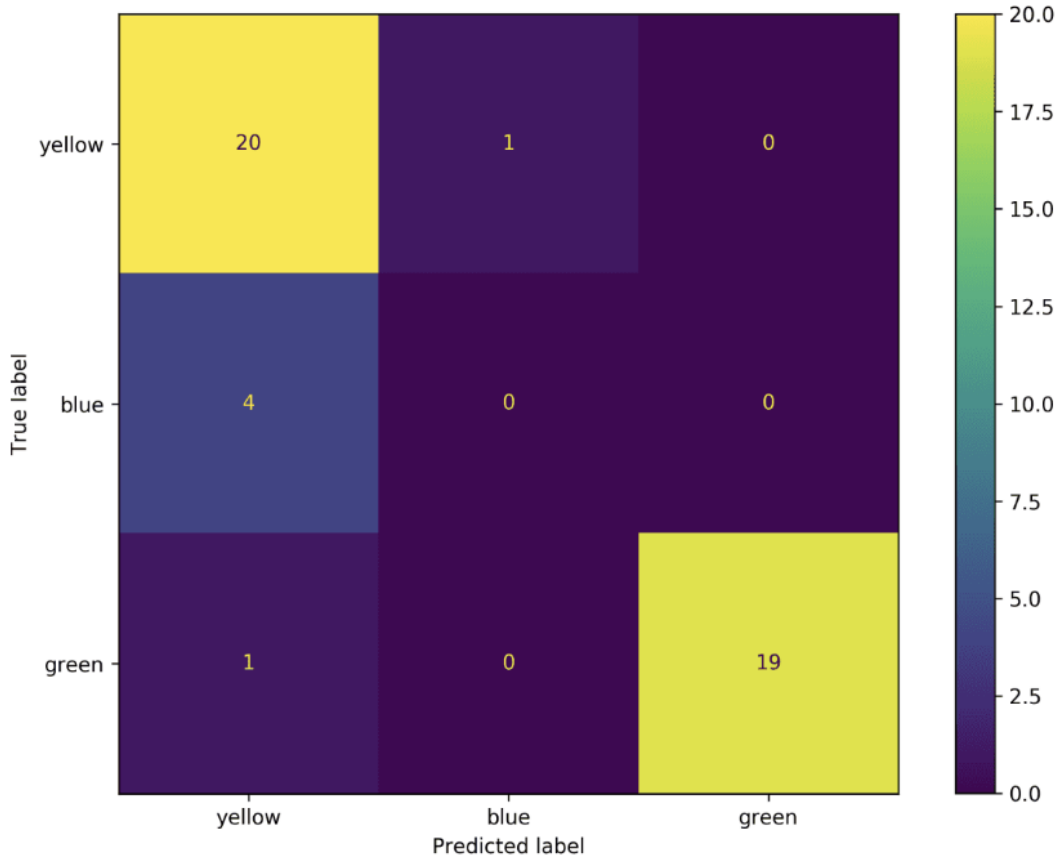
Source - Яндекс учебник ML

Metrics. Classification



Source - Яндекс учебник ML

Metrics. Classification



Micro Average

$$P = \frac{\frac{1}{3}(20+0+19)}{\frac{1}{3}(20+0+19)+\frac{1}{3}(5+1+0)} = 0.87$$

Macro Average

$$P = \frac{1}{3}\left(\frac{20}{20+5}+\frac{0}{0+1}+\frac{19}{19+0}\right) = 0.6$$

Source - Яндекс учебник ML

Metrics. Classification

В случае пары Precision-Recall существует популярный способ скомпоновать их в одну метрику - взять их среднее гармоническое.

$$F1 - score = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}} = 2\frac{Recall \cdot Precision}{Recall + Precision}$$

Если одна из метрик приоритетнее, то можно воспользоваться F_β

$$F_\beta = (\beta^2 + 1)\frac{Recall \cdot Precision}{Recall + \beta^2 Precision}$$

Source - Яндекс учебник ML

Metrics. Classification

TPR (true positive rate) – это полнота, доля положительных объектов, правильно предсказанных положительными:

$$TPR = \frac{TP}{TP + FN}$$

FPR (false positive rate) – это доля отрицательных объектов, неправильно предсказанных положительными:

$$FPR = \frac{FP}{FP + TN}$$

Area Under Curve (AUC) + Receiver Operating Characteristics (ROC) = AUC-ROC

$\Rightarrow$ **Press here to see the curve**

Source - Яндекс учебник ML

Metrics. Regression

- Mean Squared Error (MSE) = $\frac{1}{N}\sum_i^N (y_i - \hat{y}_i)^2$
- Root Mean Squared Error (RMSE) = $\sqrt{\frac{1}{N}\sum_i^N (y_i - \hat{y}_i)^2}$
- $R^2 = 1 - \frac{\sum_i^N (y_i - \hat{y}_i)^2}{\sum_i^N (y_i - \bar{y}_i)^2}$ – коэффициент детерминации
- Mean Absolute Error (MAE) = $\frac{1}{N}\sum_i^N |y_i - \hat{y}_i|$

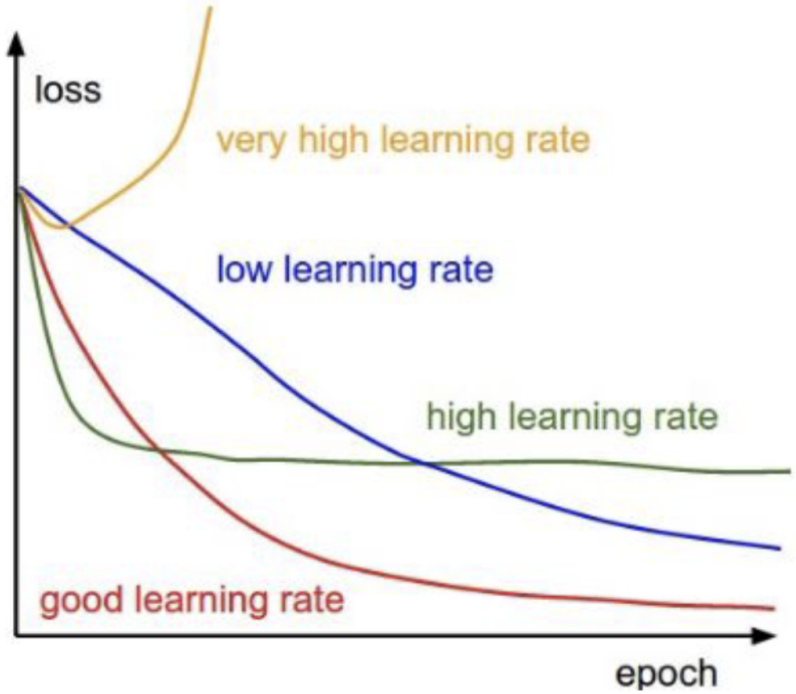
$$Huber\ Loss = \begin{cases} \frac{1}{2}x^2 & \text{for } |x| \le \delta \\ \delta(|x| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

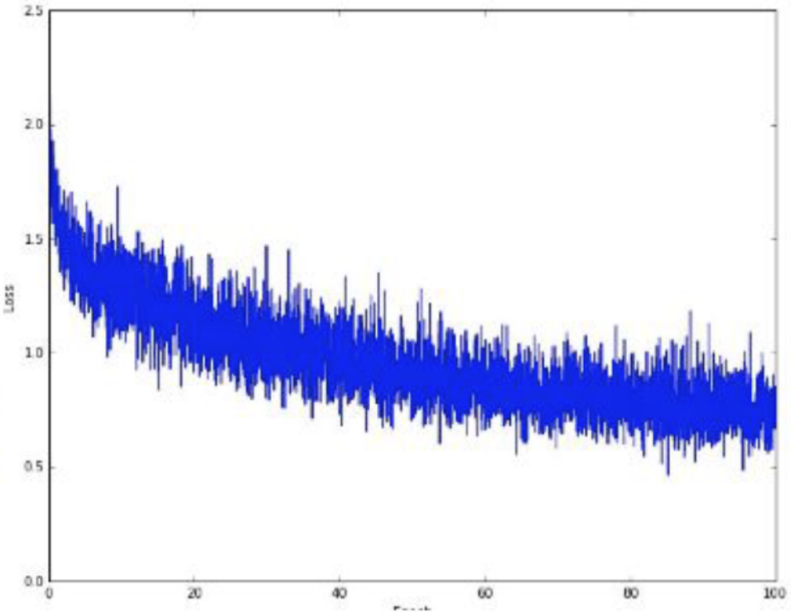
Source - Яндекс учебник ML

Optimizers

Stochastic gradient descent is used to optimize NN parameters.

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





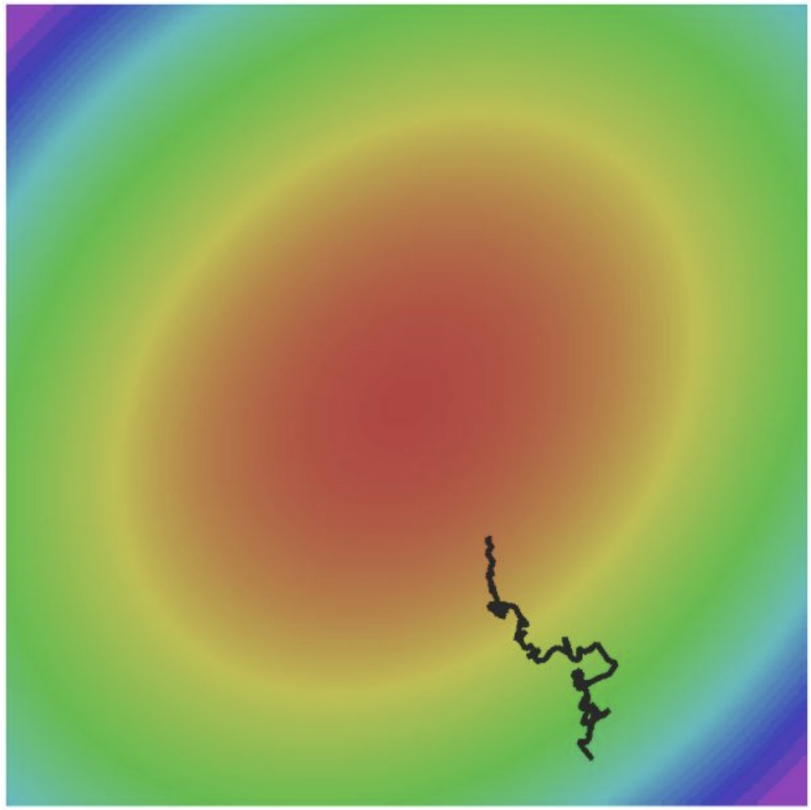
Source - girafe.ai

Problems with SGD

Our gradients come from minibatches so they can be noisy!

