# Feed Forward Neural Networks

ML4SE

Denis Litvinov

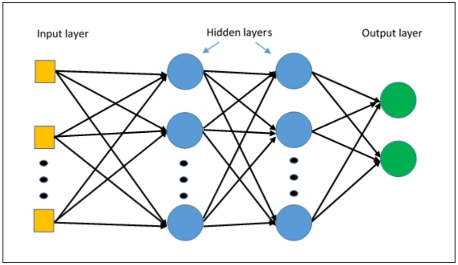
October 6, 2021

## Table of Contents

## General Architecture

NN as a composition of functions

$$F(x) = f_{w_n} \circ f_{w_{n-1}} \circ .. f_{w_1}(x)$$



## Composition of linear functions

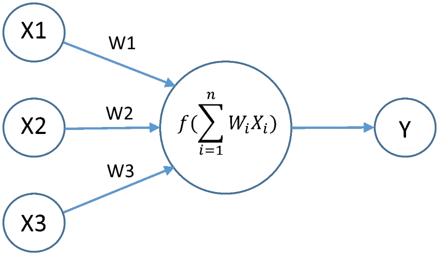
$$F(x) = XW_1 W_2$$

where $X \in R^{NxD_1}$ - features
$W_1 \in R^{D_1 x D_2}$, $W_2 \in R^{D_2 x K}$ - weight matrices
$K$ - number of classes

## Neuron

$$y = \sum_{i=1}^{N} f(w_j i x_i + b)$$

where $f$ - some non-linear activation function
$w_i$ - learnable weights
$b$ - learnable bias, usually incorporated into X
$y$ - output of neuron
$x$ - input of neuron

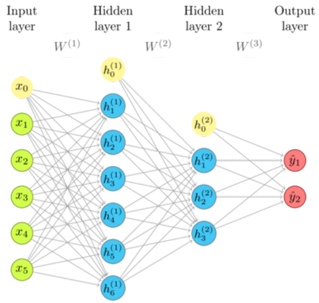


## Dense Layers

It's more convenient to express the same thing in vector form

$$Y = f(XW)$$

where $X \in R^{NxD_1}$ - input of layer
$Y \in R^{NxD_2}$ - output of layer
$W \in R^{D_1 xD_2}$ - learnable weight matrix



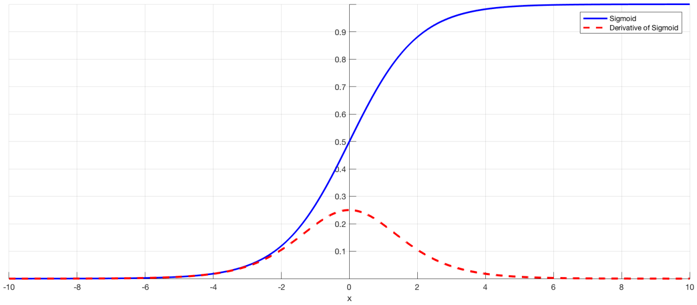
## Activation Functions

Why we do not use linear activations?
Activation functions supposed to be nice in the sense of gradient properties.

## Sigmoid

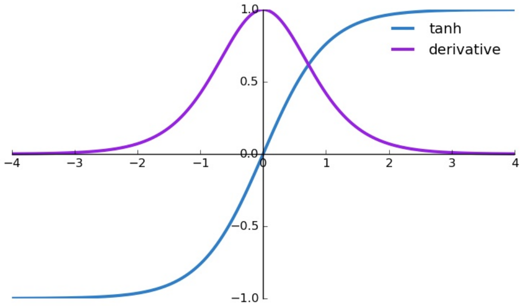
$$\sigma(z) = \frac{1}{1 + \exp^{-z}}$$

- vanishing gradient
- bad output distribution



## Tanh

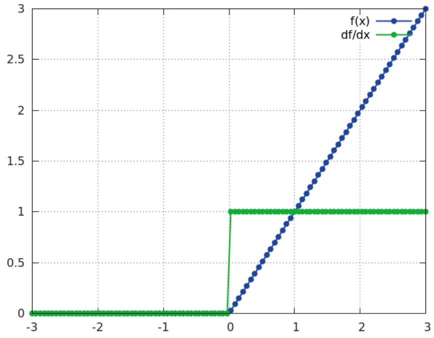
- vanishing gradient



## RELU

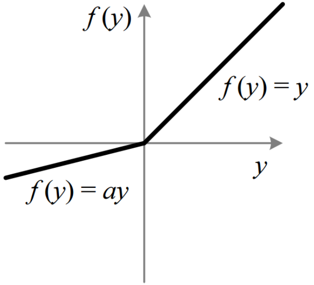
$$RELU(z) = \max(0, z)$$

- dead neurons if $z < 0$



## PRELU

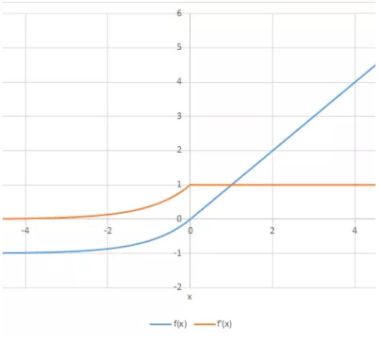
$$PRELU(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha z, & \text{if } z < 0 \end{cases} \qquad (1)$$



## ELU

$$ELU(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha(\exp^z - 1), & \text{if } z < 0 \end{cases} \qquad (2)$$

