Obučavanje Neuronskih Mreža
Prvi Deo

Predavač: Aleksandar Kovačević

Slajdovi preuzeti sa CS 231n, Stanford

http://cs231n.stanford.edu/

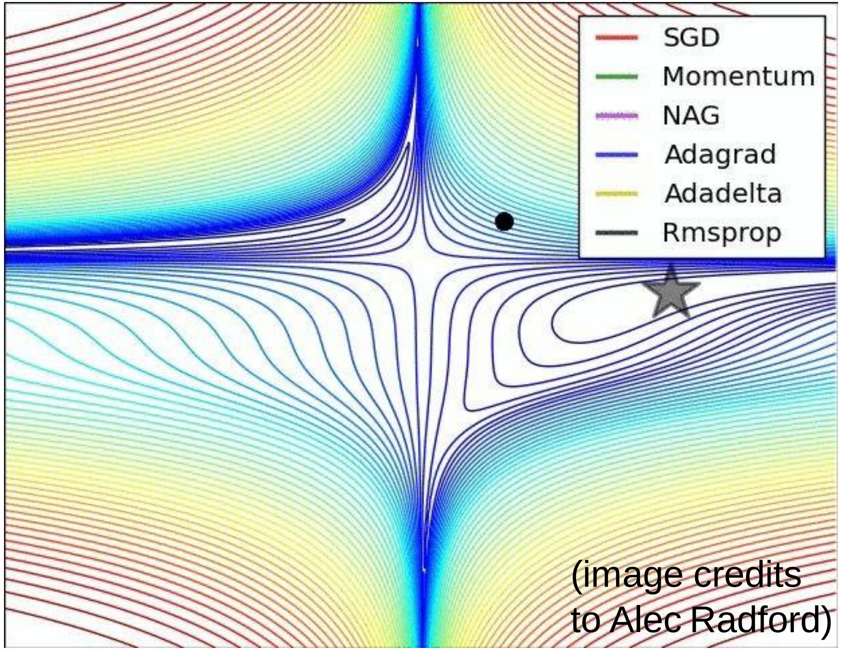
Šta smo do sada naučili...

Stohastički Gradijentni Spust sa Mini-Podskupovima (*Mini-batch SGD*)

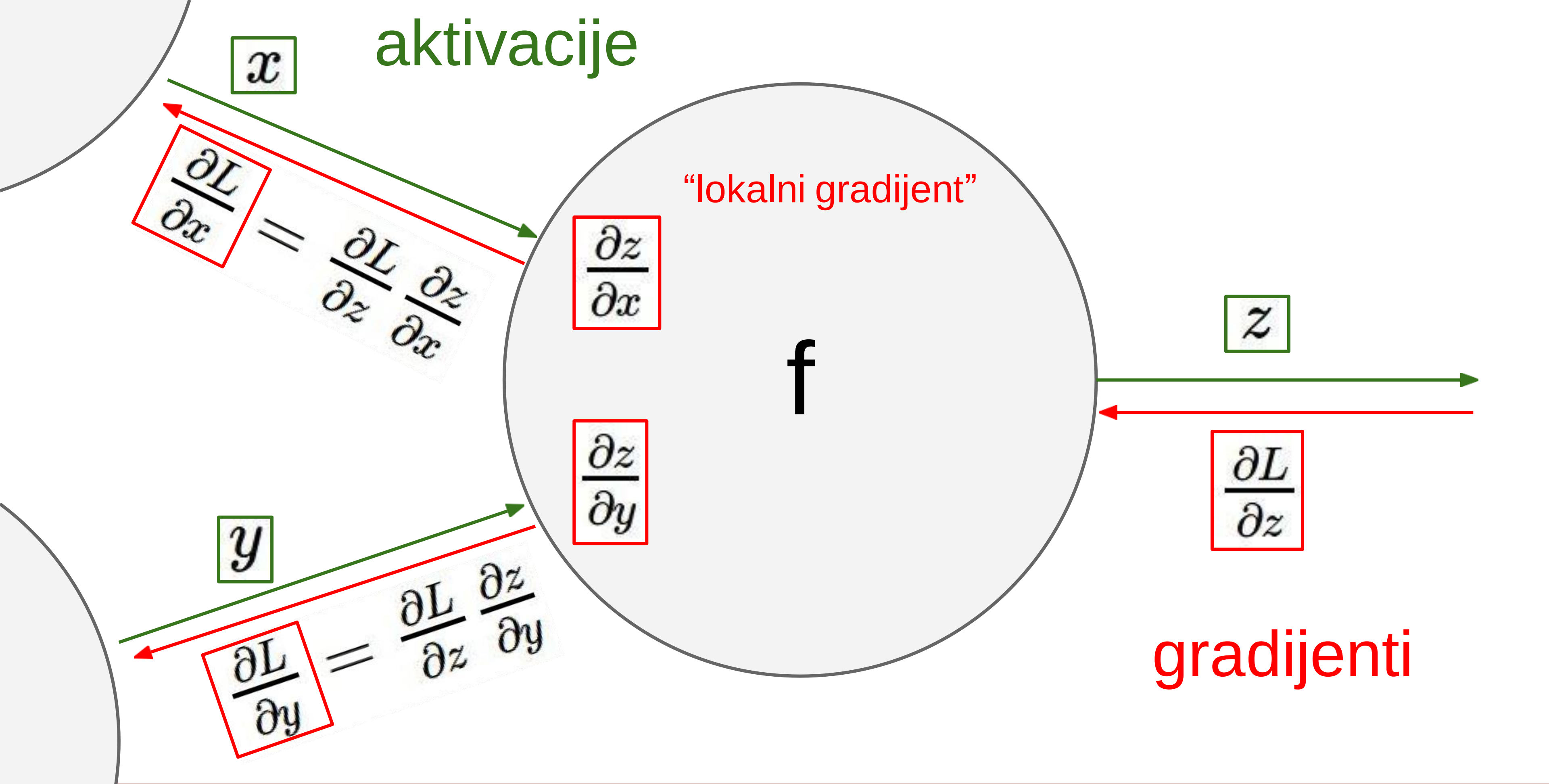
Loop:

1. Semplujemo podskup obučavajućeg skupa
2. Za svaki primer iz podskupa
    1. izračunavamo izlaz iz mreže („guramo“ ga unapred kroz mrežu – feed forward)
    2. kada dobijemo izlaz za primer, izračunamo vrednost funkcije greške i tu vrednost dodajemo na ukupnu grešku za taj podskup
    3. izračunavamo gradijente za svaki parametar pomoću Backpropagation. Za svaki parametar dodajemo gradijent na zbir gradijenata koje on čuva.
3. Izračunavamo prosek gradijenta za svaki parametar. Prosek je zbir gradijenata (dobijen iz koraka 2.3) za ceo podskup podeljen sa veličinom podskupa.
4. Menjamo vrednosti parametara pomoću proseka gradijenta koji mu odgovara.

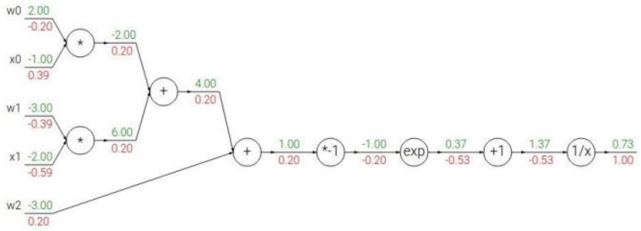
Šta smo do sada naučili...



(image credits
to Alec Radford)


aktivacije
x
y
z
"lokalni gradijent"
f
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradijenti

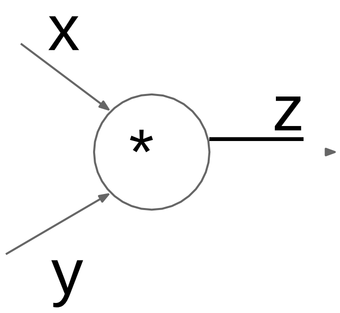
Implementacija: forward/backward API



Graf (ili Net) objekat. *(Pseudo kod)*

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

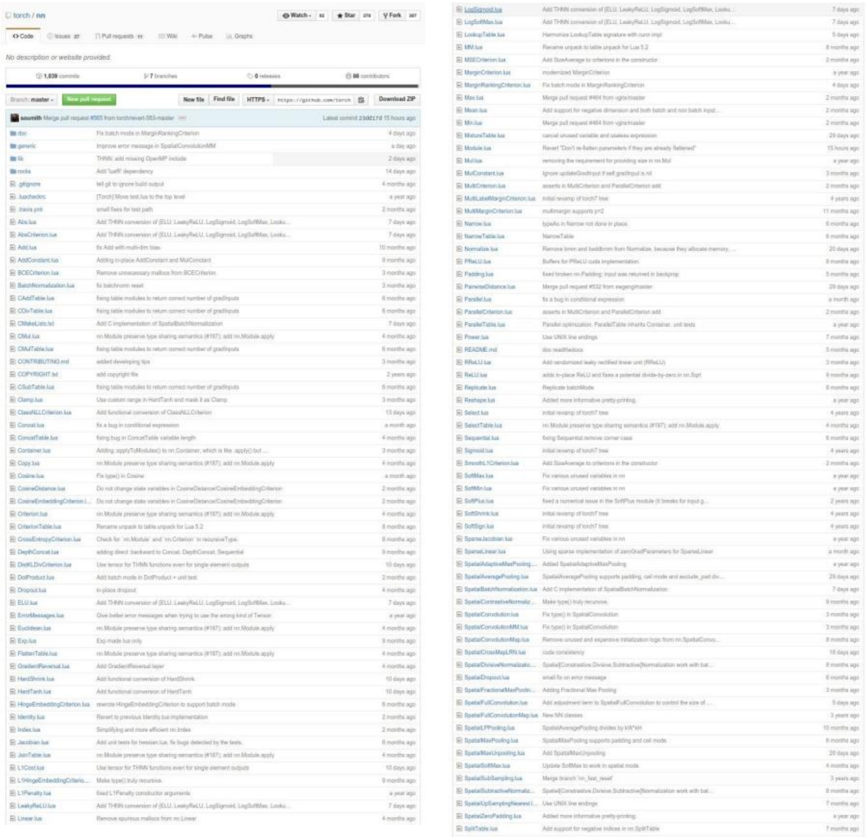
Implementacija: forward/backward API




```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

(x,y,z su skalari)

Primer: Torch Slojevi



=

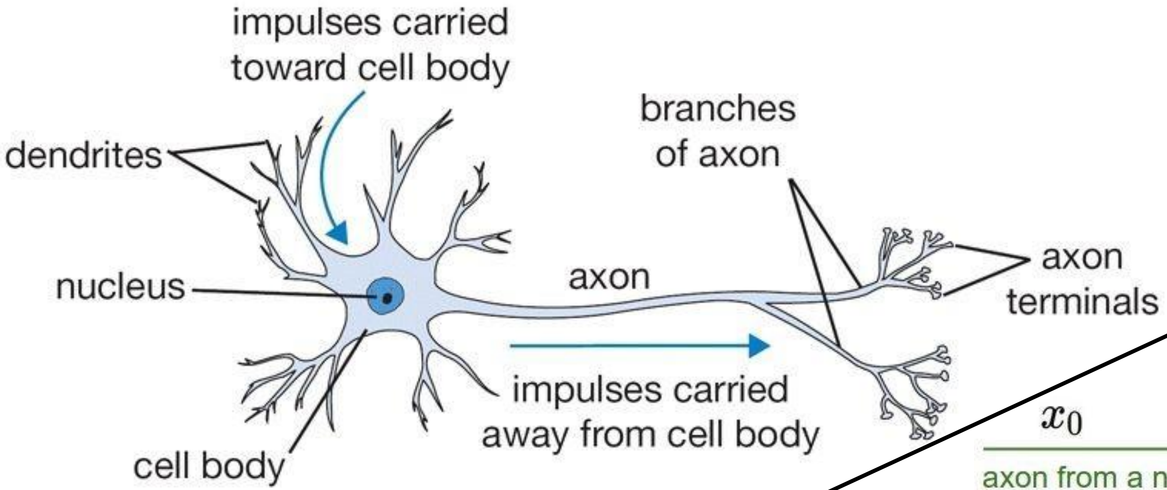
Neuronske Mreže: prvo bez analogije sa ljudskim mozgom

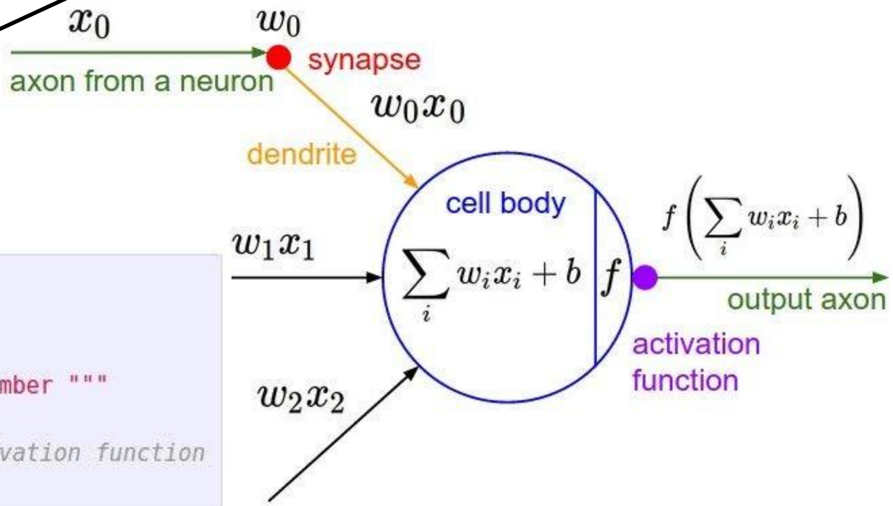
(**Ranije**) Linearna skor funkcija: $f = Wx$

(**Sada**) Neuroska mreža sa dva sloja: $f = W_2 \max(0, W_1 x)$

Neuroska mreža sa tri sloja:

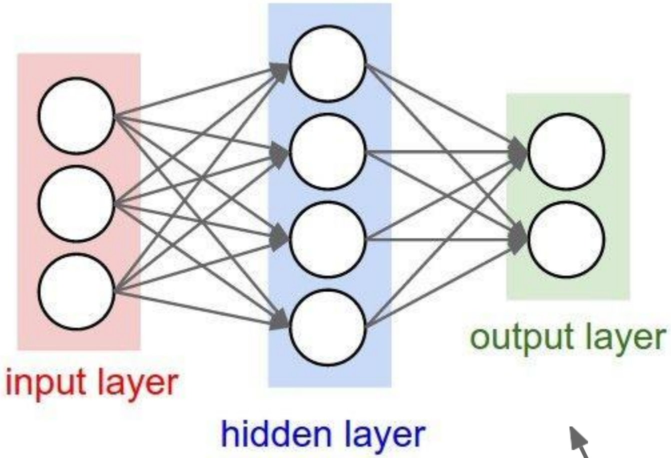
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

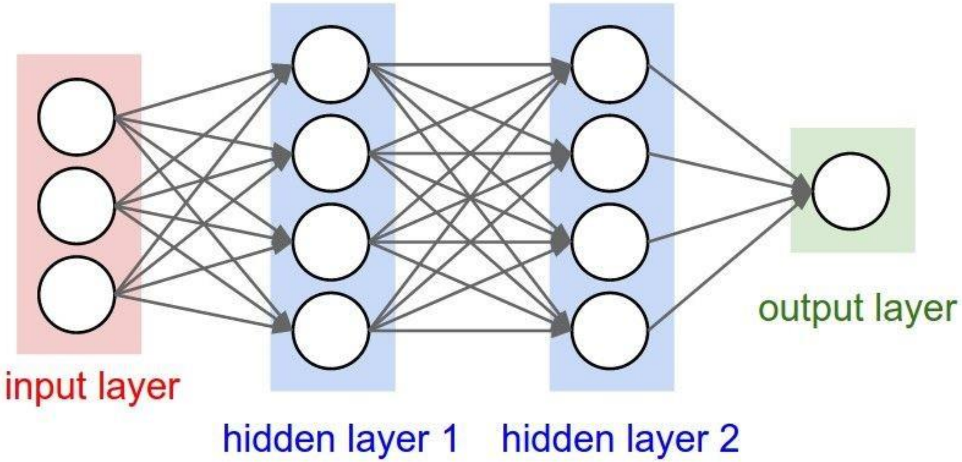