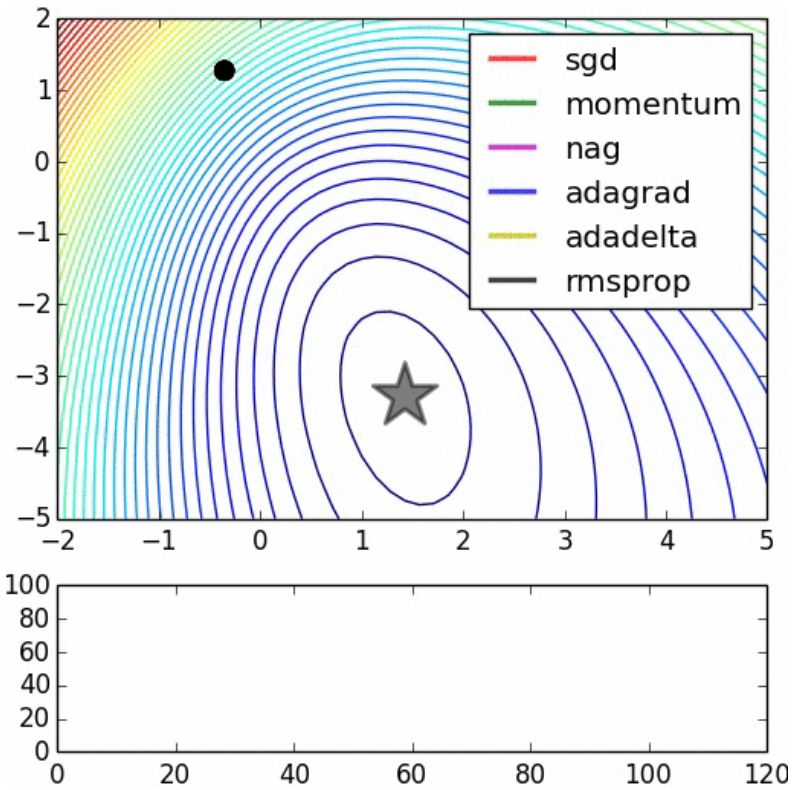
$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

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



Source - girafe.ai

Compare Optimizers



→ press here

Source - girafe.ai

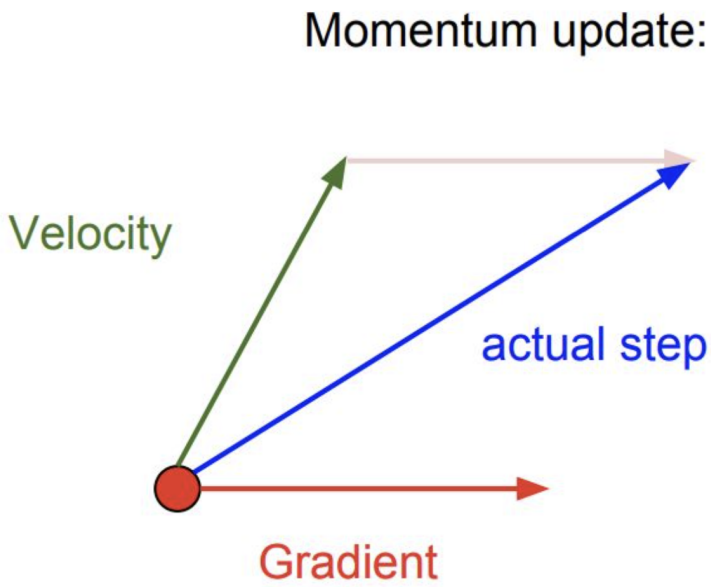
Momentum

Simple SGD

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

SGD with momentum

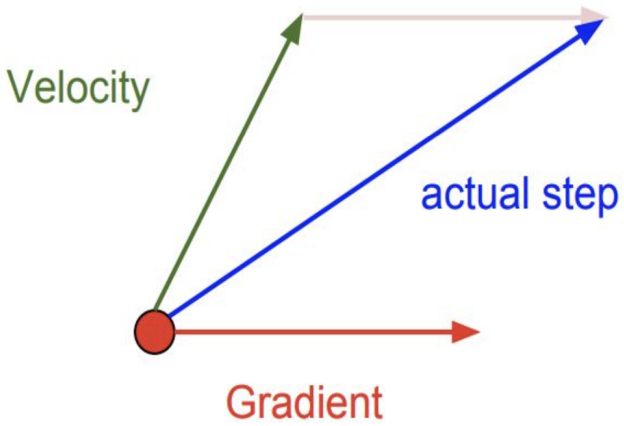
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$



Source - girafe.ai

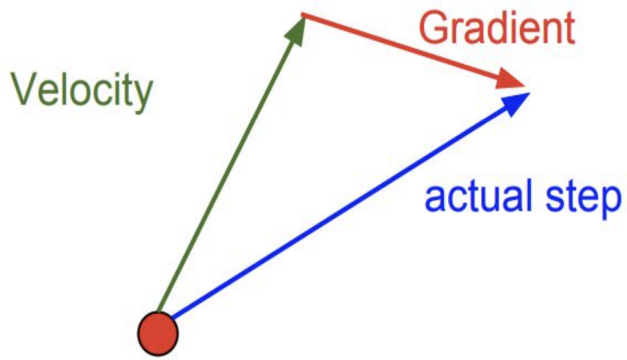
Nesterov Momentum

Momentum update:



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

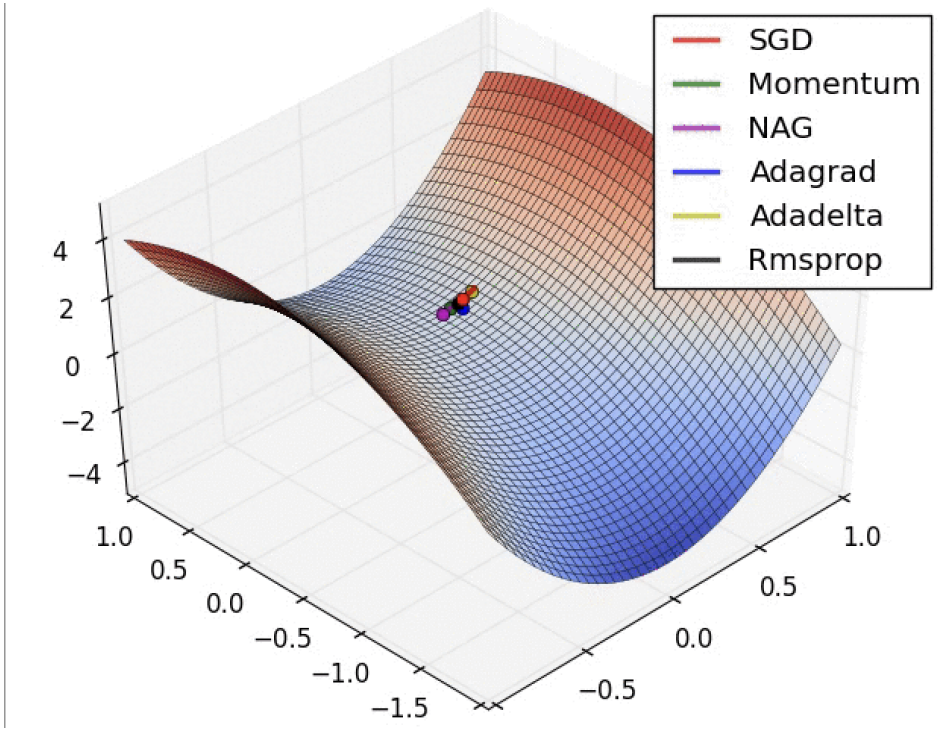
Nesterov Momentum



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

Source - girafe.ai

Compare Optimizers

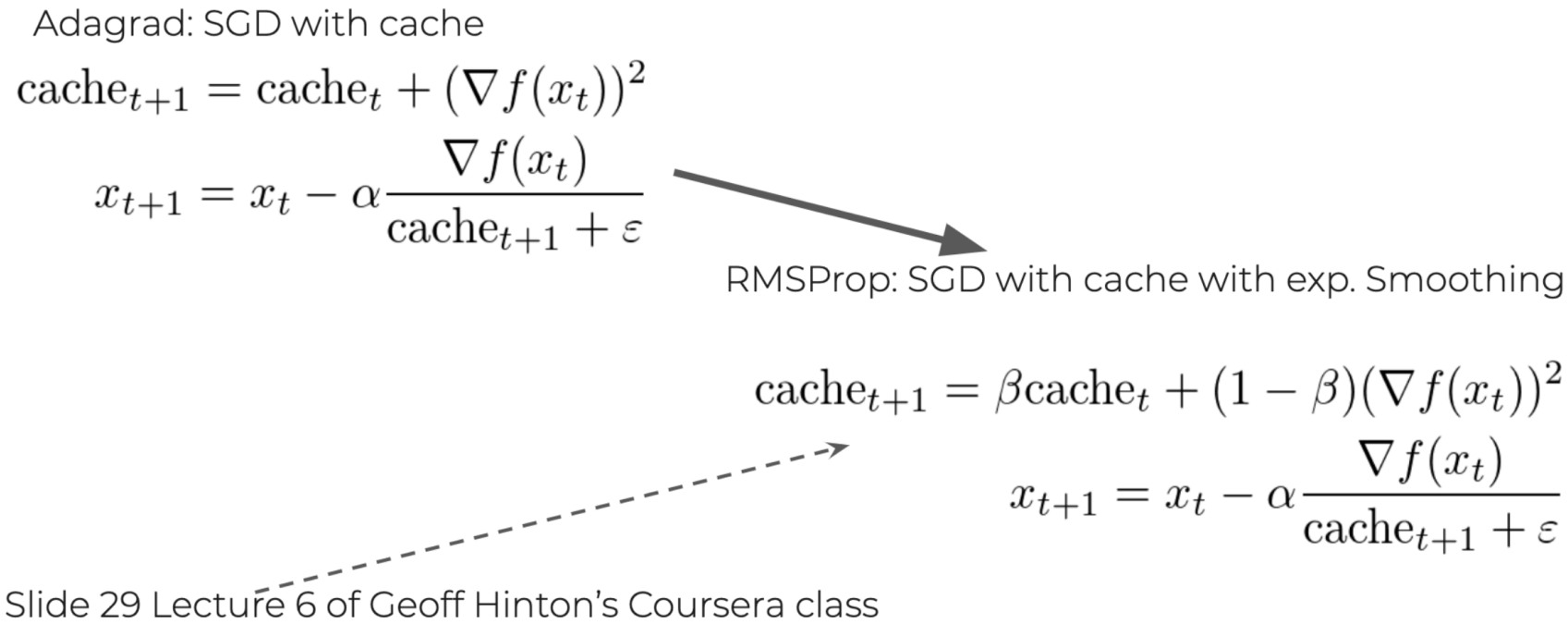


→ press here

Source - girafe.ai

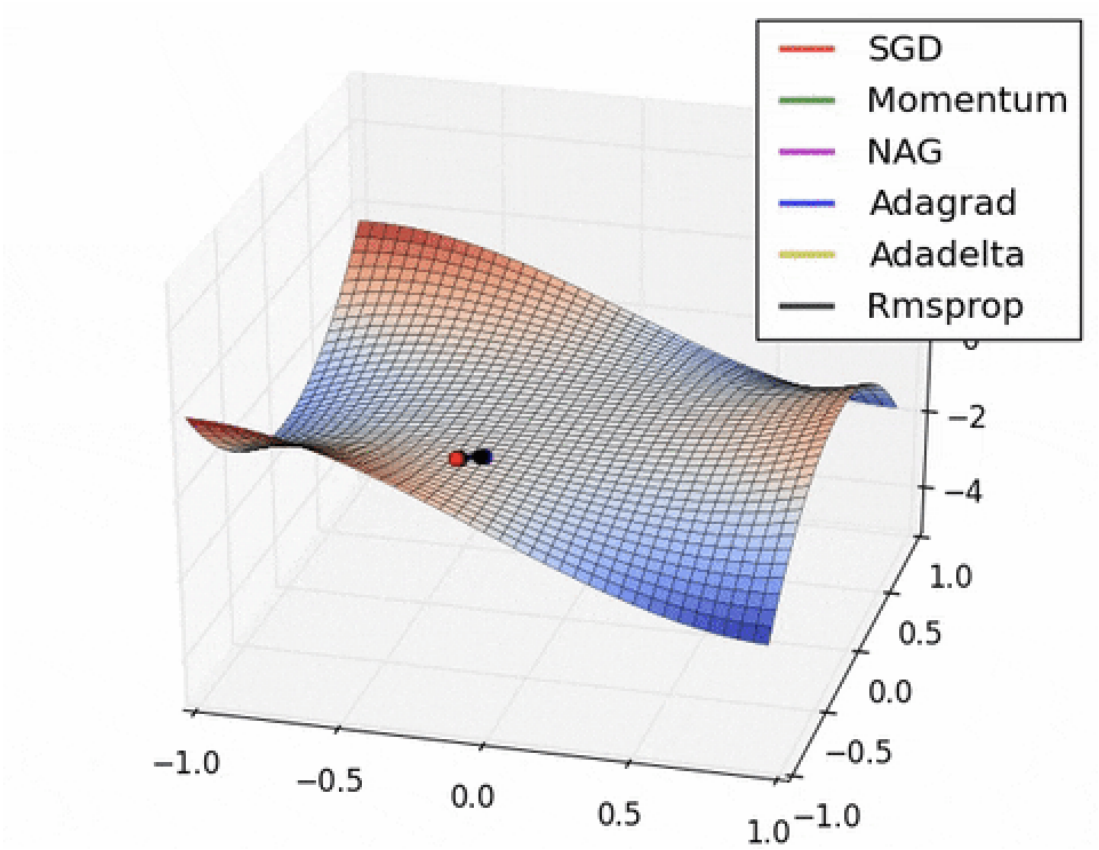
AdaGrad and RMSProp

We normalise gradients instead of accumulating them. But then we get a strictly monotonic cache. The solution is RMSProp.



Source - girafe.ai

Compare Optimizers



→ **press here**

Source - girafe.ai

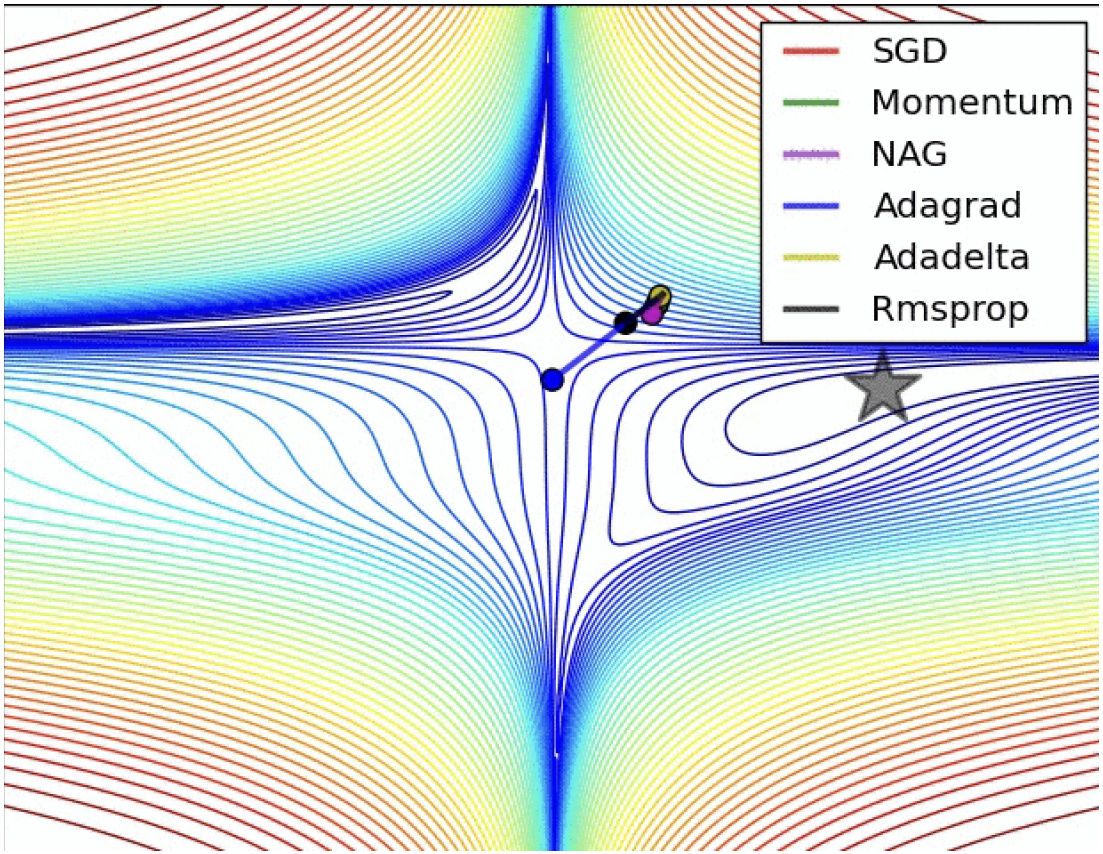
Adam

Let's combine the momentum idea and RMSProp normalization.

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

Source - girafe.ai

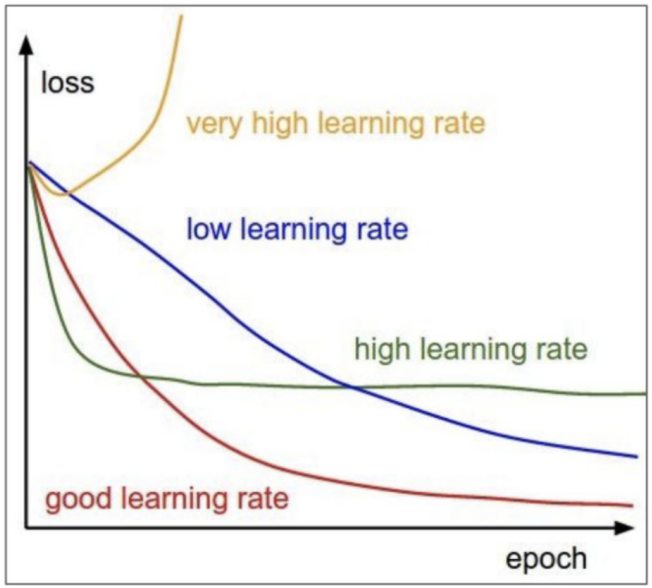
Compare Optimizers

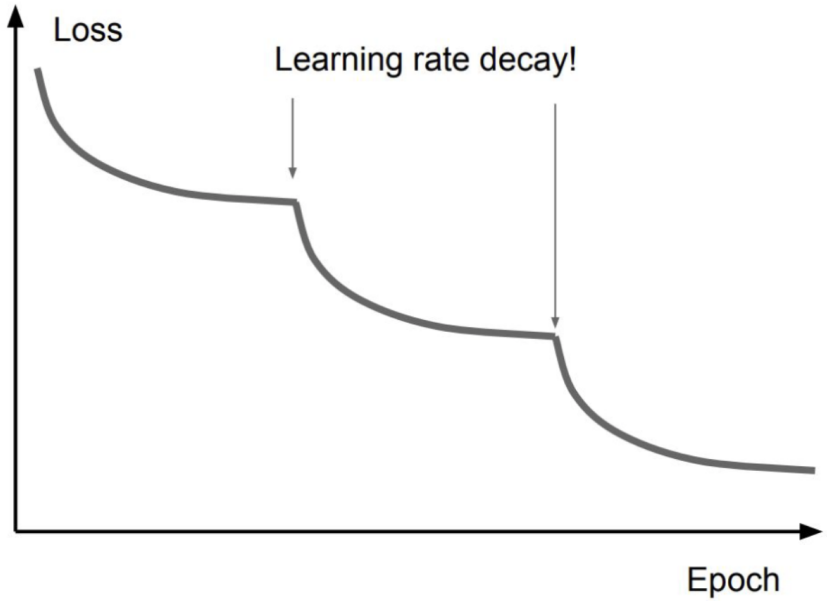


→ **press here**

Source - girafe.ai

Learning Rate Changing





Source - girafe.ai

Data Normalization

Standard normalization

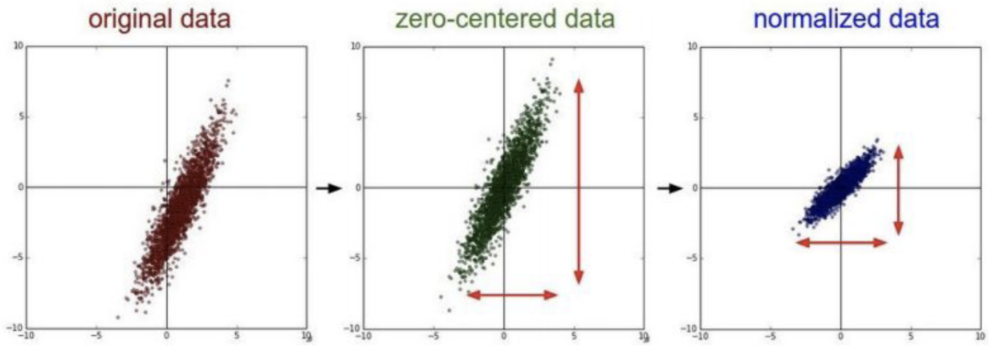
$$\hat{x} = \frac{x - \mu}{\sigma}$$

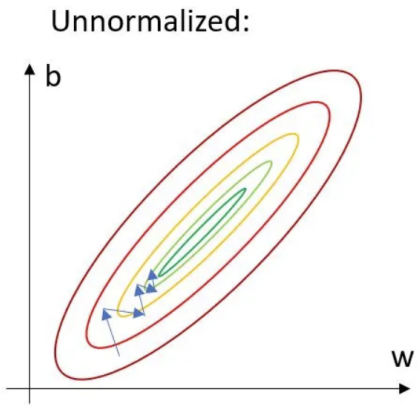
where

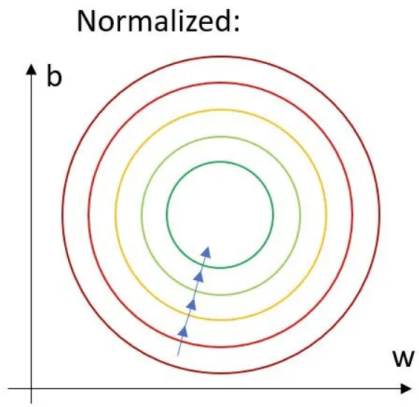
$\mu = \frac{1}{N}\sum_i^N x_i; \sigma = \sqrt{\frac{1}{N}\sum_i^N (x_i - \mu)^2}$

MinMax normalization

$$\hat{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

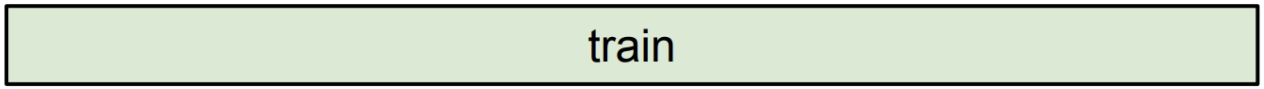






Data. Cross-Validation

Idea #1: Choose hyperparameters
that work best on the **training data**

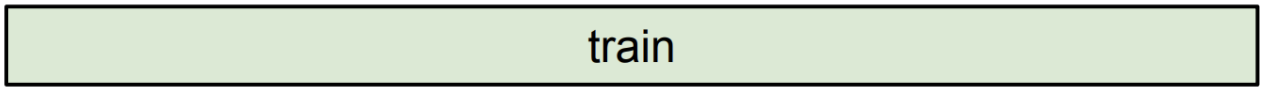


Idea #2: choose hyperparameters
that work best on **test** data

Source - Stanford CS231n

Data. Cross-Validation

Idea #1: Choose hyperparameters that work best on the **training data**



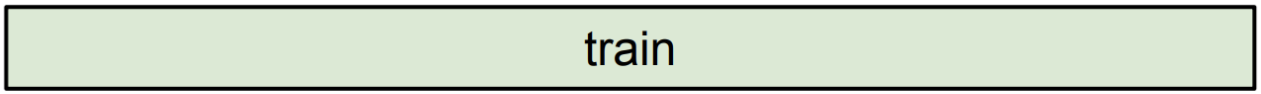
Idea #2: choose hyperparameters that work best on **test** data

Idea #3: Split data into **train**, **val**; choose hyperparameters on val and evaluate on test

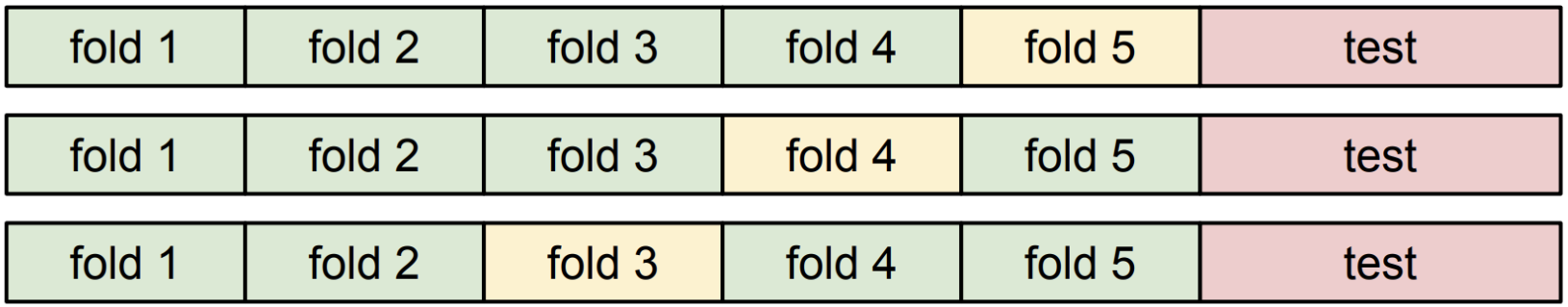
Better!

Source - Stanford CS231n

Data. Cross-Validation



Idea #4: **Cross-Validation**: Split data into **folds**,
try each fold as validation and average the results

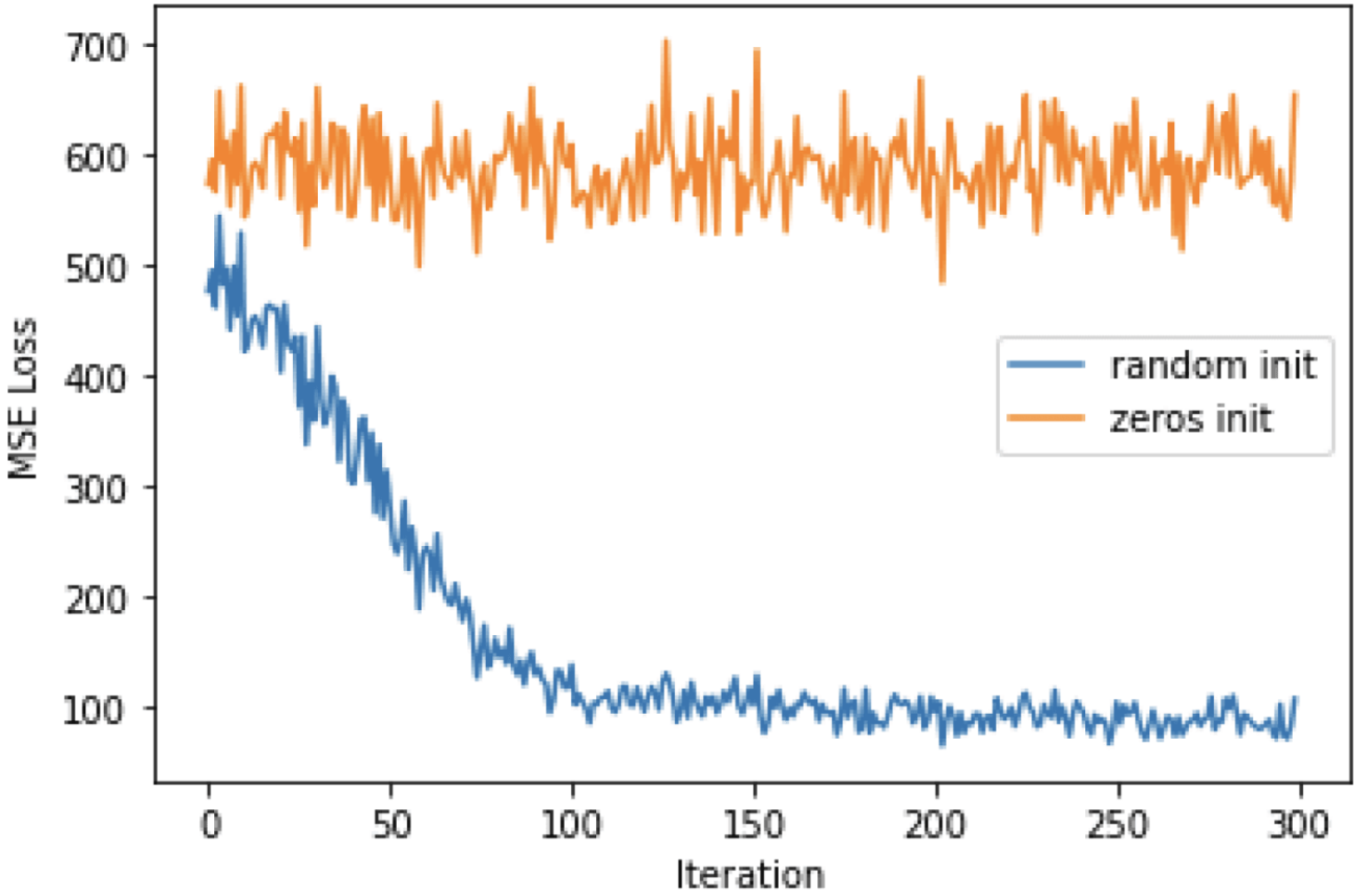


Useful for small datasets, but not used too frequently in deep learning

Source - Stanford CS231n

Weights Initialization

Initialisation with zeros is forbidden (but there are exceptions).