```python
class Neuron:
  # ...
  def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

Neuronske Mreže: Arhitekture





"2-slojna NN", ili
"Neuronska mreža koja ima jedan skriveni sloj"

"3-slojna NN", ili
"Neuronska mreža koja ima dva skrivena sloja"

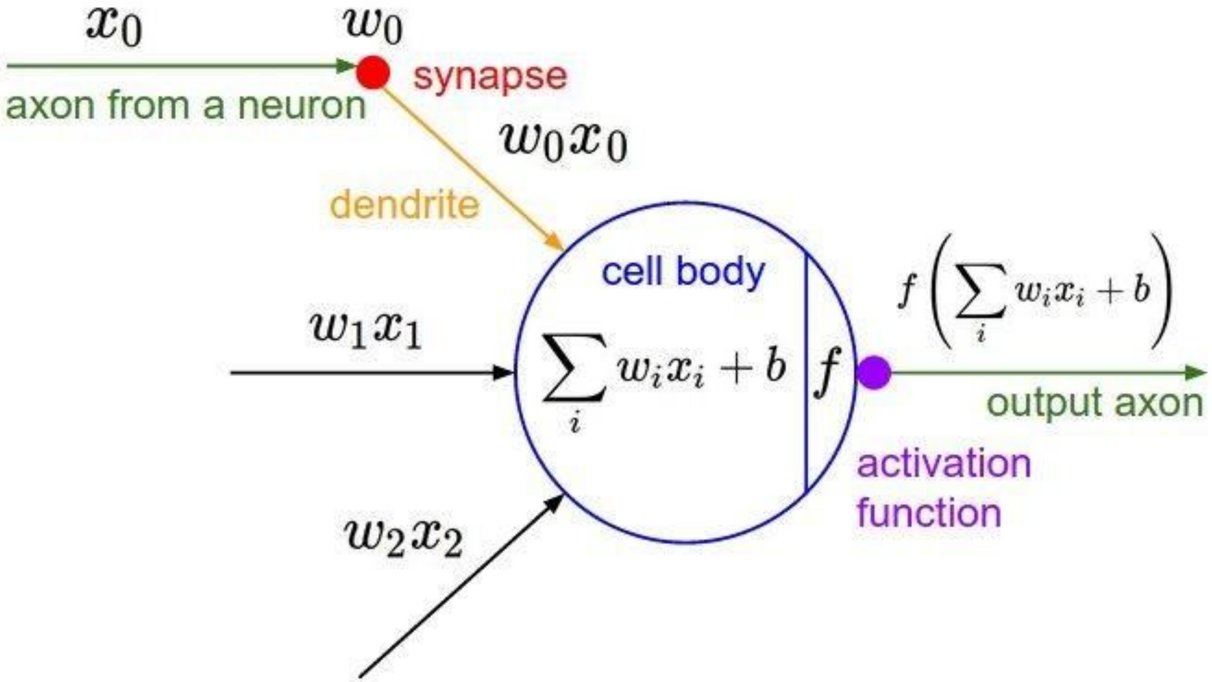
"Potpuno-povezani" slojevi

Obučavanje Neuronskih Mreža

Malo istorije...

Funkcije Aktivacije

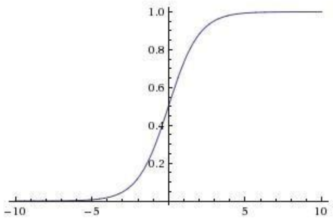
Funkcije Aktivacije



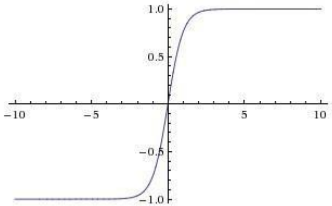
Funkcije Aktivacije

Sigmoid

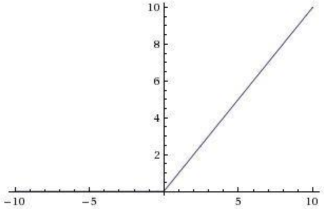
$$\sigma(x) = 1/(1+e^{-x})$$



tanh tanh(x)

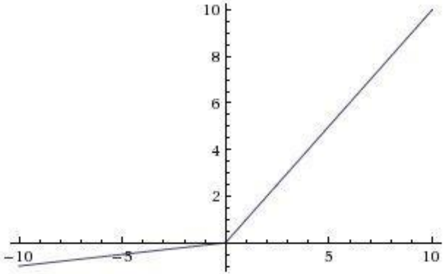


ReLU max(0,x)



Leaky ReLU

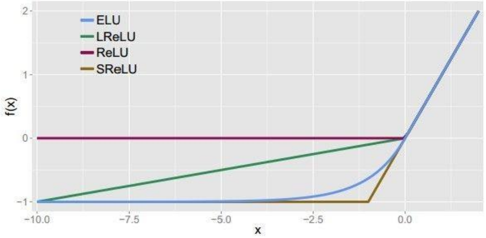
max(0.1x, x)



Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

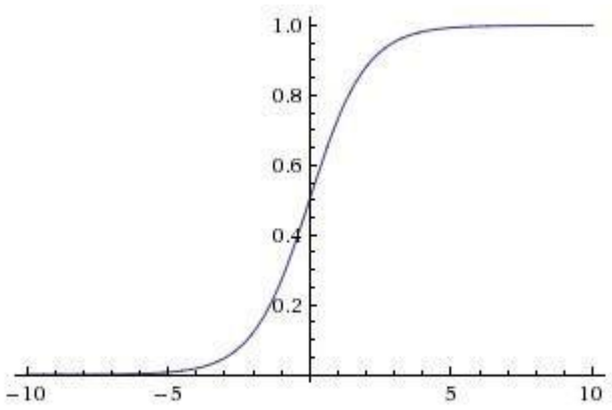
ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$



Funkcije Aktivacije

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

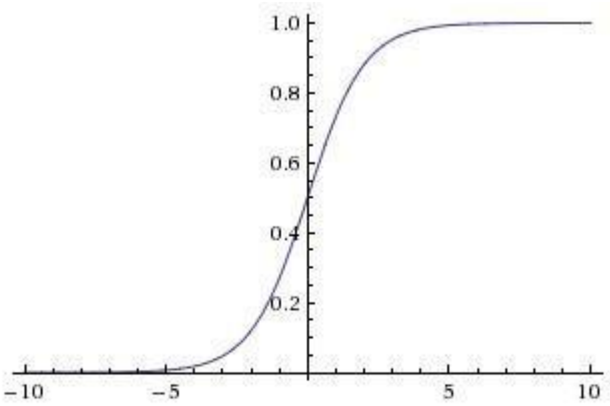


Sigmoid

- „Skuplja“ ulaz na raspon [0,1]
- Istorijski najpopularniji izbor za aktivaciju

Funkcije Aktivacije

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

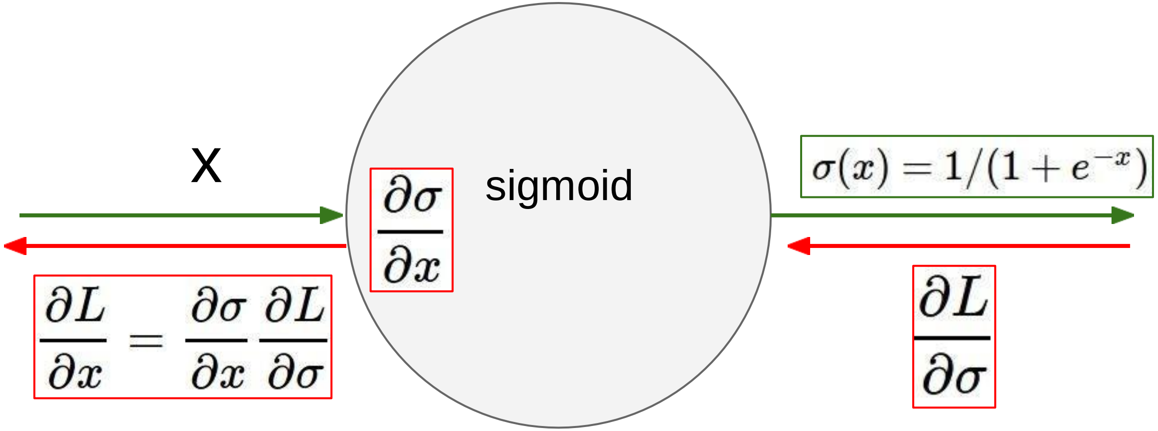


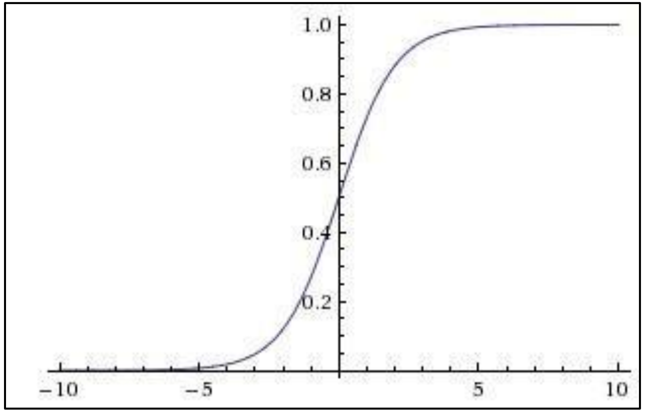
Sigmoid

- „Skuplja“ ulaz na raspon [0,1]
- Istorijski najpopularniji izbor za aktivaciju

Tri Problema:

1. Neuroni u saturaciji (jako male ili velike vrednosti) „ubijaju“ gradijent

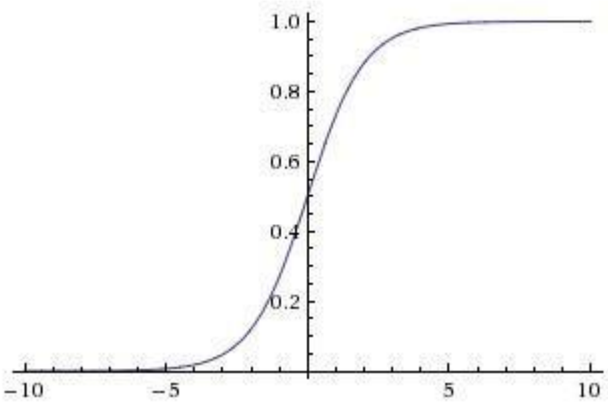




Šta se dešava kad je x = -10?
Šta se dešava kad je x = 0?
Šta se dešava kad je x = 10?

Funkcije Aktivacije

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



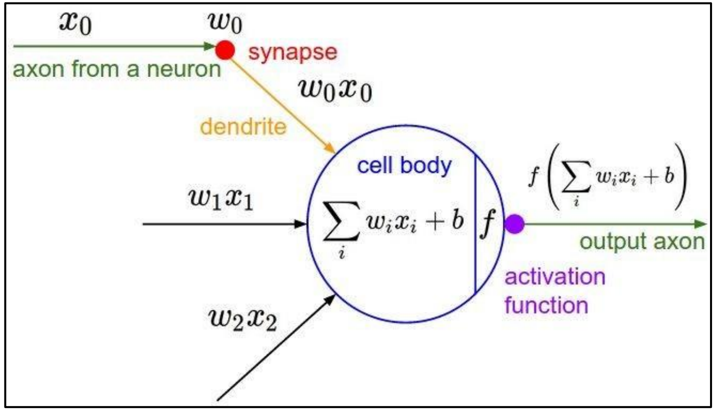
Sigmoid

- „Skuplja“ ulaz na raspon [0,1]
- Istorijski najpopularniji izbor za aktivaciju

Tri Problema:

1. Neuroni u saturaciji (jako male ili velike vrednosti) „ubijaju“ gradijent
2. Izlaz nema srednju vrednost 0 tj. nije *zero-centered*

Šta se dešava kad su svi ulazi u neuron (x) pozitivni:

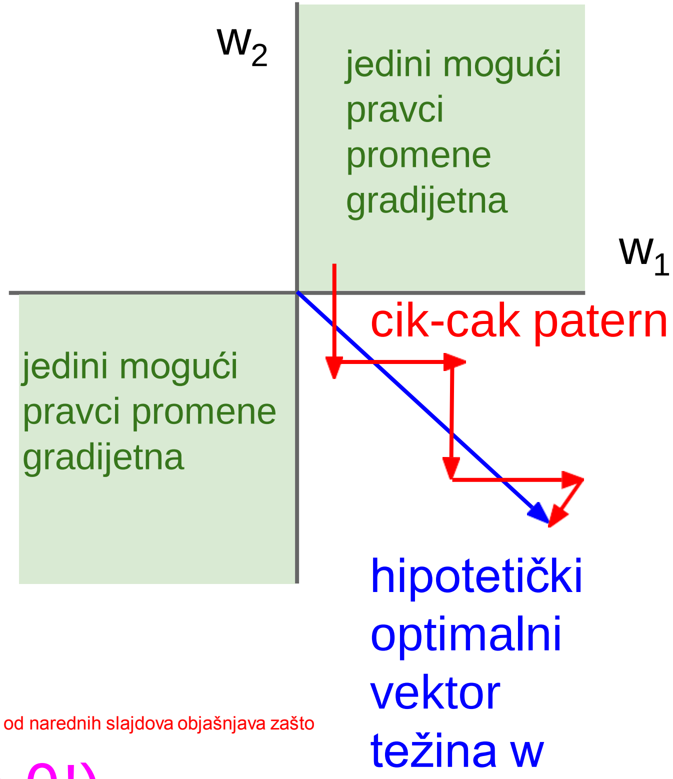


$$f\left(\sum_i w_i x_i + b\right)$$

Kako tada izgledaju gradijenti **w**?