Weights Calibrated Random Numbers Initialization

$\mu(w) = 0, \mu(x) = 0$

$$\text{Var}(s) = \text{Var}(\sum_{i}^{n} w_i x_i)$$

$$= \sum_{i}^{n} \text{Var}(w_i x_i)$$

$$= \sum_{i}^{n} [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i)\text{Var}(w_i)$$

$$= \sum_{i}^{n} \text{Var}(x_i)\text{Var}(w_i)$$

$$= (n\text{Var}(w))\ \text{Var}(x)$$

Weights Calibrated Random Numbers Initialization

Следовательно, дисперсия результата линейно зависит от дисперсии входных данных с коэффициентом $n_{in}Var(w)$.

Увеличение дисперсии промежуточных представлений с каждым новым преобразованием может вызвать численные ошибки или насыщение функций активации (таких как $tanh$ и $sigmoid$)

Снижение дисперсии может привести к почти нулевым промежуточным представлениям (плюс «линейному» поведению $tanh$ и $sigmoid$ в непосредственной близости от нуля)

Поэтому для начальной инициализации весов имеет смысл использовать распределение, дисперсия которого позволила бы сохранить дисперсию результата.

$$\forall w_i \sim \mathbb{N}(0, \frac{1}{n_{in}})$$

Или проще: $\forall w_i, Var(w_i) = \frac{1}{n_{in}}$

Source - Яндекс учебник ML

Xavier and Normalized Xavier Weights Initialization

В прошлой инициализации если мы хотим, чтобы сохранялись дисперсии и промежуточных представлений, и градиентов, у нас возникают сразу два ограничения:

$$Var(w_i) = \frac{1}{n_{in}} \text{ and } Var(w_i) = \frac{1}{n_{out}}$$

Understanding the difficulty of training deep feedforward neural networks. Xavier Glorot and Yoshua Bengio.

$$\forall i, Var(w_i) = \frac{2}{n_{in} + n_{out}}$$

Данная инициализация хорошо подходит именно для $tanh$, т.к. в выводе явно учитывается симметричность функции активации относительно нуля.

Normalized Xavier Initialization:

$$\forall i, Var(w_i) \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right]$$

Source - Яндекс учебник ML

Xavier and Normalized Xavier Weights Initialization

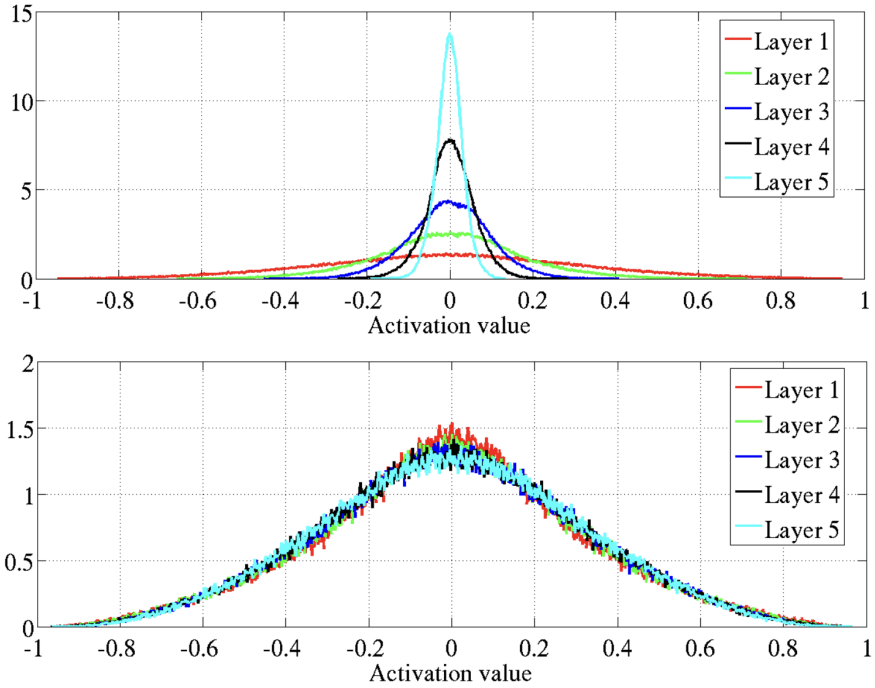


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

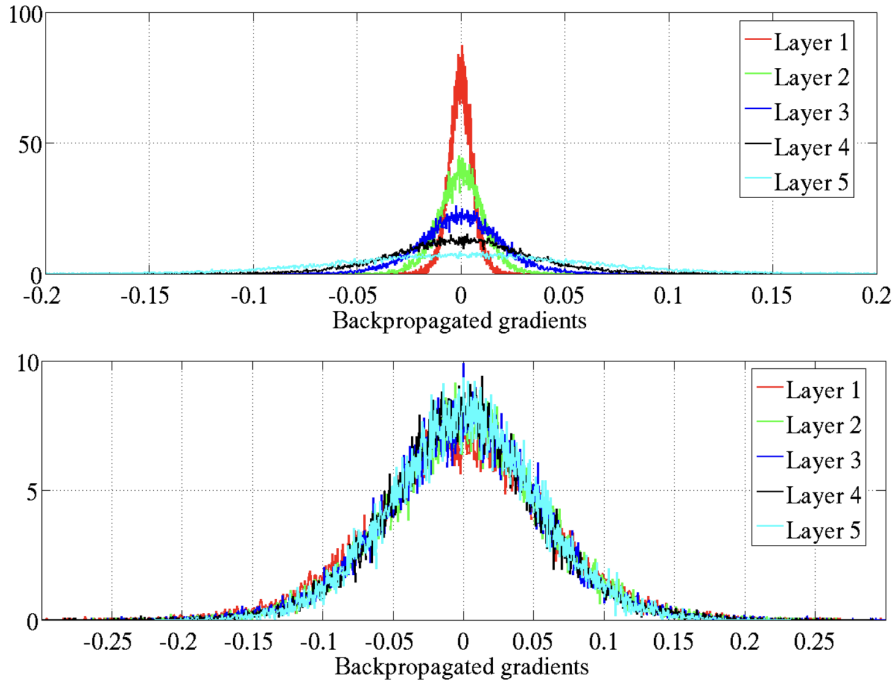


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

He (or Kaiming) Weights Initialization

Xavier initialization во многом опиралась на поведение функции активации $tanh$.

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Kaiming He et al.

$\mathbf{x} = ReLU(\mathbf{z}_{prev}), \mathbb{E}(\mathbf{z}_{prev}) = 0$

$Var(\mathbf{w}^T\mathbf{x}) = \sum_i^N (\mathbb{E}[x_i]^2 Var(w_i) + \mathbb{E}[w_i]^2 Var(x_i) + Var(w_i)Var(x_i)) = (\mathbb{E}[x_i]^2 + \mathbb{V}(x_i))\mathbb{V}(w_i)$

Так как $Var(x) = \mathbb{E}[x_i^2] - \mathbb{E}[x_i]^2$, то

$Var(\mathbf{w}^T\mathbf{x}) = n_{in} Var(w_i)\mathbb{E}(x_i^2)$

$\mathbb{E}(z_{prev}) = 0 \Rightarrow \mathbb{E}(x_i^2) = \frac{1}{2} Var(z_{prev})$

$\Rightarrow Var(\mathbf{w}^T\mathbf{x}) = \frac{1}{2} n_{in} Var(w_i) Var(z_{prev}) \Rightarrow \forall i, \frac{1}{2} n_{in} Var(w_i) = 1$

$$\forall w_i \sim \mathbb{N}(0, \frac{2}{n_{in}})$$

Source - Яндекс учебник ML

He (or Kaiming) Weights Initialization

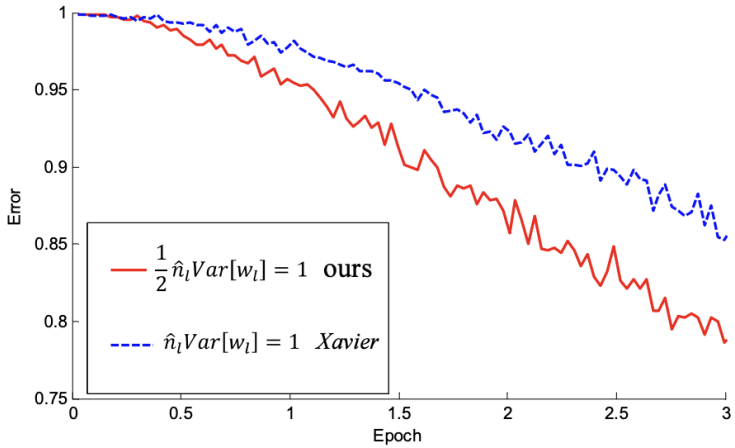


Figure 2. The convergence of a **22-layer** large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and "*Xavier*" (blue) [7] lead to convergence, but ours starts reducing error earlier.

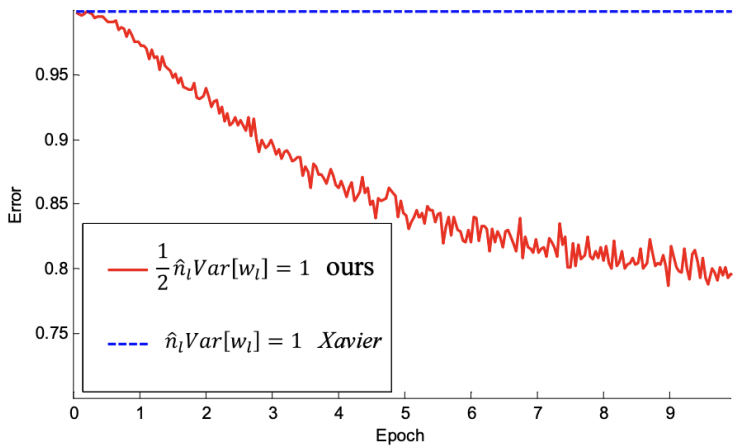
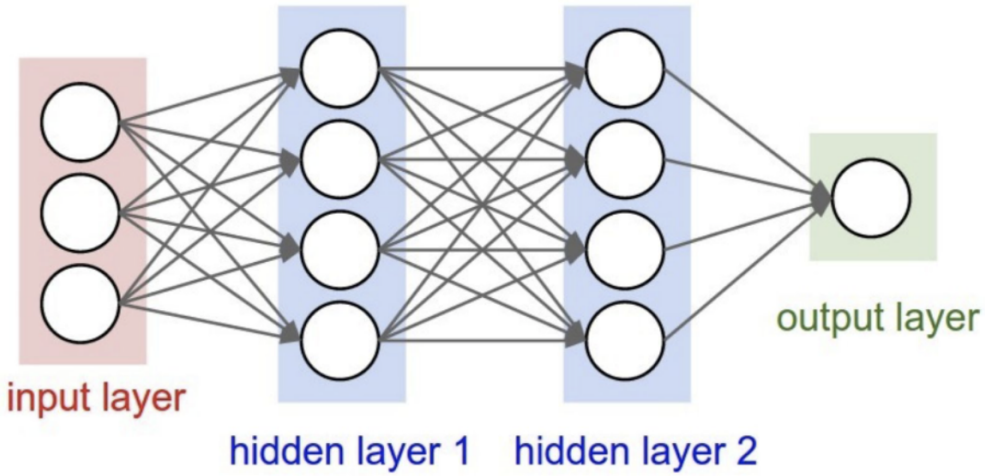


Figure 3. The convergence of a **30-layer** small model (see the main text). We use ReLU as the activation for both cases. Our initialization (red) is able to make it converge. But "*Xavier*" (blue) [7] completely stalls - we also verify that its gradients are all diminishing. It does not converge even given more epochs.

Batch Normalization

Problem:

- Consider a neuron in any layer beyond first
- At each iteration we tune it's weights towards better loss function
- But we also tune it's inputs. Some of them become larger, some – smaller
- Now the neuron needs to be re-tuned for it's new inputs



Source - girafe.ai

Batch Normalization

Допустим мы обновляем параметры W^k k-слоя.

$$x^k = f(x^{k-1}, W^k)$$

$$\nabla_{W^k} L = g(x^{k-1}, x^k, x^{k+1}, \ldots; W^k)$$

$$W^k_{new} = W^k - \alpha \nabla_{W^k} L = W^k - \alpha g(x^{k-1}, x^k, \ldots)$$

Далее аналогично обновляем W^{k-1}. Но это приведёт к изменению представления, которое пришло на вход k-му слою, которое мы не учитываем:

$$x^k_{new} = f(x^{k-1}, W^k_{new}) = f(x^{k-1}, W^k + \phi)$$

где $\phi = W^k_{old} - W^k_{new}$

То есть параметры $(k-1)$-го слоя будут обновлены исходя из предположения, что данные приходят из некоторого распределения на x_k, которое параметризовалось W^{k-1}, но теперь параметры изменились и данные могут обладать иными свойствами.

Source - Яндекс учебник ML

Batch Normalization

Input: $x : N \times D$

Learnable params:
$\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$
$\hat{x} : N \times D$

Output: $y : N \times D$

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

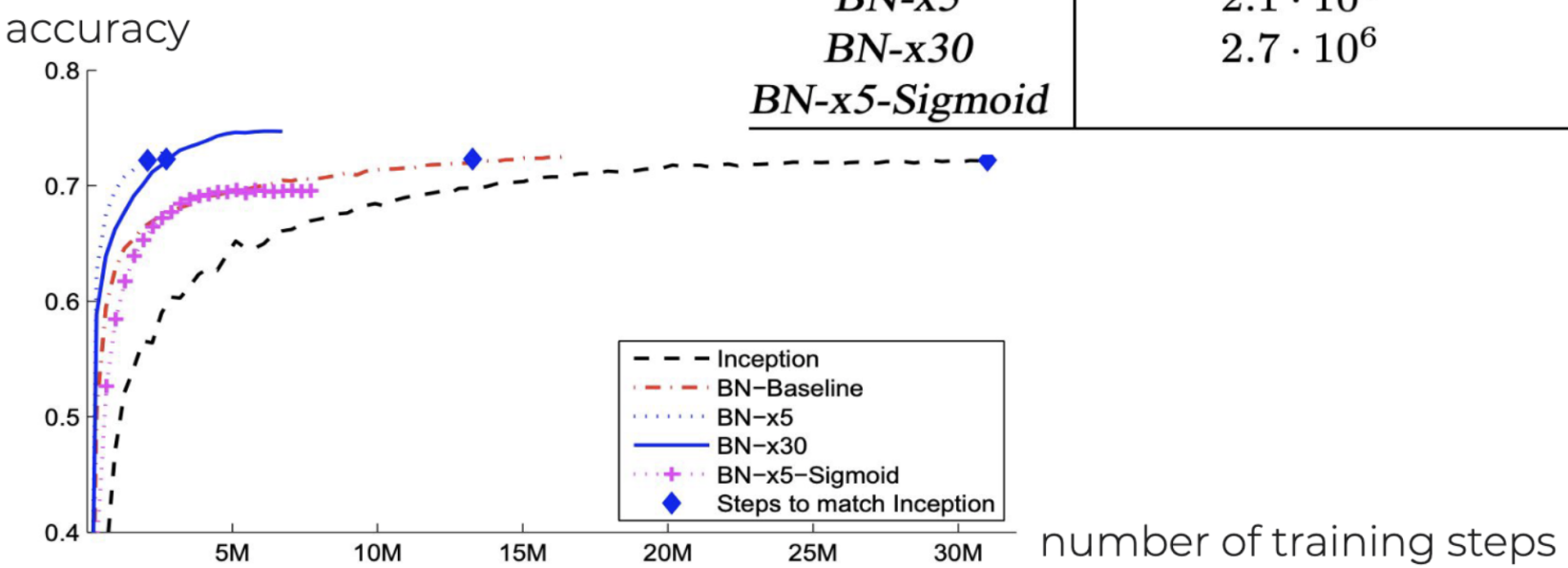
$\mu_* = \lambda\mu_* + (1-\lambda)\mu$ and $\sigma_* = \lambda\sigma_* + (1-\lambda)\sigma$ $\Leftarrow$ Need check

How to learn γ, β parameters see in this YouTube video:
NN - 22 - Batch Normalization - Derivatives and Inference