Šta se dešava kad su svi ulazi u neuron (x) pozitivni:

$$f\left(\sum_i w_i x_i + b\right)$$



Kako tada izgledaju gradijenti **w**?

Svi su uvek pozitivni ili su svi negativni :(jedan od narednih slajdova objašnjava zašto

(zato želimo da su vrednosti centrirane oko 0!)

Šta se dešava kad su svi ulazi u neuron (x) pozitivni:

Da bi pronašli hipotetički optimalni w sa slike, backprop mora u isto vreme da povećava w_1 i smanjuje w_2.

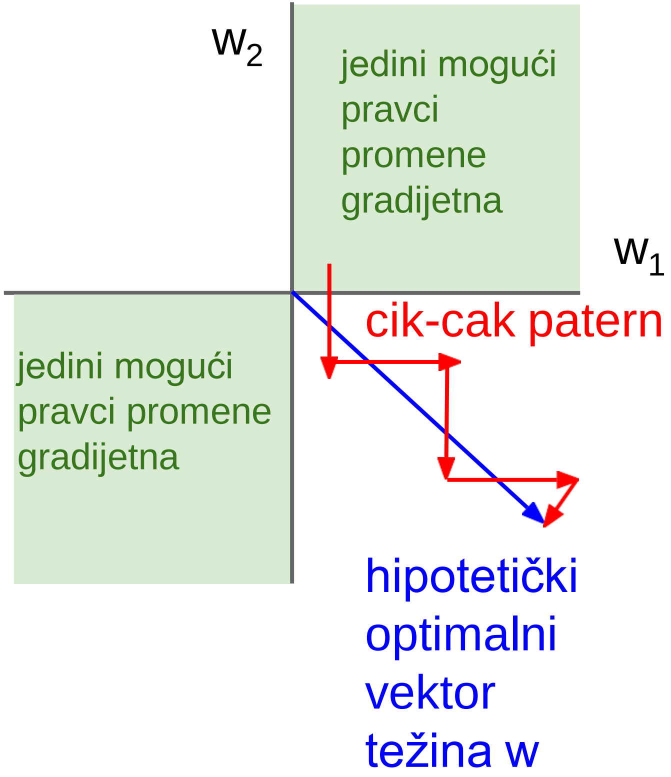
To neće biti moguće jer mu gradijenti to ne dozvoljavaju pošto su istog znaka.

To će onda rezultovati time da će backprop menjati w_1 i w_2 jedno po jedno.

Prvo će malo da smanji w_2, pa posle poveća w_1.

Tako dobijamo cik-cak patern.

Naravno neće bukvalno menjati samo jedan, već će promene jednog biti toliko male da deluje kao da se ne menja.



Šta se dešava kad su svi ulazi u neuron (x) pozitivni:

Da još malo diskutijemo:

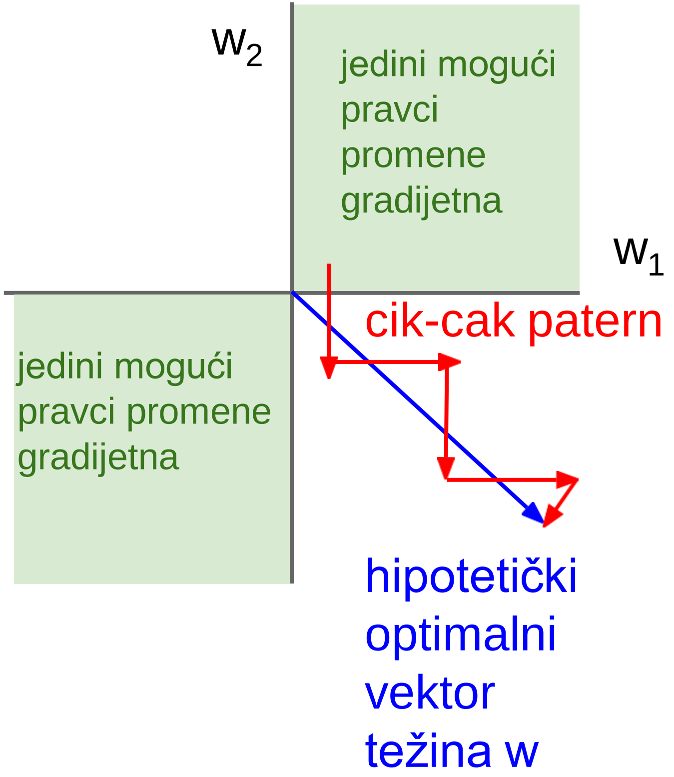
Šta dobijamo ako bi izlaz funkcije aktivacije bio centriran oko 0?

Znači da bi onda neki izlazi bili negativni, a neki pozitivni.

To znači da bi onda ulazi u neke od narednih neurona bili malo negativni malo pozitvni.

To bi nam onda omogućilo da gradijenti težina budu malo negativni malo pozitivni.

Tako bi izbegli cik-cak patern jer bi mogli da u isto vreme jednu težinu povećavamo, a drugu smanjujemo i da se tako krećemo do idealnog w.



Šta se dešava kad su svi ulazi u neuron (x) pozitivni:

Zašto su gradijenti w svi uvek pozivini ili uvek negativni:

$$f = \sum w_i x_i + b$$

$$\frac{df}{dw_i} = x_i$$

$$\frac{dL}{dw_i} = \frac{dL}{df}\frac{df}{dw_i} = \frac{dL}{df}x_i$$

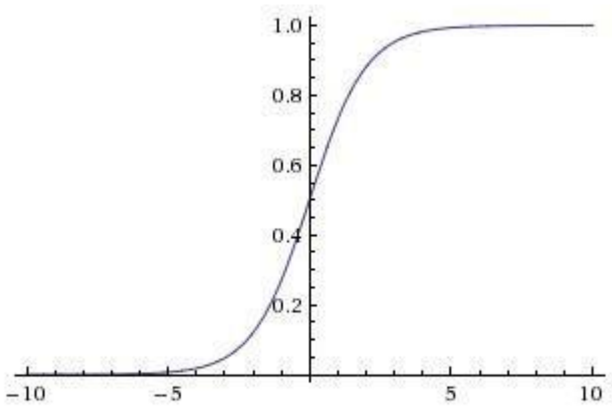
za sigmoid važi da je dl/df: (1-f)*f

L bi bila neka funkcija od *f*, npr. sigmoid.

Ako su svi *x* pozitivni onda su gradijenti ili pozitivni ili negativni, a to zavisi samo od izvoda *dL/df*

Funkcije Aktivacije

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



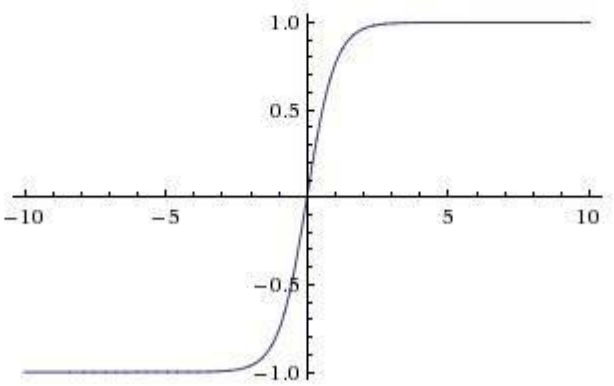
Sigmoid

- „Sabija“ ulaz na raspon [0,1]
- Istorijski najpopularniji izbor za aktivaciju

Tri Problema:

1. Neuroni u saturaciji (jako male ili velike vrednosti) „ubijaju“ gradijent
2. Izlaz nema srednju vrednost 0 tj. nije *zero-centered*
3. Računanje *exp* može biti skupo

Funkcije Aktivacije

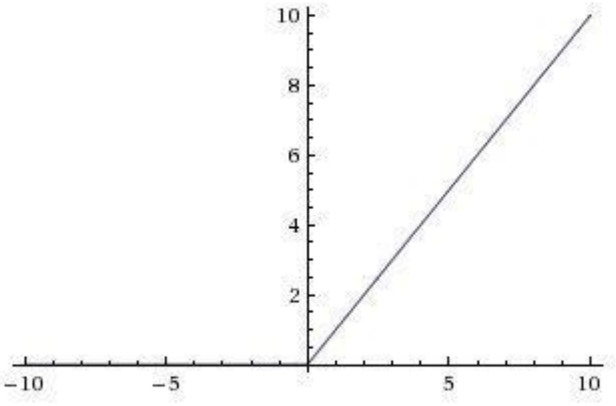


tanh(x)

- „Sabija“ ulaz na raspon [-1,1]
- izlaz je centiran oko 0
- i dalje „ubija“ gradjent prilikom staturacije :(

[LeCun et al., 1991]

Funkcije Aktivacije



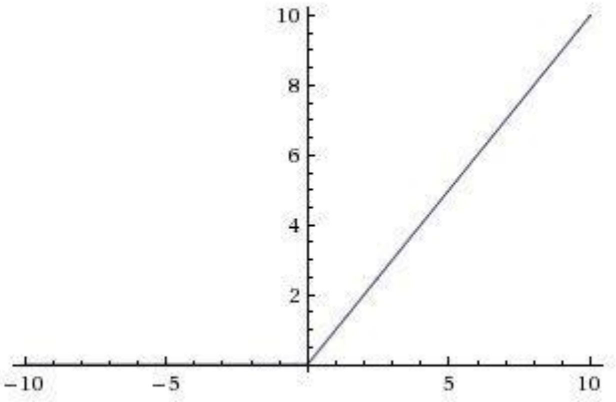
ReLU
(Rectified Linear Unit)

- Izračunava **f(x) = max(0,x)**

- Nema stauracije (u + regionu)
- Vrlo brzo izračunavanje
- Praksa je pokazala da GD mnogo brže konvergira nego kad koristimo sigmoid ili tanh (npr. nekad i do 6x brže)

[Krizhevsky et al., 2012]

Funkcije Aktivacije



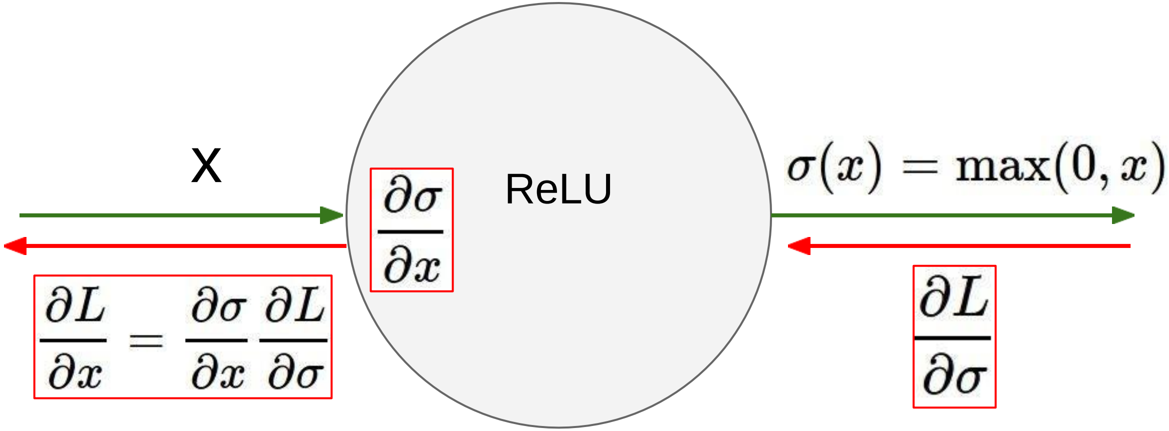
ReLU
(Rectified Linear Unit)

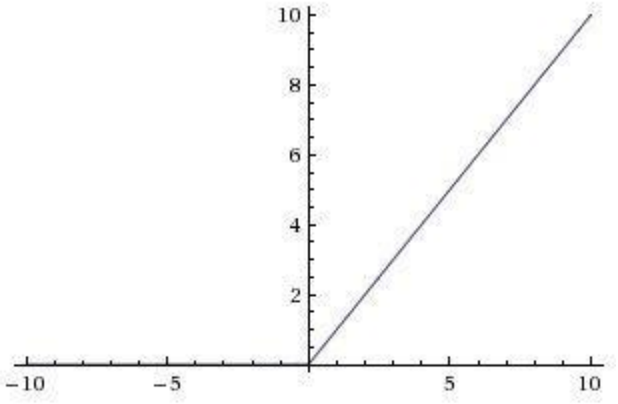
- Izračunava **f(x) = max(0,x)**

- Nema saturacije (u + regionu)
- Vrlo brzo izračunavanje
- Praksa je pokazala da GD mnogo brže konvergira nego kad koristimo sigmoid ili tanh (npr. nekad i do 6x brže)

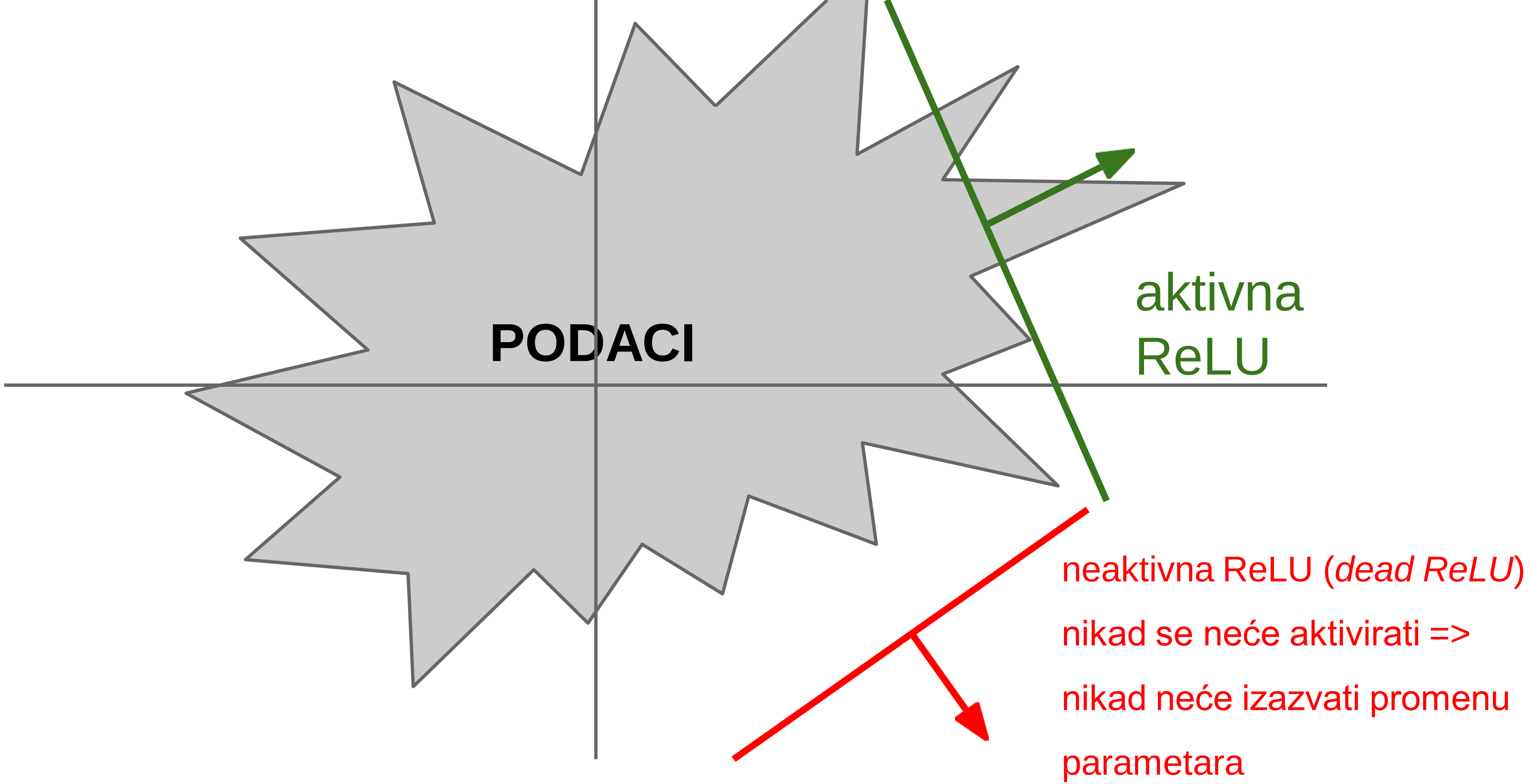
- Izlaz nije centriran oko 0
- Problem:

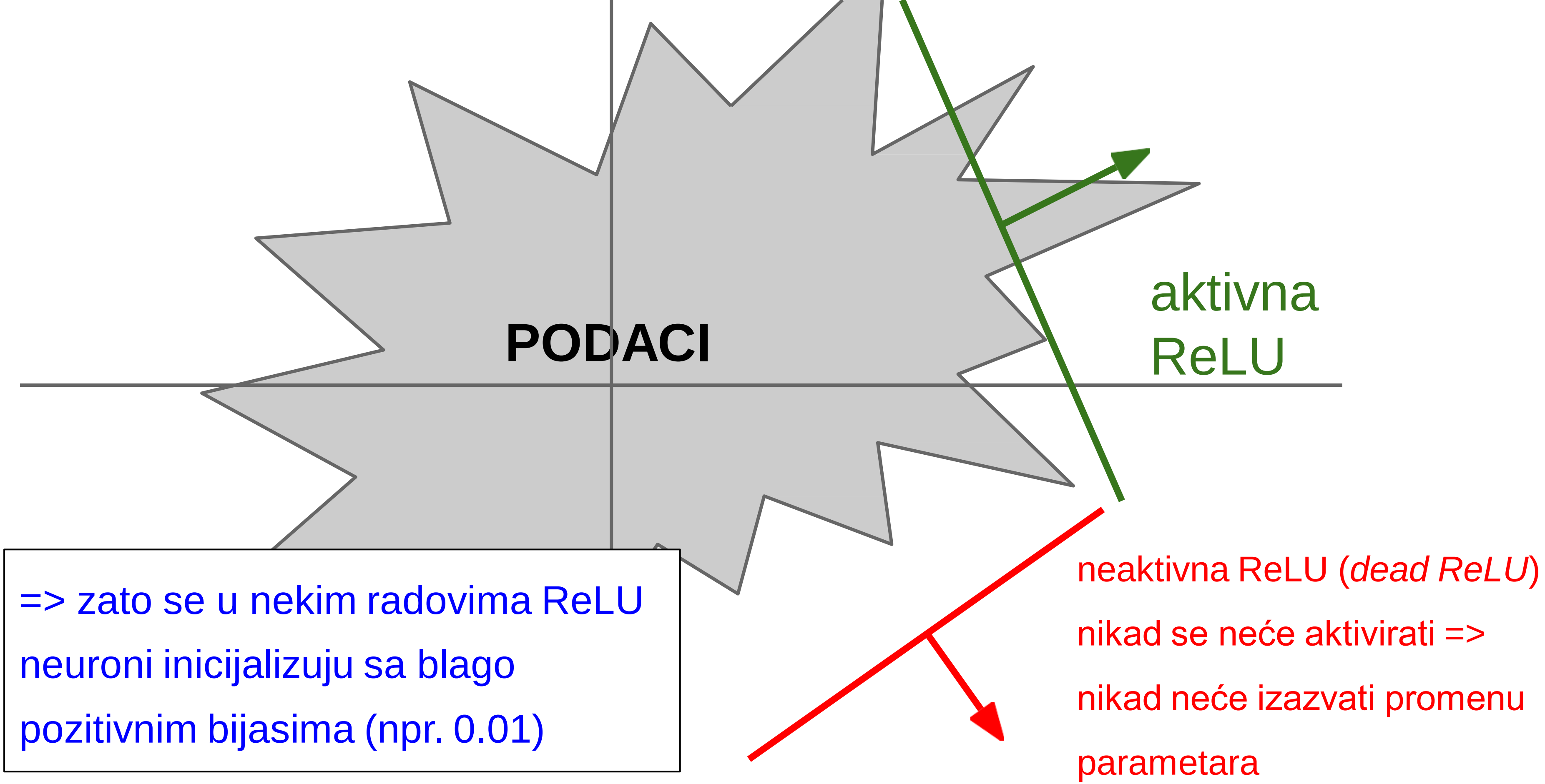
koji je gradijent kad je x < 0?





Šta se dešava kad je x = -10?
Šta se dešava kad je x = 0?
Šta se dešava kad je x = 10?

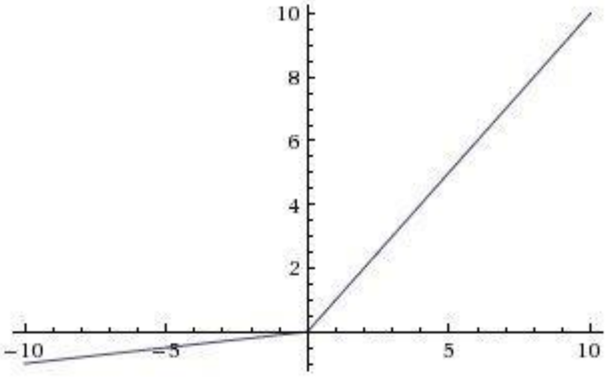

PODACI
aktivna
ReLU
neaktivna ReLU (*dead ReLU*)
nikad se neće aktivirati =>
nikad neće izazvati promenu
parametara



=> zato se u nekim radovima ReLU neuroni inicijalizuju sa blago pozitivnim bijasima (npr. 0.01)

Funkcije Aktivacije

[Mass et al., 2013]
[He et al., 2015]



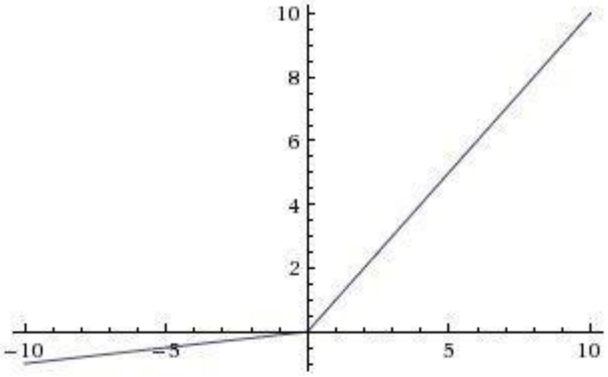
- Nema stauracije (u + regionu)
- Vrlo brzo izračunavanje
- Praksa je pokazala da GD mnogo brže konvergira nego kad koristimo sigmoid ili tanh (npr. nekad i do 6x brže)
- **neće “umreti”.**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Funkcije Aktivacije

[Mass et al., 2013]
[He et al., 2015]



- Nema stauracije (u + regionu)
- Vrlo brzo izračunavanje
- Praksa je pokazala da GD mnogo brže konvergira nego kad koristimo sigmoid ili tanh (npr. nekad i do 6x brže)
- **neće "umreti".**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

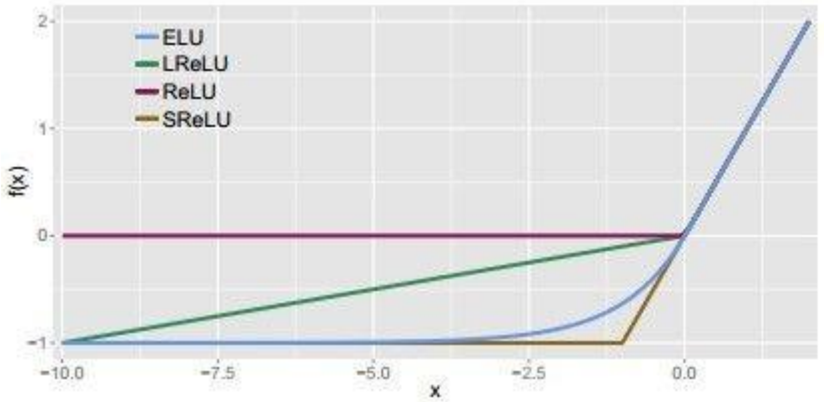
$$f(x) = \max(\alpha x, x)$$

koristimo backprop da odredimo parametar alfa kao i sve ostale

Funkcije Aktivacije

[Clevert et al., 2015]

Exponential Linear Units (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

- Sve dobre strane ReLU
- Ne „umire“
- Izlaz bliži centriranosti oko 0

- Moramo da izračunavamo exp()

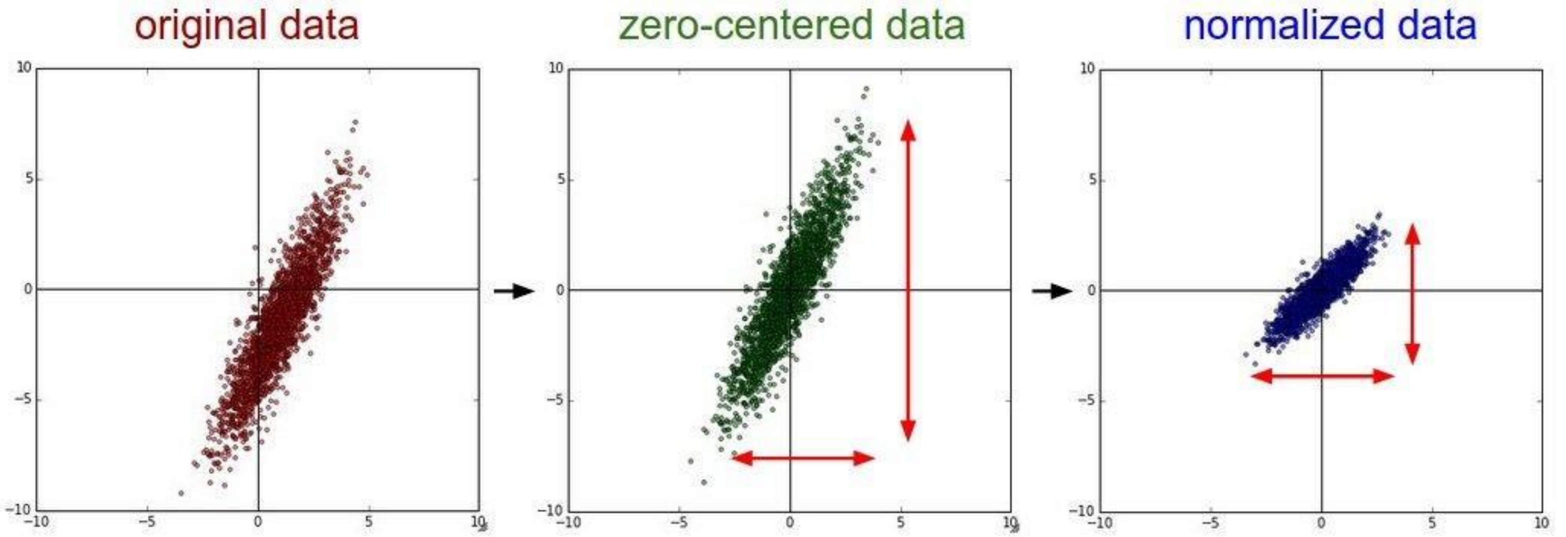
TLDR: U praksi:

- Koristite ReLU. Korak učenja (*learning rate*) je nešto na šta treba obratiti pažnju
- Probajte Leaky ReLU / Maxout / ELU
- Probajte tanh, ali ne očekujte puno
- Ne koristite sigmoid

Predprocesiranje podataka

Data Preprocessing

Korak 1: Predprocesirati podatke




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

(X [NxD] je matrica podataka, svaki primer je jedan red)

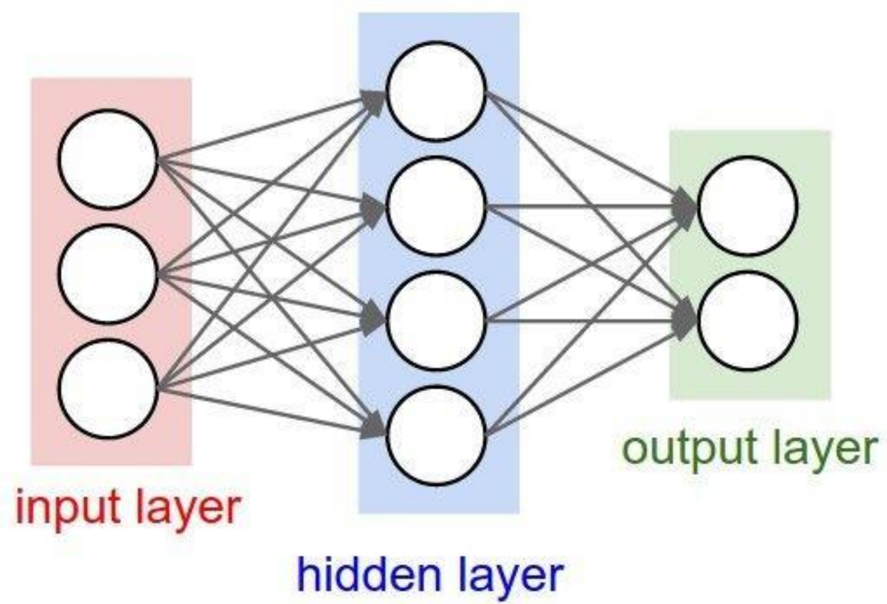
TLDR: U praksi, za slike: samo centriranje

Npr. CIFAR-10 skup podataka sa [32,32,3] slikama

- Oduzimamo srednju sliku (npr. AlexNet)
  (za svaki piksel srednja vrednost po celom skupu podataka = srednja slika [32,32,3])
- Oduzimamo srednju sliku po svakoj boji (npr. VGGNet) (srednja vrednost svih piksela u celom skupu podataka za svaku boju R, G, B posebno = 3 broja)

Inicijalizacija težina

- Šta se dešava kada je W=0 na početku?



- Prva ideja: **Mali slučajno odabrani brojevi** (Gausijana sa srednjom vrednosti 0 i standardnom devijacijom od 1e-2)

```
W = 0.01* np.random.randn(D,H)
```

- Prva ideja: **Mali slučajno odabrani brojevi** (Gausijana sa srednjom vrednosti 0 i standardnom devijacijom od 1e-2)

```
W = 0.01* np.random.randn(D,H)
```

Radi dobro za manje mreže. Kod većih mreža aktivacije brzo postaju previše male da bi backprop mogao da bilo šta uradi.

Pogledajmo statistike aktivacija za jednu mrežu

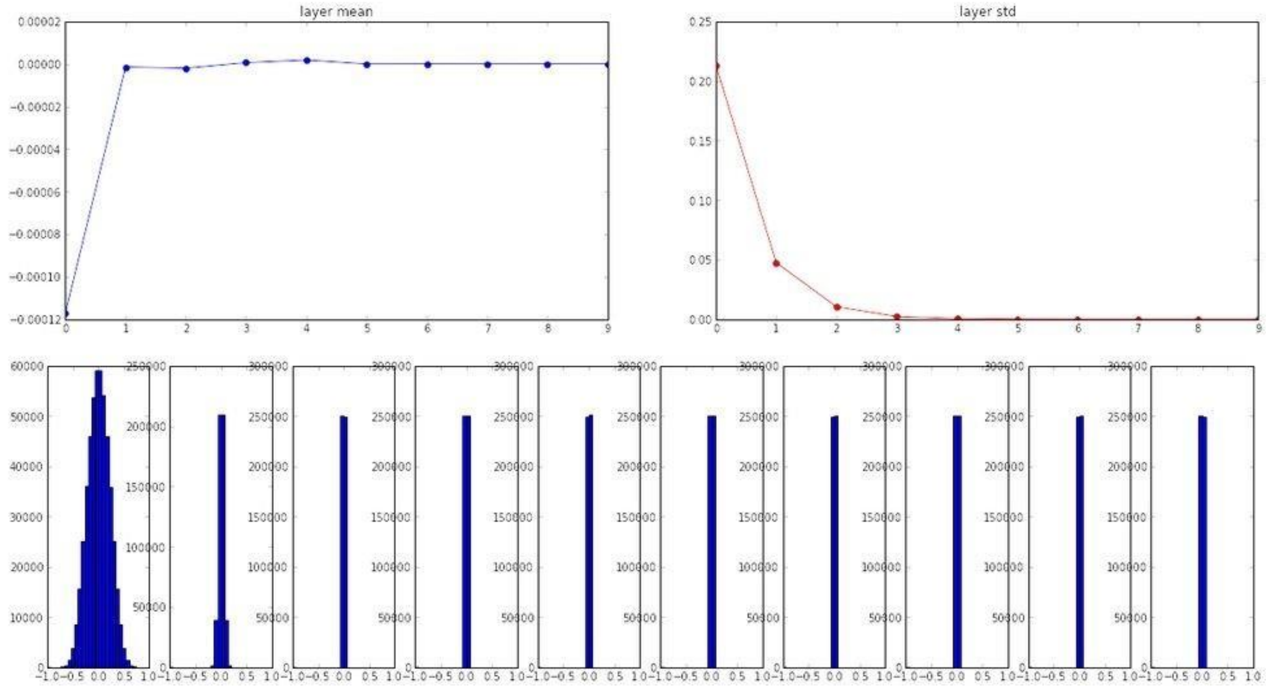
Npr. Mreža od 10 slojeva sa 500 neurona u svakom sloju, koristi se tanh i inicijializacija kao na prethodnom slajdu

```python
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

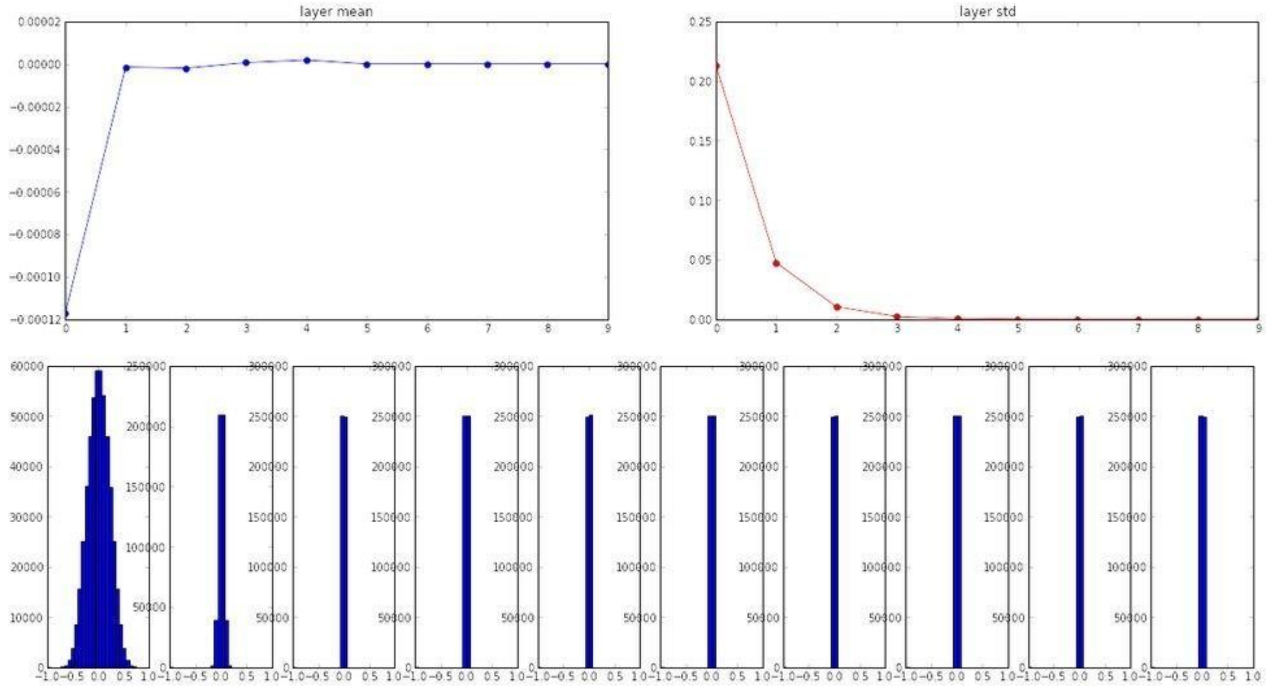
```python
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```python
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



Sve aktivacije postaju 0!

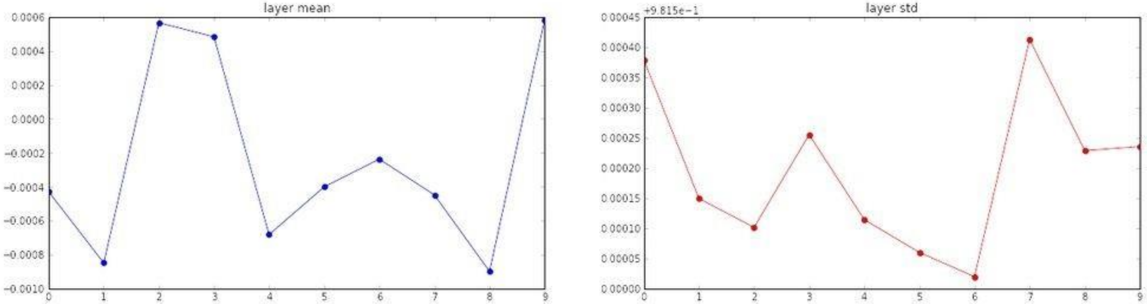
Šta se onda dešava sa gradijentima?

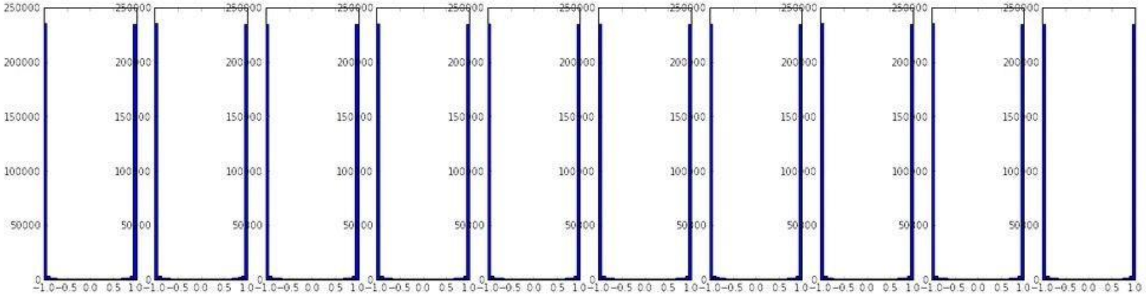
Pomoć: razmislite o propagaciji unazad kroz W*X deo.

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

*1.0 umesto *0.01

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```



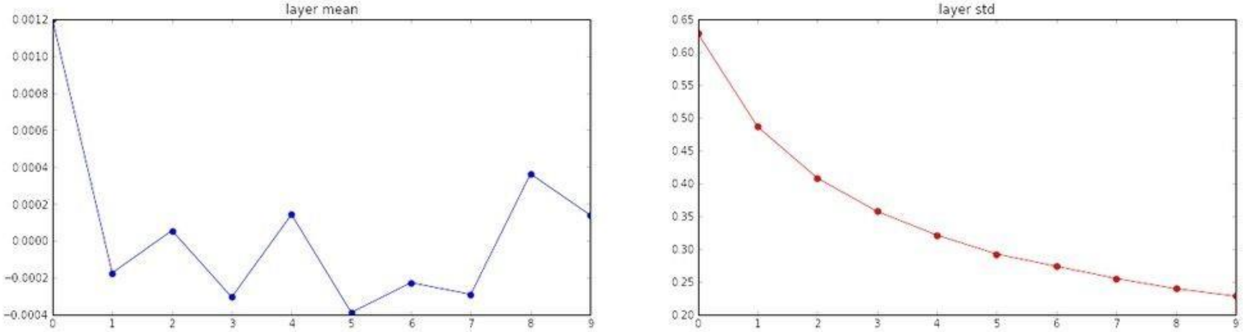


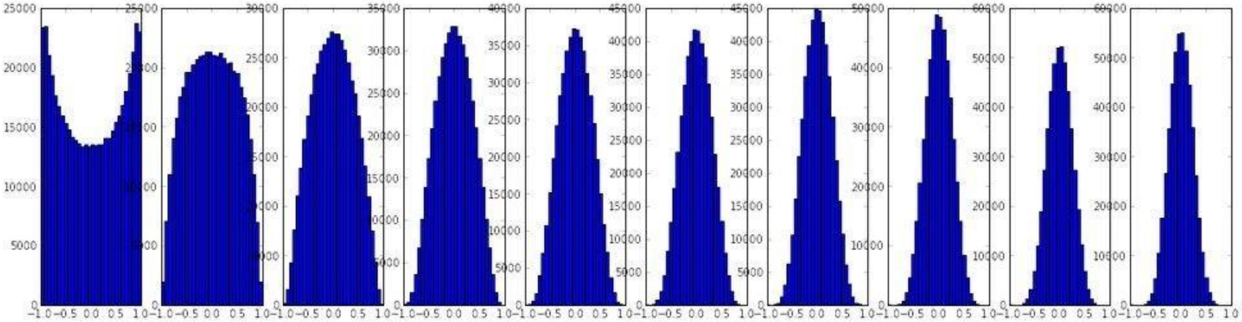
Skoro svi neuroni su saturirani, vrednosti su -1 ili 1. Gradijenti će svi biti 0.

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

“Xavier inicijalizacija”
[Glorot et al., 2010]



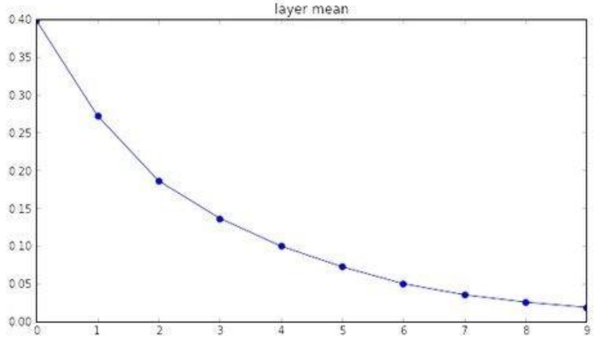


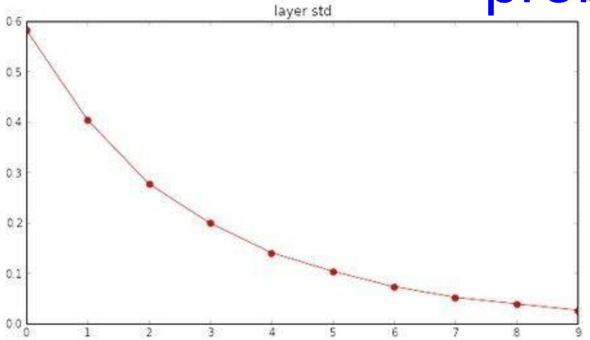
Razumna inicijalizacija.
Rešava problem koji smo imali u prethodnim slajdovima.
Ne radi dobro sa ReLU

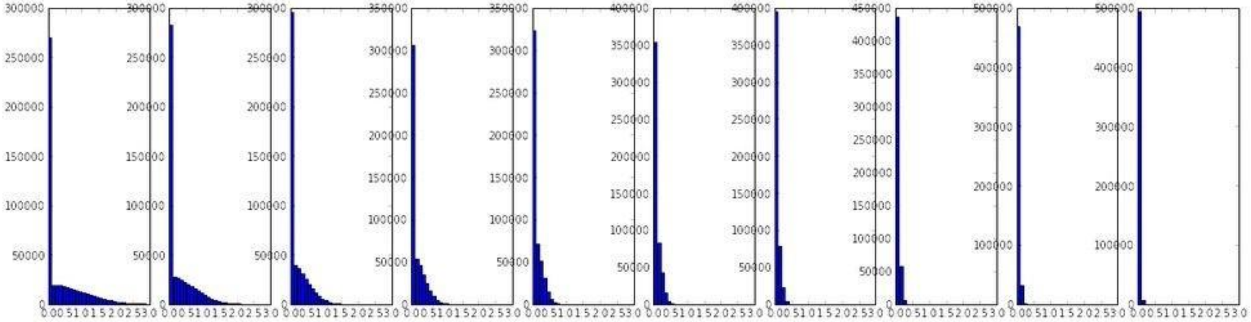
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```python
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