Source - Stanford CS231n and Яндекс учебник ML

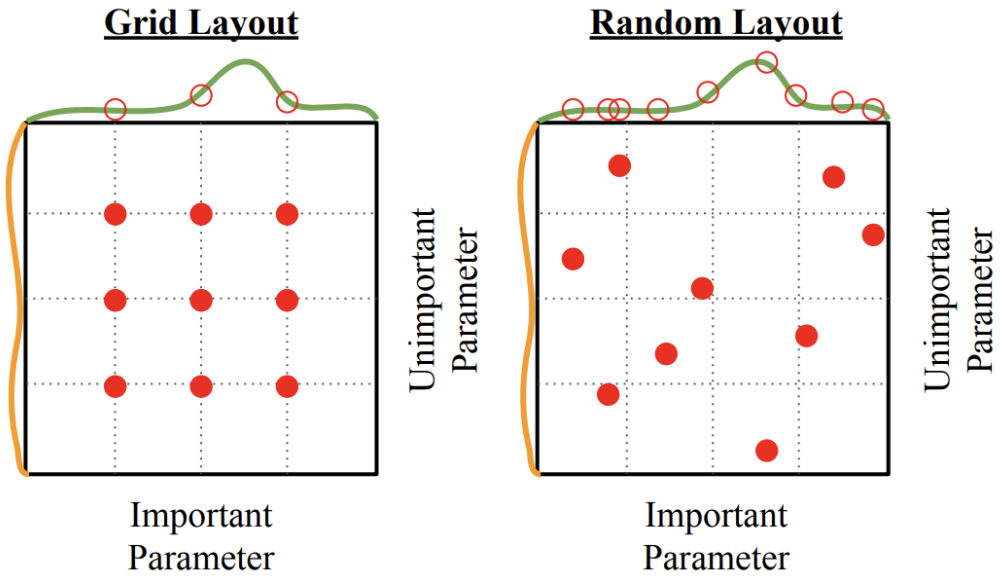
Batch Normalization

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



Source - girafe.ai

Hyperparameter Search



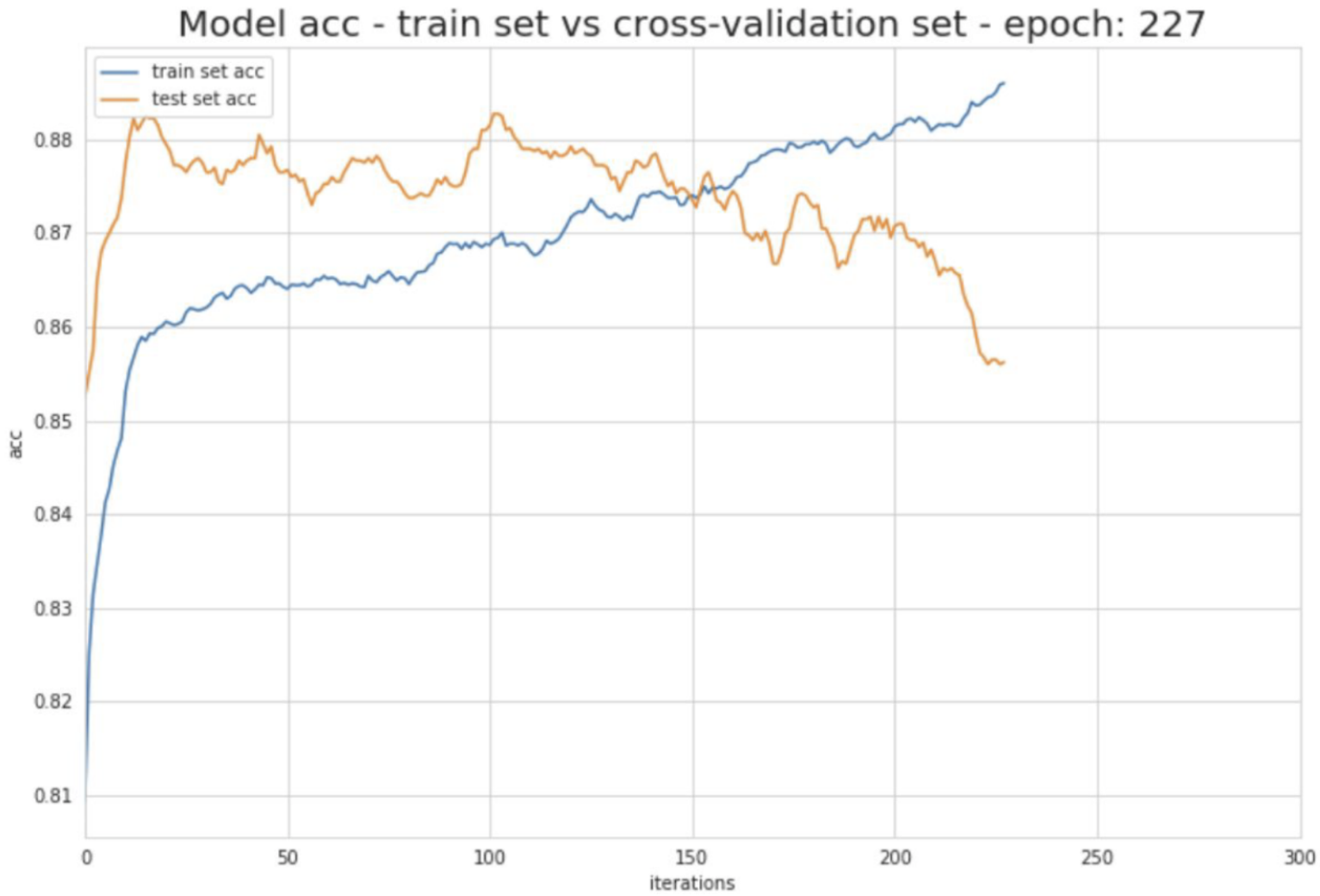
Coarse to fine search

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

Source - Stanford CS231n

Overfitting Problem



Source - girafe.ai

Regularization. L1, L2

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

In common use:

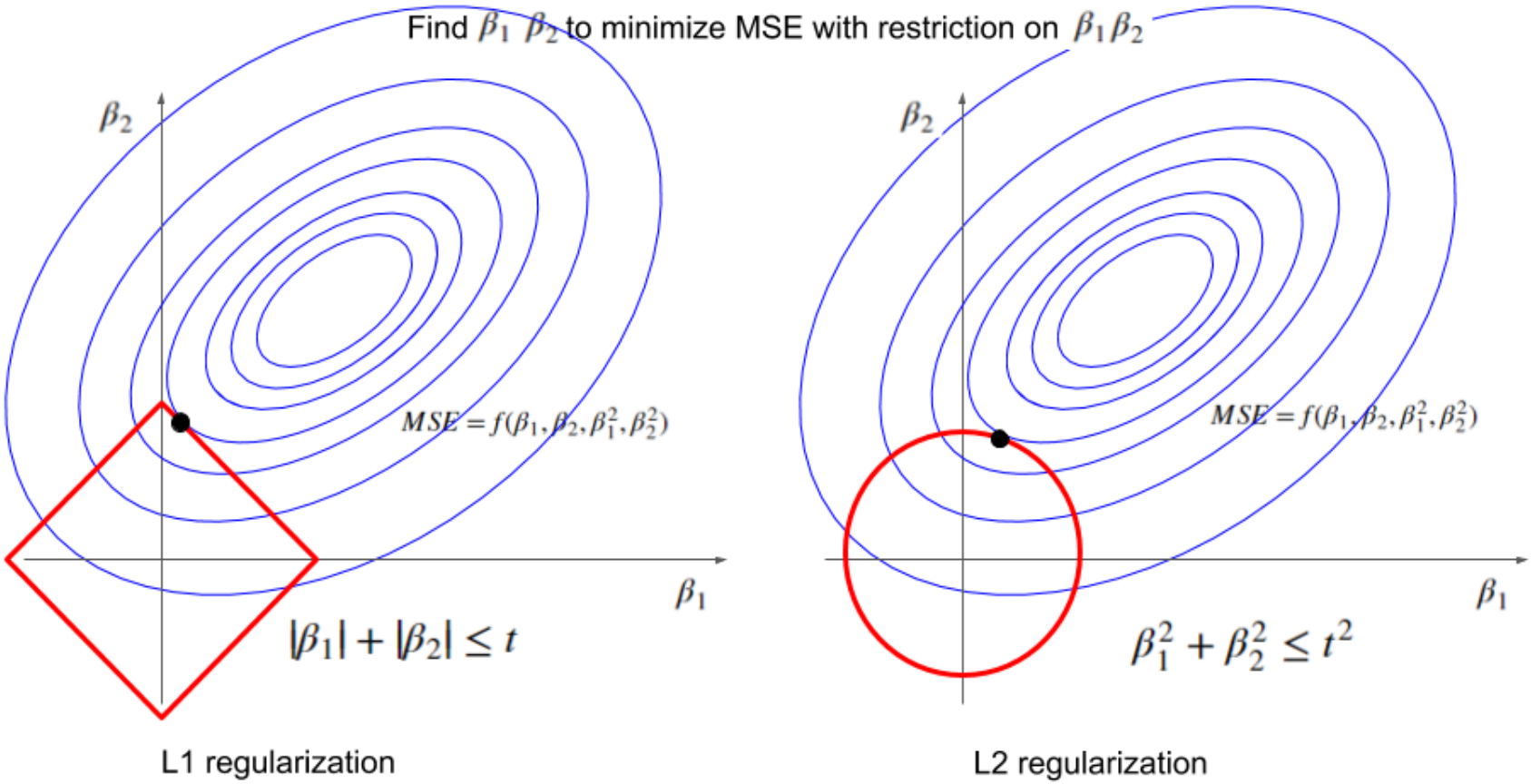
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

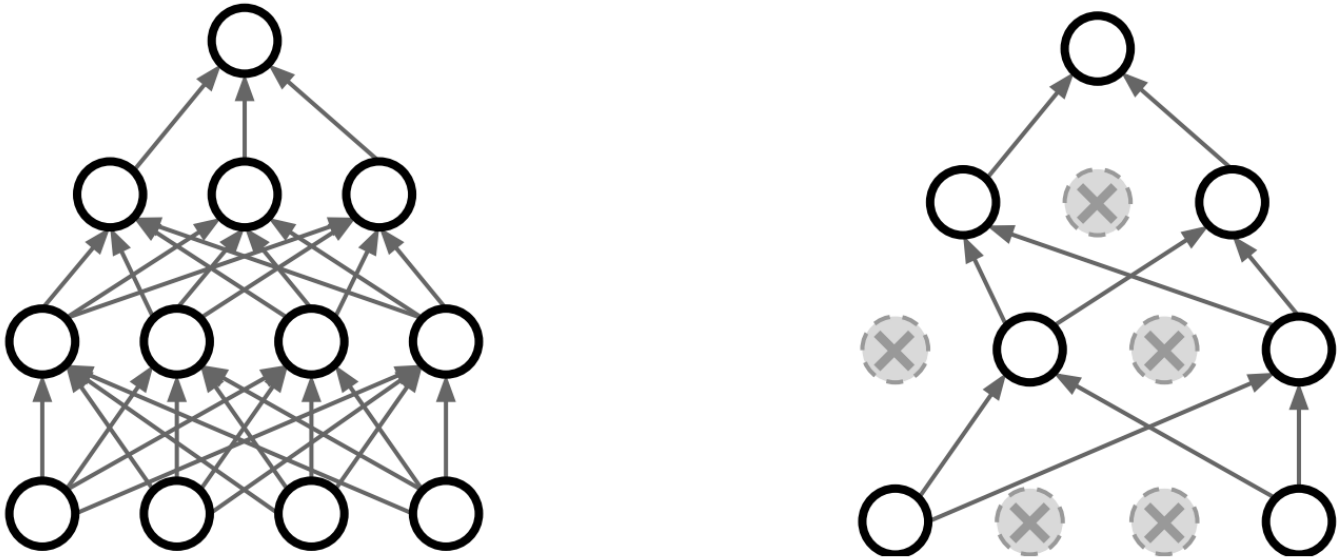
Source - Stanford CS231n

Regularization. L1, L2



Regularization. Dropout

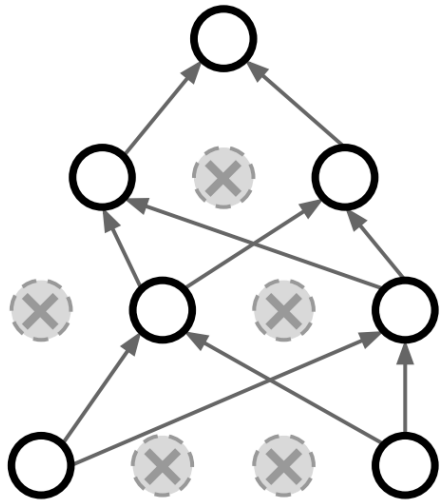
In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