Kada koristimo ReLU opet imamo stari problem



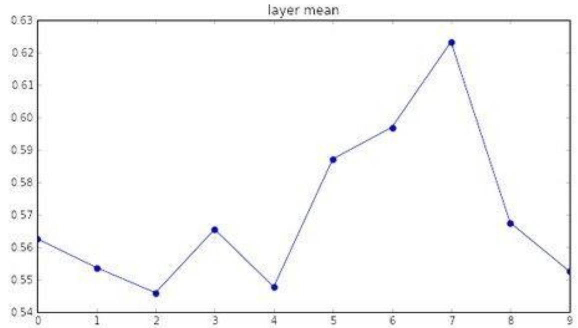


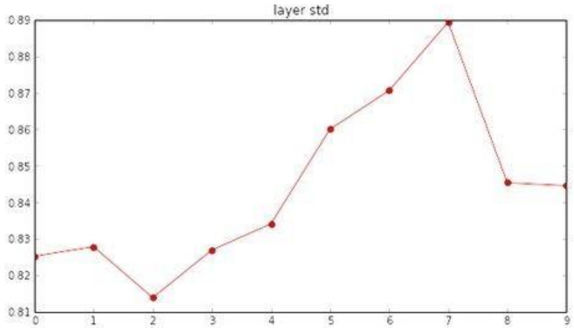


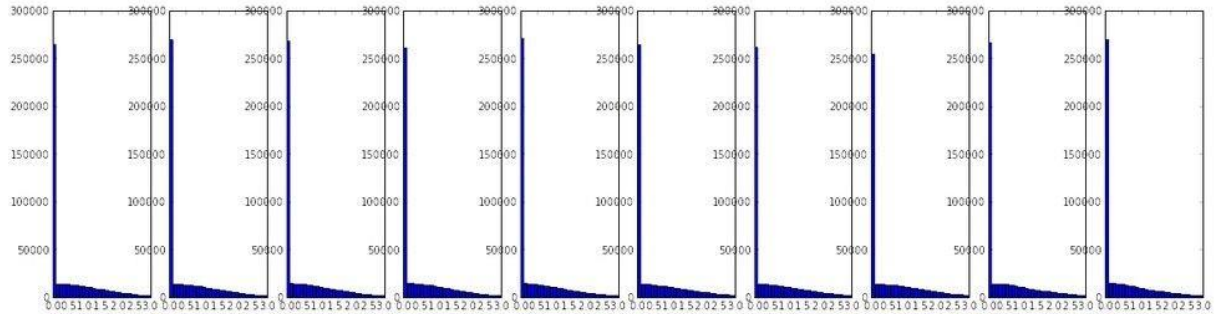
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```python
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

He et al., 2015
(autori su samo dodali /2 gore)



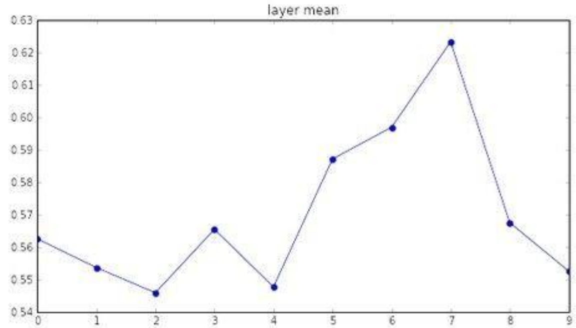


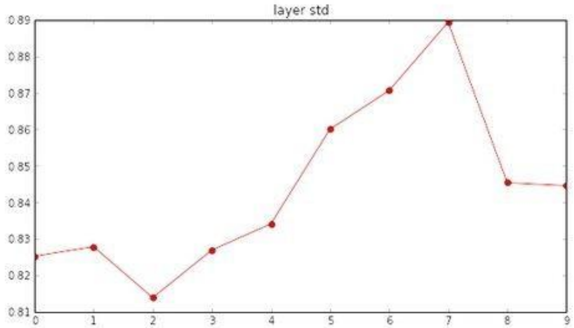


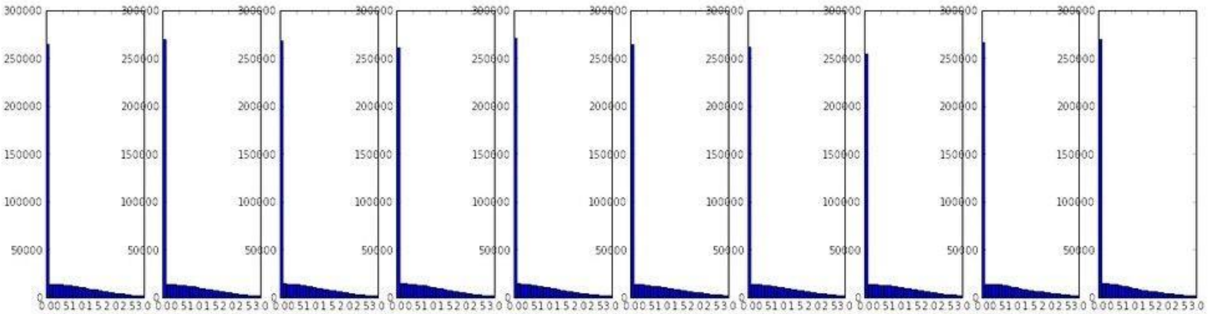
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

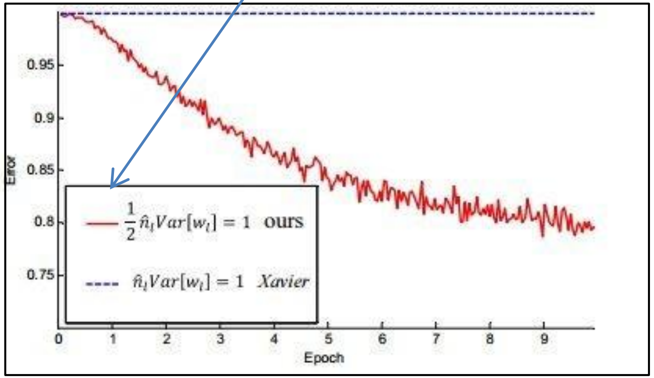
He et al., 2015
(note additional /2)







ReLU sa Xavier ima grešku 100%, a ako dodamo /2 sve radi



Inicijalizacija težina (kao i sve kod Deep Learning) je aktivna oblast istraživanja...

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

...

Batch Normalizacija

[Ioffe and Szegedy, 2015]

“Želite da aktivacije budu Gausijane? Transformišite ih u Gausijane .”

recimo da sačuvamo aktivacije jednog podskupa (*batch*) za jedan sloj. Da bi od njih dobili Gausijanu sa sredjom vrednosti 0 i st.dev. 1 treba da primenimo:

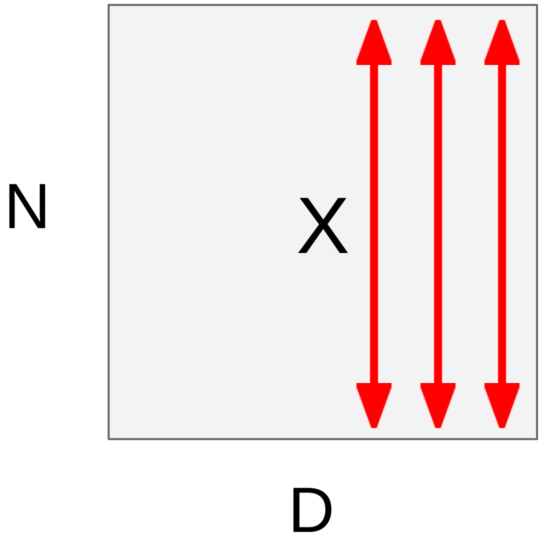
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

ovo je diferencijabilna fukcija, backprop neće imati problema...

Batch Normalizacija

[Ioffe and Szegedy, 2015]

“Želite da aktivacije budu Gausijane? Transformišite ih u Gausijane .”



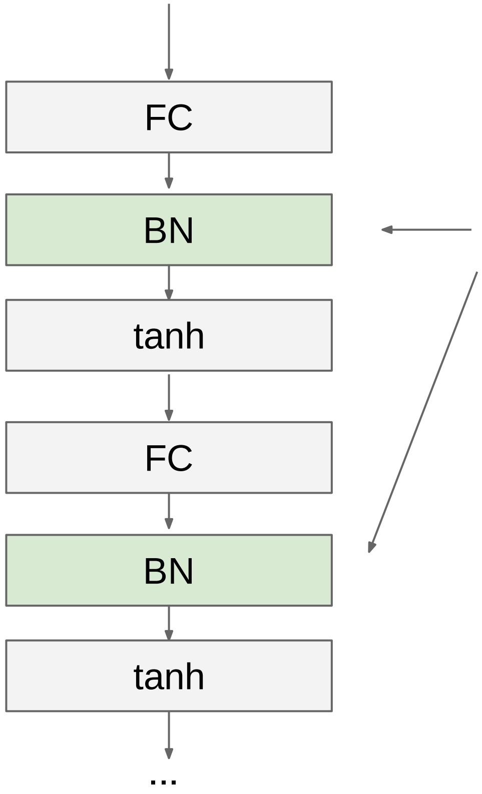
1. izračunavamo srednju vrednost i st. dev. po svakoj dimenziji (atributu) za N primera u podskupu.

2. Normalizujemo

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalizacija

[Ioffe and Szegedy, 2015]



Obično se ubacuje posle svakog potpuno povezanog (ili konvolutivnog) sloja pre funkcije aktivacije

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalizacija

[Ioffe and Szegedy, 2015]

Normalizacija:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Dozvoljavamo mreži da „zgnječi“ i pomeri aktivacije ako želi:

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}$$

Mreža čak može i da nauči:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

i time poništi Batch Norm.

Batch Normalizacija

[Ioffe and Szegedy, 2015]

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

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

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

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

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

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

- Poboljšava tok gradijenata (*gradient flow*) kroz mrežu
- Dozvoljava veće korake učenja
- Mreža više nije toliko osetljiva na inicijalizacije težina

Batch Normalizacija

[Ioffe and Szegedy, 2015]

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

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

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

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

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

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

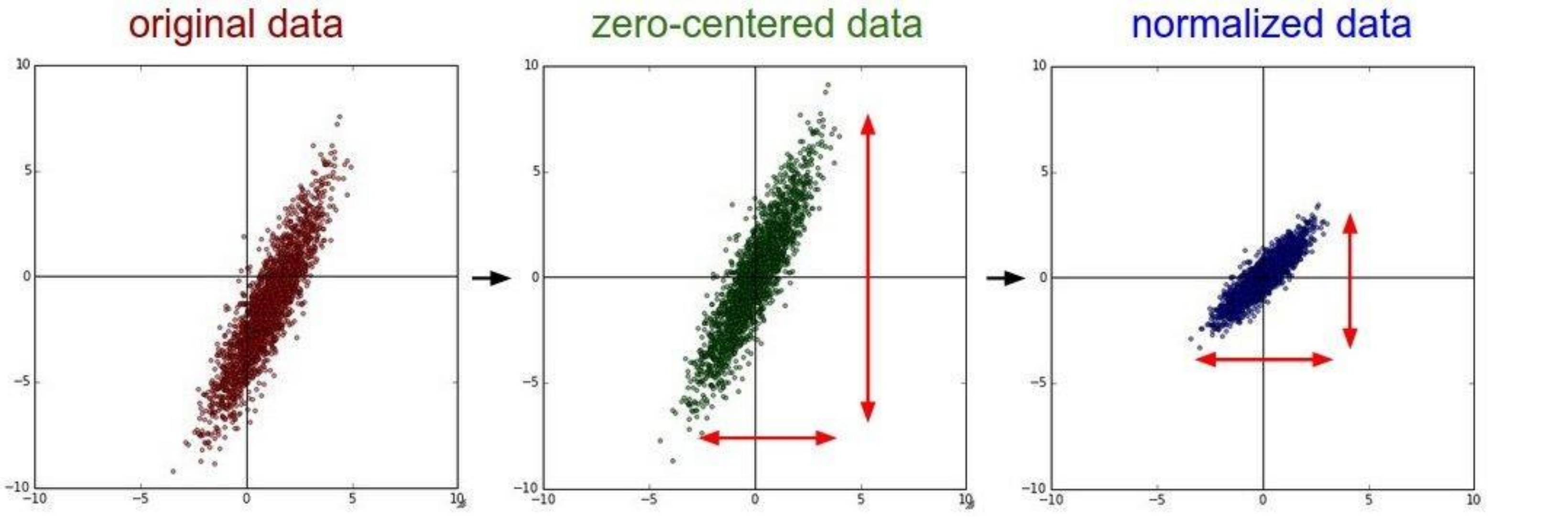
Napomena: kada primenjujemo mrežu na test podatke BatchNorm slojevi funkcionišu drugačije:

Srednja vrednost i st. dev se sad ne izračunavaju. Koriste se one koje su dobijene tokom obučavanja. – mreža je „naučila“ na te vrednosti, dok bi vrednosti iz test podataka pokvarile predikcije

(npr. možemo uzeti prosek srednjih vrednosti i st.dev. za jedan BN sloj tokom obučavanja)

Kako Pratiti Proces Obučavanja
Babysitting the Learning Process

Korak 1: Predprocesirati podatke



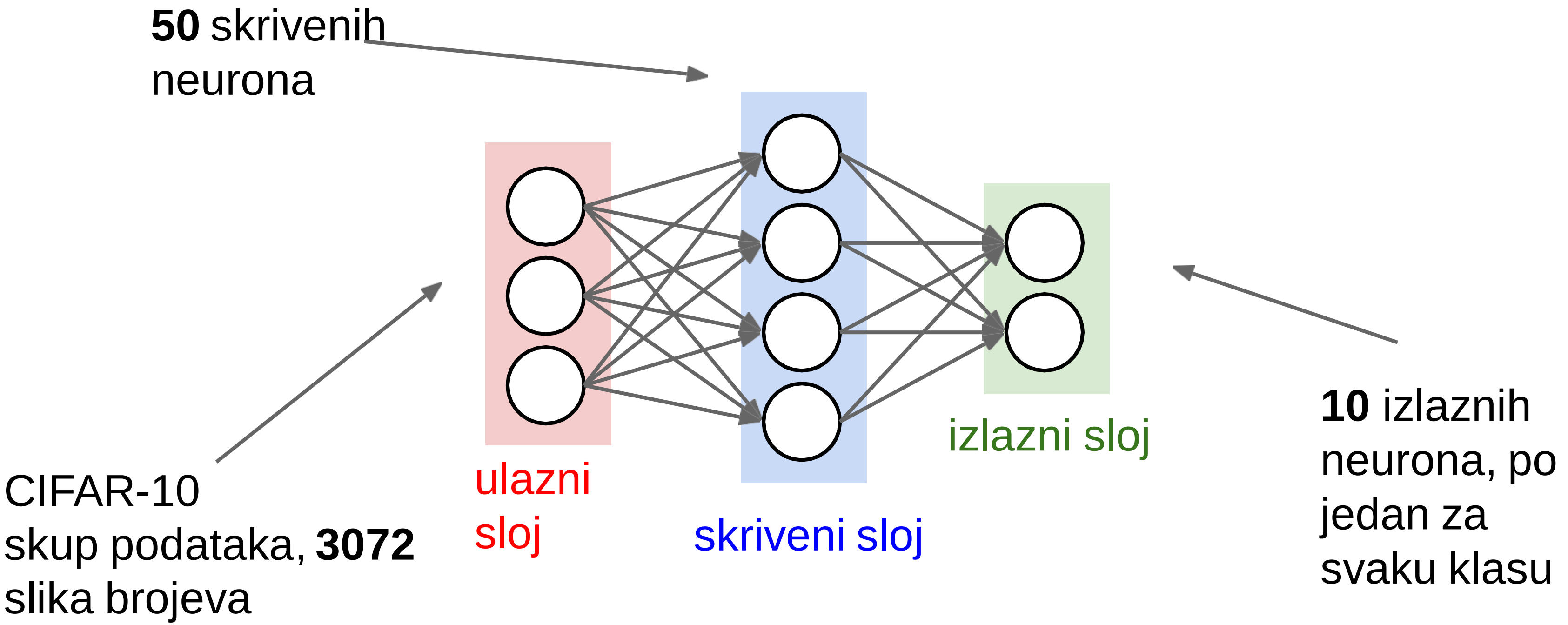

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

(X [NxD] je matrica podataka, svaki primer je jedan red)

Korak 2: Odabrati Arhitekturu Mreže:

recimo da krenemo sa mrežom sa jednim skrivenim slojem od 50 neurona:



Proveravamo da li smo dobro implementirali funkciju greške: puštamo na ulaz ceo skup podataka, ali na ne-obučenu mrežu

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

Treba da dobijemo skoro maksimalnu vrednost koja je moguća za tu funkciju greške.

```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

isključimo regularizaciju

```
2.30261216167
```

greška ~2.3.
“tačna“ za 10
klasa - softmax

metod vraća ukupnu grešku i gradijente
za sve parametre

Proveravamo da li smo dobro implementirali funkciju greške: puštamo na ulaz ceo skup podataka, ali na ne-obučenu mrežu

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

Treba da dobijemo skoro maksimalnu vrednost koja je moguća za tu funkciju greške.

```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden_size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

povećamo regularizaciju

```
3.06859716482
```

greška je porasla, što je dobro

Krećemo sa obučavanjem…

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

Savet: Za početak probajte da overfitujete (dobijete training grešku od skoro 0%) mali podskup podataka. To je još jedan znak da je implementacija dobra.

Kod gore:

- uzimamo prvih 20 primera iz CIFAR-10
- isključimo regularizaciju (reg = 0.0)
- koristimo običan SGD

Krećemo sa obučavanjem…

Savet: Za početak probajte da overfitujete (dobijete training grešku od 0%) mali podskup podataka.

Vrlo mala greška, train tačnost je 1.00 (100%), što je dobro!

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Krećemo sa obučavanjem…

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Krećemo sa obučavanjem…

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Greška jedva pada

Krećemo sa obučavanjem…

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

greška ne pada:
korak učenja je previše mali

Greška jedva pada:
korak učenja je previše mali

Krećemo sa obučavanjem…

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

greška ne pada:
korak učenja je previše mali

Greška jedva pada:
korak učenja je previše mali

Primetite da train/val tačnost stiže do 20%, otkud sad to? (pomoć: ovo je softmax greška)

Krećemo sa obučavanjem…

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

Šta ako povećamo korak učenja na 10^6? Šta bi moglo loše da se dogodi ☺?

Krećemo sa obučavanjem…

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

greška ne pada: korak učenja je previše mali

greška eksplodira: korak učenja je previše veliki

NaN vrednosti za grešku vrlo verovatno znače da je korak učenja preveliki...

Krećemo sa obučavanjem…

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

3e-3 je i dalje previše. Greška eksplodira….

greška ne pada:
korak učenja je previše mali
greška eksplodira: korak učenja je previše veliki

=> Na ovaj način došli bi do nekog raspona koji nam se čini dobar i u kome bi trebali da štelujemo korak učenja fino. U ovom slučaju je to [1e-3 … 1e-5]

Krećemo sa obučavanjem…

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

Počinjemo sa malom regularizacijom i tražimo korak učenja na osnovu koga greška kreće da pada

greška ne pada:
korak učenja je previše mali
greška eksplodira: korak učenja je previše veliki

Napomena: val vrednost možemo dobiti pomoću unakrsne validacije ili validacionog skupa.
Unakrsna validacija je realnija, ali spora
Sa validacionim skupom izračunavanje greške je brže, ali greška može biti nerealna.

Optimizacija Hiperparametara

Hyperparameter Optimization

Strategija

grubo -> fino u fazama

Prva faza: samo par epoha za svaku vrednost parametara, dok ne dobijemo grubu ideju koji rasponi su dobri
Druga faza: više epoha, fino štelovanje u okviru raspona iz prve faze
… (faze možemo i ponoviti po potrebi)

Savet:
Ako je greška > 3 * greške od koje smo krenuli,
prekinuti obučavanje i promeniti nešto

Na primer: radimo grubu pretragu 5 epoha

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=5, reg=reg,
                                   update='momentum', learning_rate_decay=0.9,
                                   sample_batches = True, batch_size = 100,
                                   learning_rate=lr, verbose=False)
```

```
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)
```

dobre vrednosti

Sada radimo finiju pretragu...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

podešavamo raspon

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
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)
```

53% - relativno dobra tačnost za 2-slojnu mrežu sa 50 neurona u skrivenom sloju.

Sada radimo finiju pretragu...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

podešavamo raspon

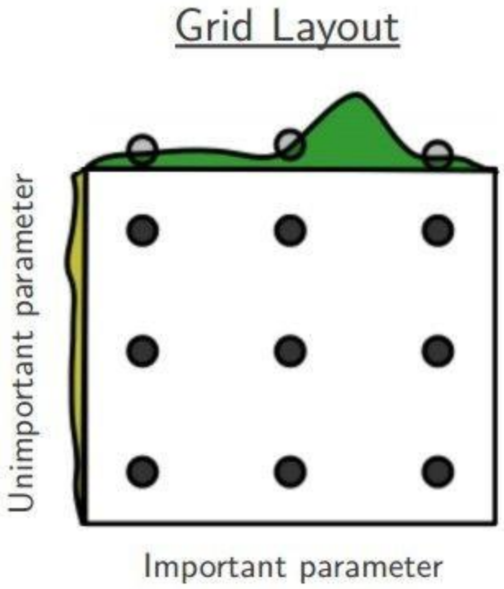
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