Source - Stanford CS231n

Regularization. Dropout

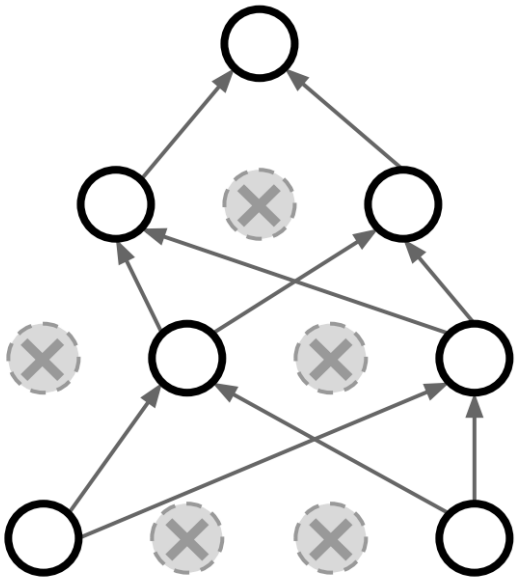
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```



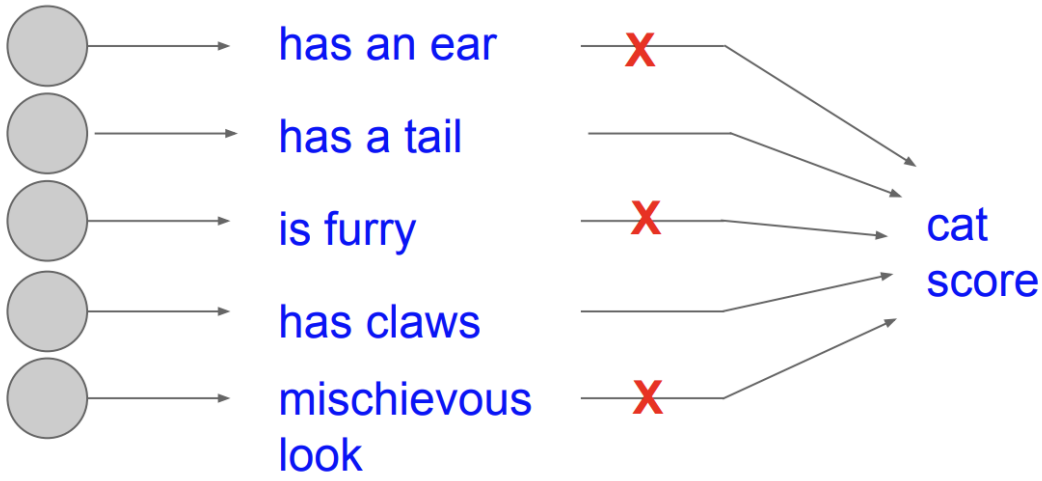
Source - Stanford CS231n

Regularization. Dropout

How can this possibly be a good idea?



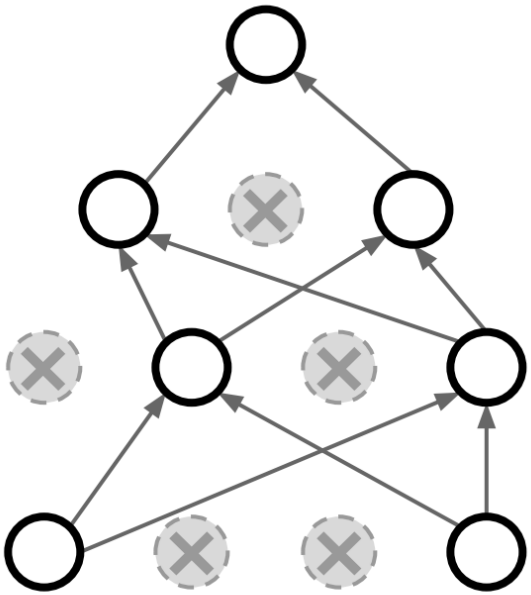
Forces the network to have a redundant representation;
Prevents co-adaptation of features



Source - Stanford CS231n

Regularization. Dropout

How can this possibly be a good idea?



Another interpretation:

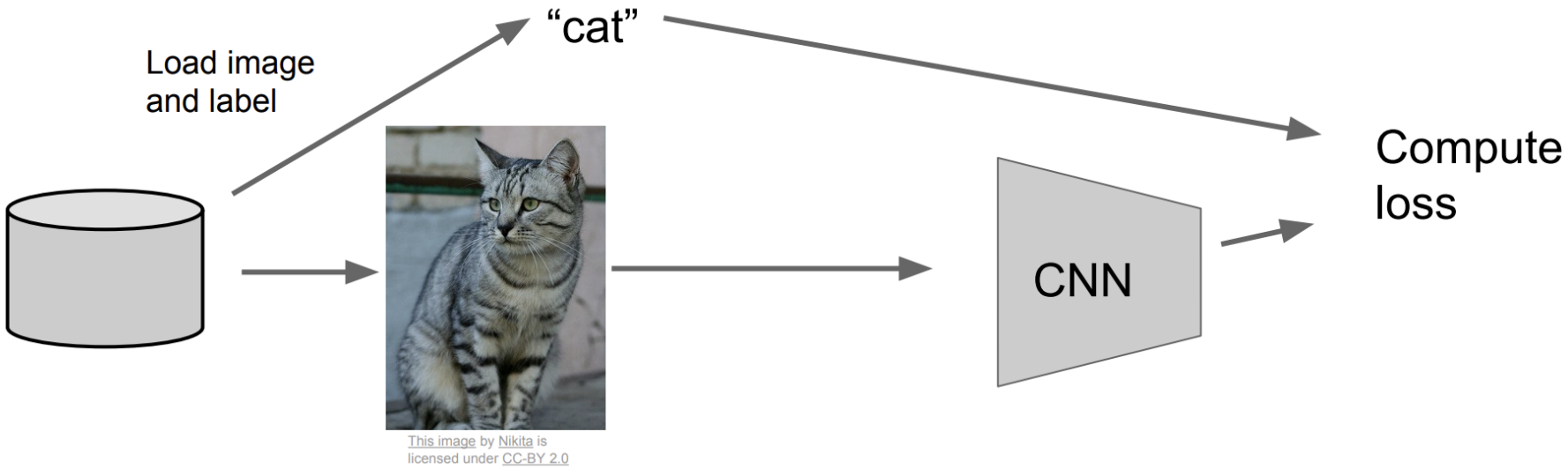
Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

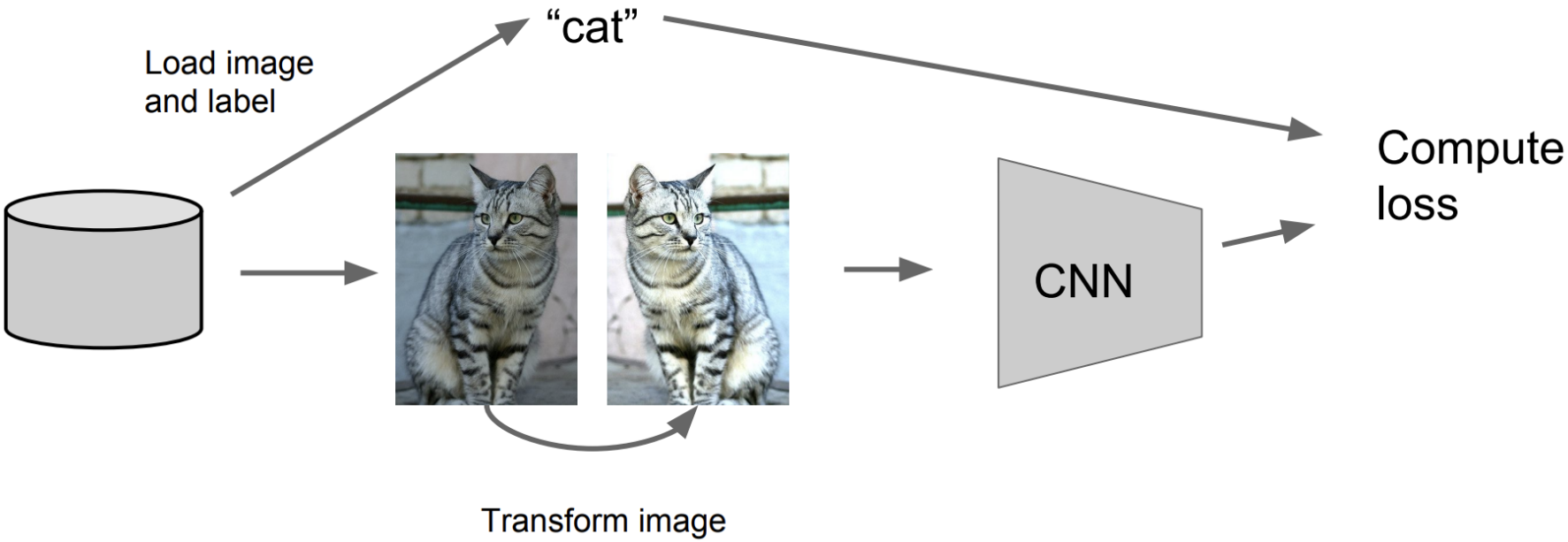
Source - Stanford CS231n

Regularization. Data Augmentation



Source - Stanford CS231n

Regularization. Data Augmentation



Use more - filters, adding noise, rotation, randomize contrast and brightness...

Source - Stanford CS231n