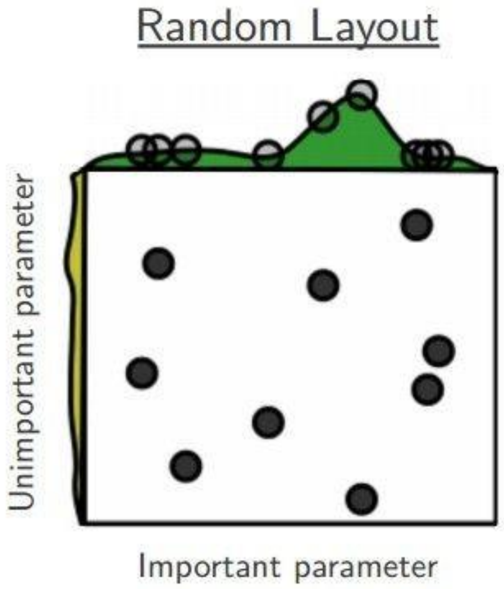
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, 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)
```

53% - relativno dobra tačnost za 2-slojnu mrežu sa 50 neurona u skrivenom sloju.

Ovaj rezultat za val od 0.531.. bi trebalo da nas brine. Zašto?

Random Pretraga vs. Pretraga po Mreži (*Grid Search*)





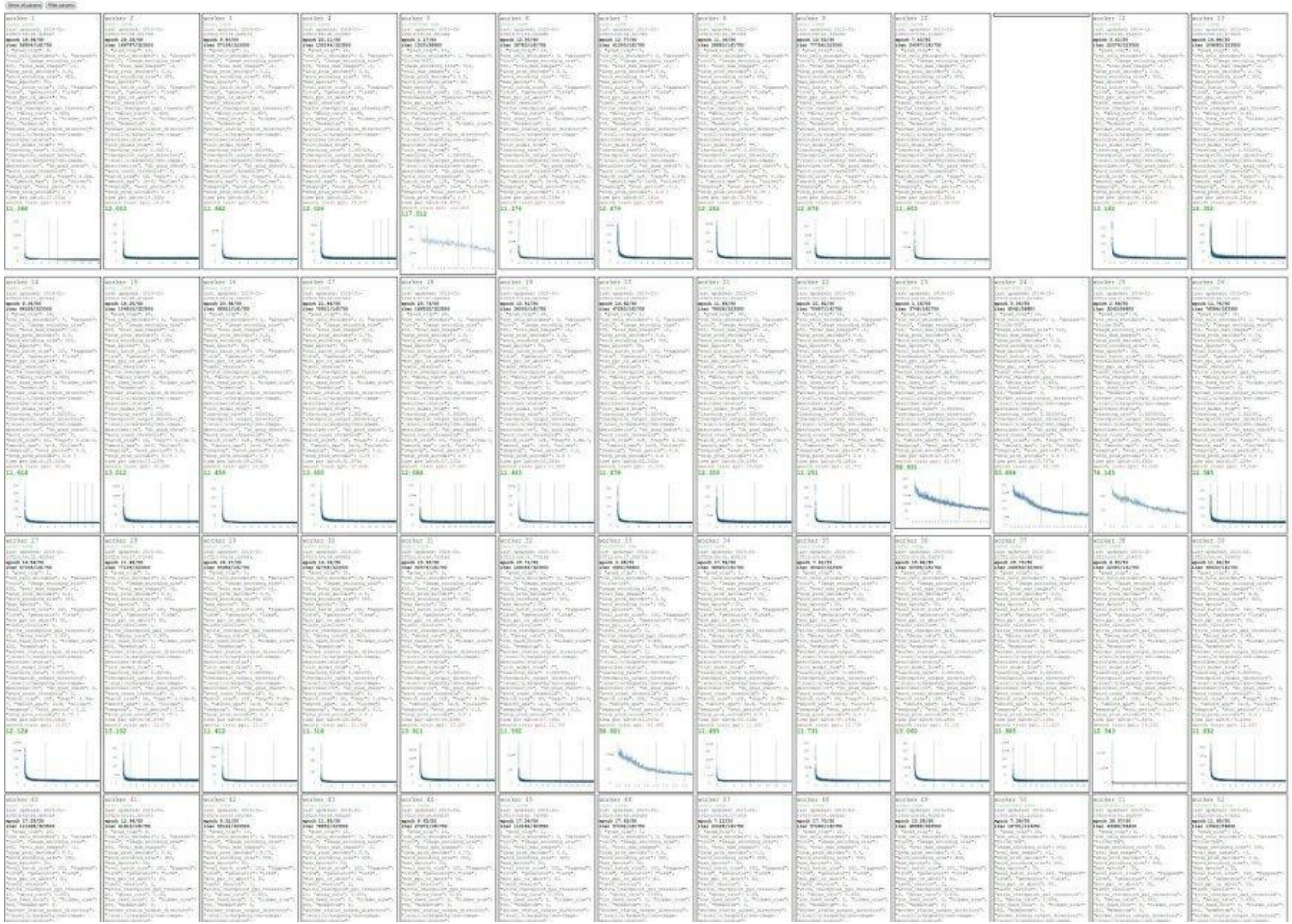
Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Hiperparametri sa kojima se igramo:

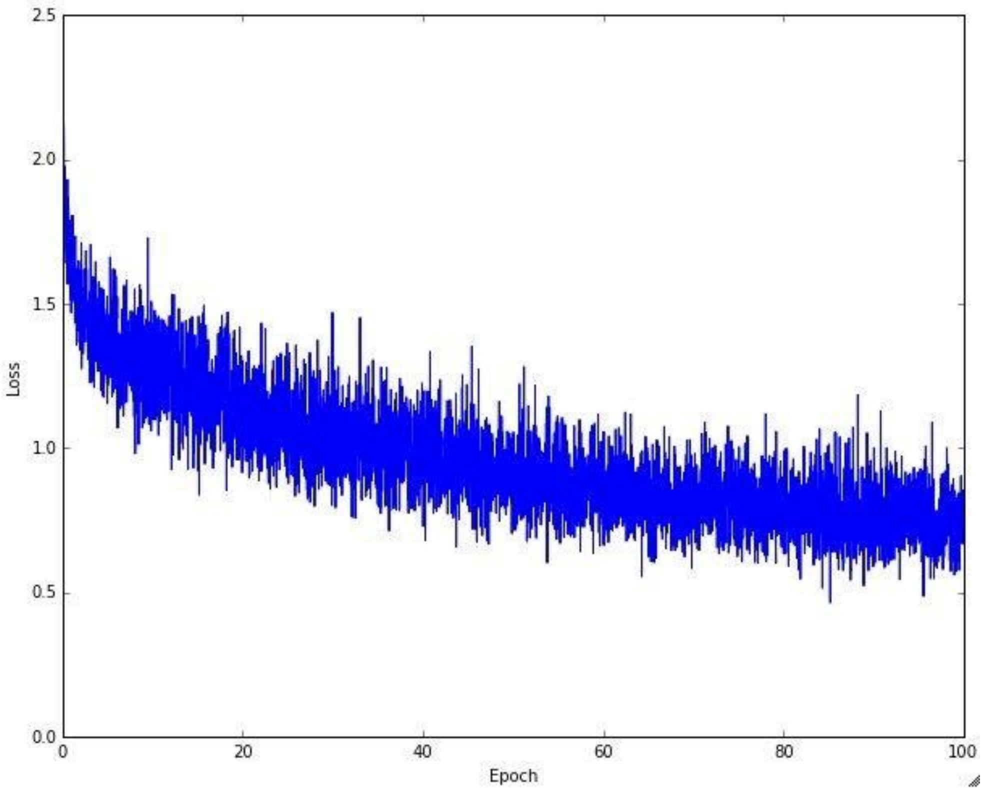
- arhitektrura mreže
- korak učenja, kako ga menjamo kroz vreme, način promene parametara
- regularizacija (L2/Dropout)
- Neke od ovih pojmova tek treba da učimo

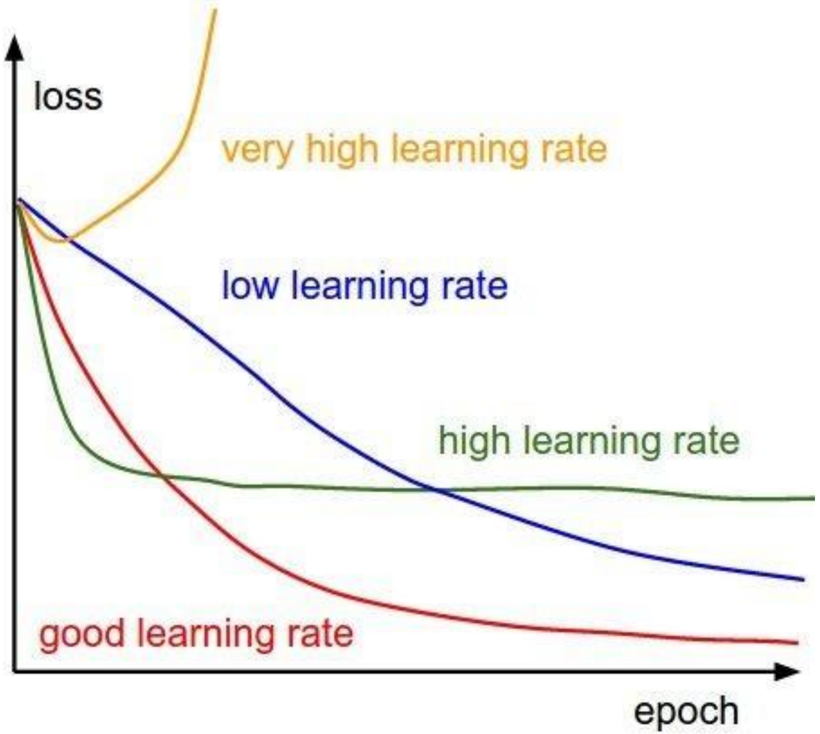
ovako izgleda štelovanje parametra sa ciljem da smanjimo grešku

Šta bi bilo da imamo klaster od 70 mašina za obučavanje ☺

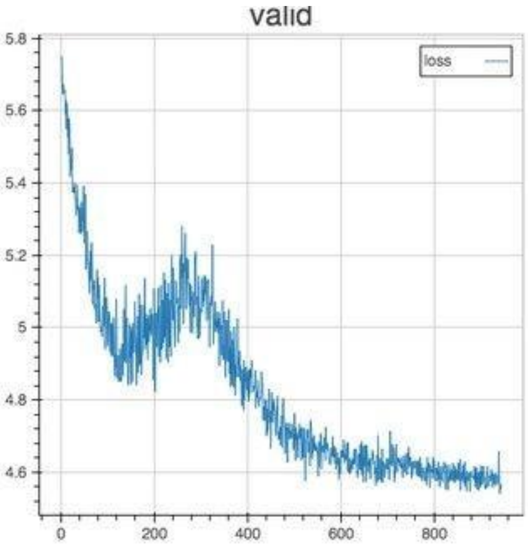


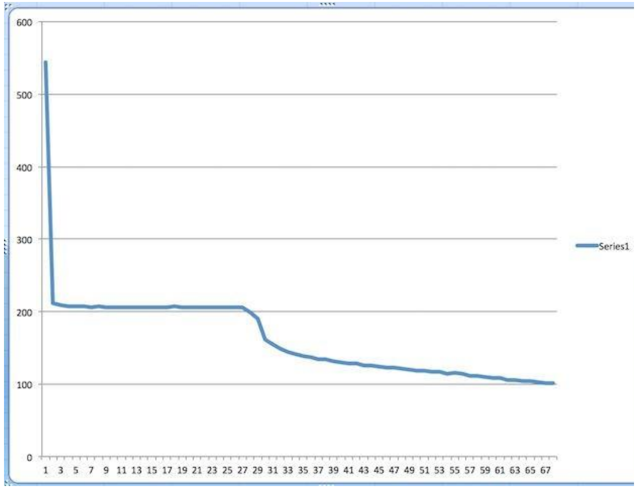
Obavezno vizualizujemo i pratimo krivu funkcije greške

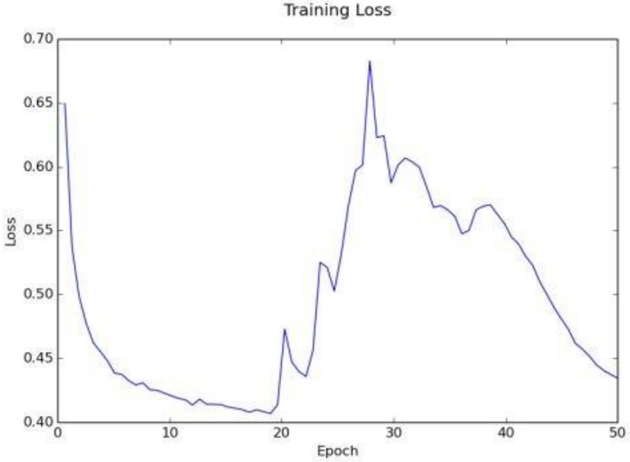




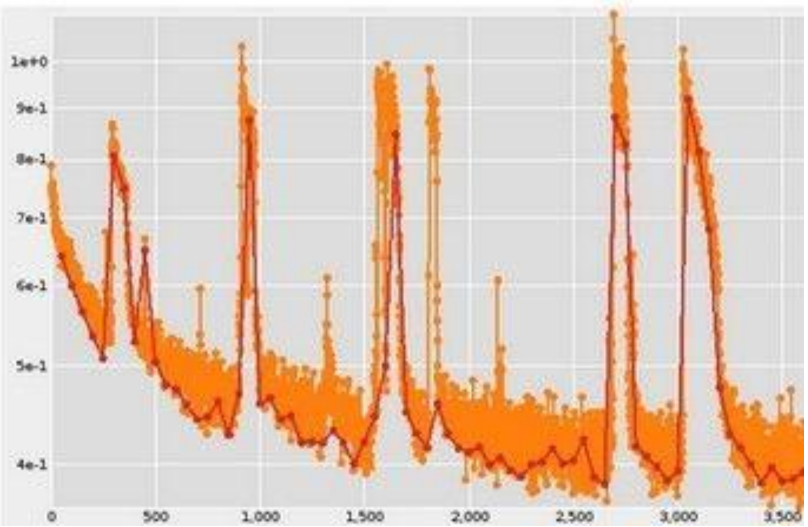
lossfunctions.tumblr.com Primeri funkcija greške

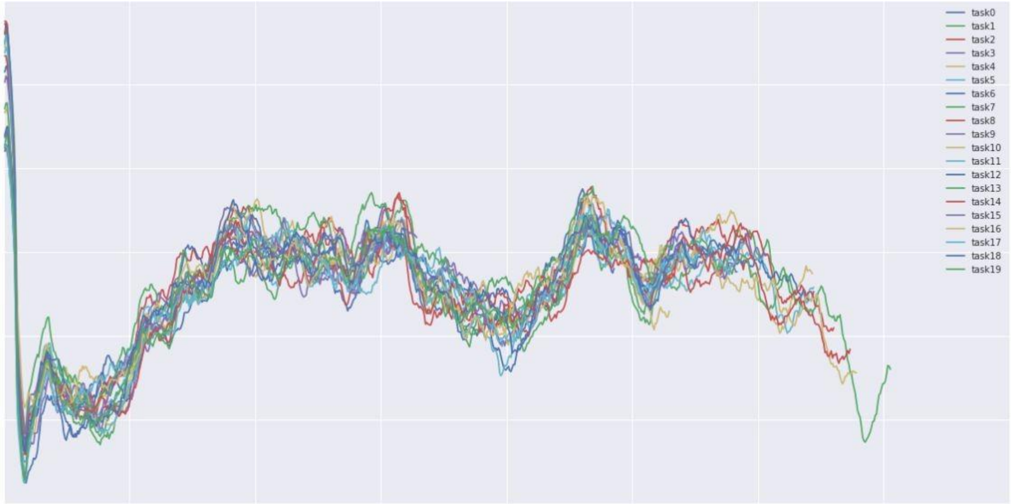


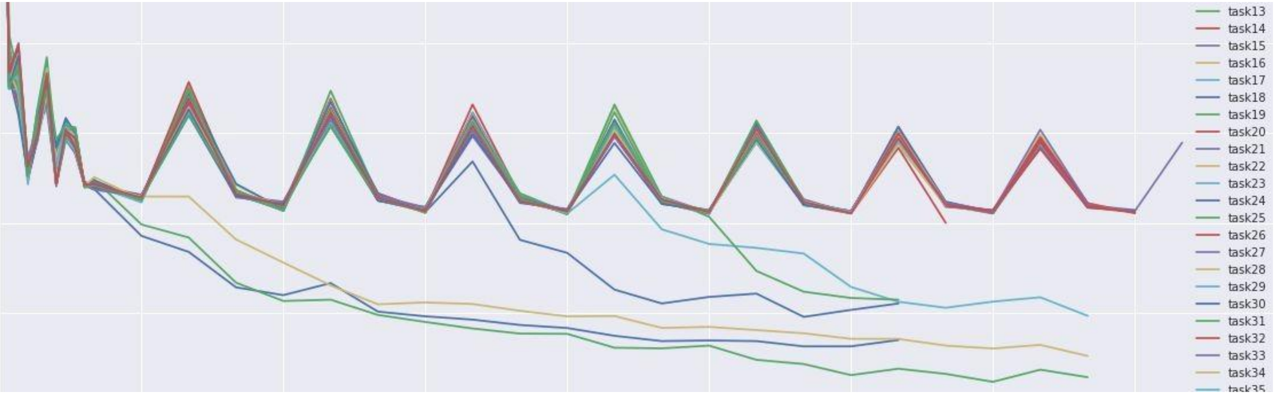


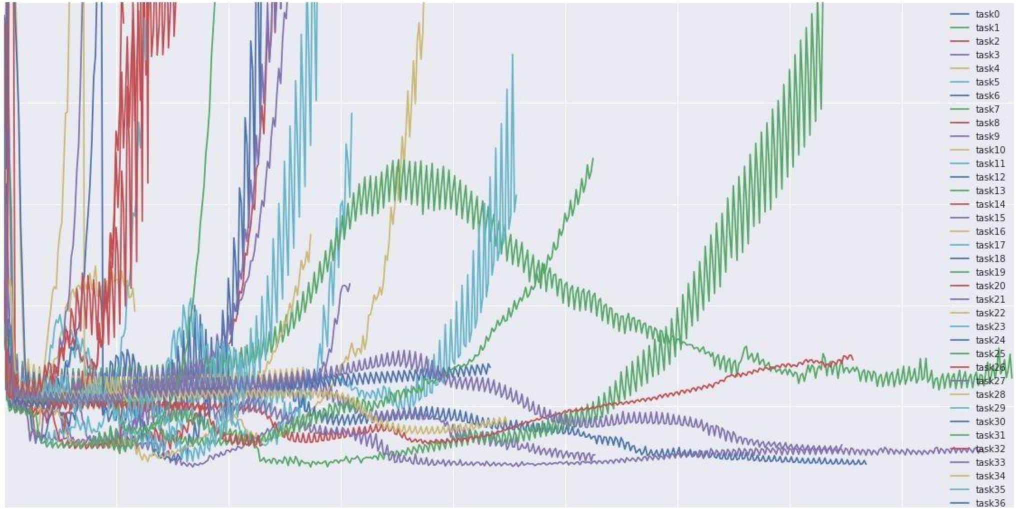


lossfunctions.tumblr.com



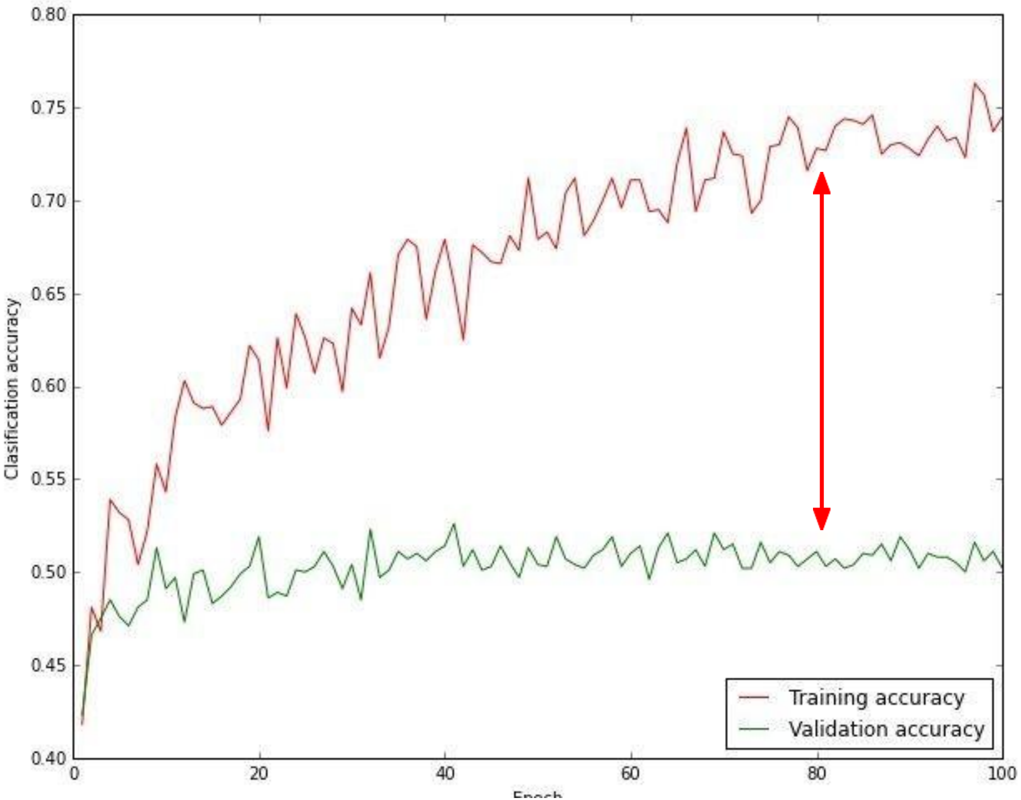






lossfunctions.tumblr.com

Obevezno vizualizujemo i pratimo tačnost:



velika razlika = overfitting

=> povećati uticaj regularizacije (lambda)?

nema razlike tj. i training accuracy ima malu vrednost

=> povećati kapacitet modela (npr. dodavanjem još neurona ili slojeva)?

Rezime

TLDRs

Danas smo se bavili sa:

- Aktivacionim Funkcijama (koristite ReLU)
- Predprocesiranjem Podataka (slike: oduzeti srednju sliku)
- Inicijalizacija Težina (koristite Xavier inicijalizaciju sa /2)
- Batch Normalizacije (koristite)
- Praćenje Procesa Učenja
- Optimizacija Hiper-parametara
  (vrednosti treba slučajno birati, iz log prostora, mada možete uvek probati i bez log)

Šta treba da radimo

- Razne načine promene parametara
- Da se više bavimo korakom učenja
- Provera Gradijenata
- Regularizacija (Dropout)
- Evaluacija (Ansambli)