- little longer computation than RELU



## Weight initialization

As we train out neural network with gradient descent, it is important to have good initial point to start. Usually use use:

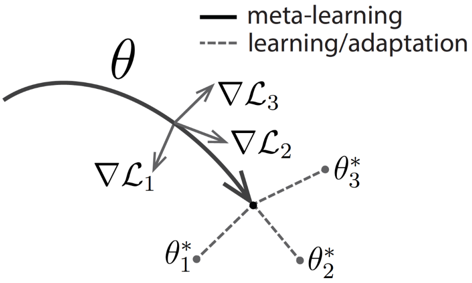
1. Uniform distribution in $[-d, d]$
2. Normal distribution $N(0, \sigma^2)$
3. Xavier distribution (discuss later, in CNN)

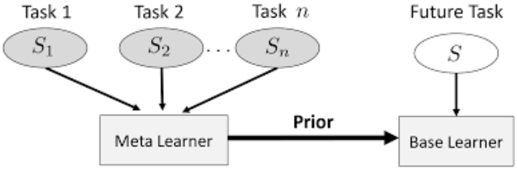
- Why we use distributions centered around zero?
- How it is connected with activation functions?
- shared weights

## Transfer Learning

1. a big model is trained on a large dataset
2. learned weights from the model are used as a initialization for another, usually smaller, downstream task

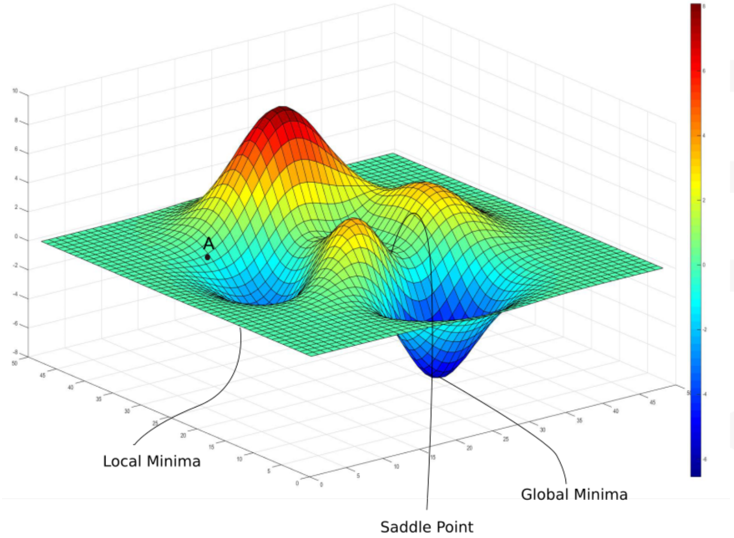
## Meta Learning





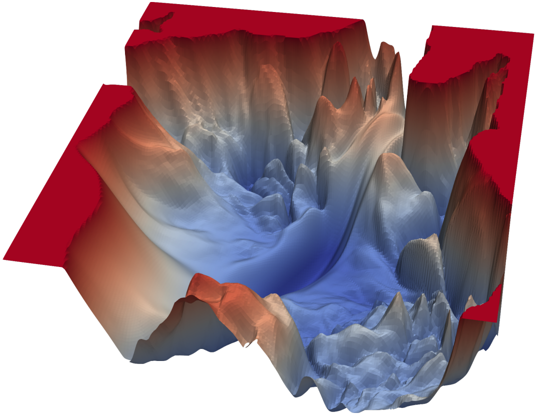
## Loss surface

### Many local minima



## Loss surface

Wide local minimum



## Vanilla SGD

```
$\theta_0 \leftarrow$ init
for **random** batch on step $t$ = 1..max_iter:
```

$$\theta_t = \theta_{t-1} - \alpha \nabla_\theta J(\theta_{t-1})$$

$J$ - loss function
$\theta_t$ - learnable parameters at step $t$
$\alpha$ - learning rate

- good theoretic properties
- slow convergence

## SGD with momentum

```
$\theta_0 \leftarrow$ init
$m_0 \leftarrow 0$
for **random** batch on step $t$ = 1..max_iter:
```

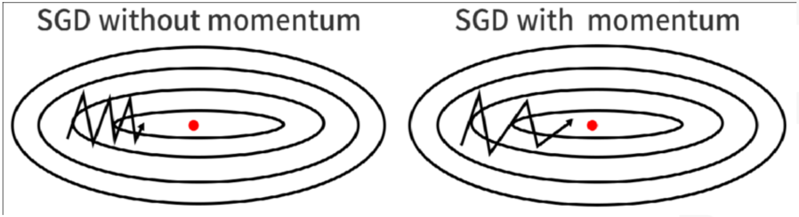
$$m_t = \beta m_{t-1} + (1 - \beta)\nabla_\theta J(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - \alpha m_t$$

where $m_t$ - accumulated gradient at step $t$
$\beta$ - momentum parameter

## SGD with momentum

- Momentum cancels moves in "random" directions from stochastic nature of SGD
- Momentum   inertia



## RmsProp

$\theta_0 \leftarrow$ init $v_0 \leftarrow 0$
for random batch on step $t$ = 1..max_iter:

$$g_t = \nabla_\theta J(\theta_{t-1})$$
$$v_t = \beta v_{t-1} + (1-\beta) g_t^2$$
$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{v_t} + \epsilon} g_t$$

where $v_t$ - accumulated squared components of gradient
$\beta$ - parameter
$\epsilon << 1$ - to prevent division by zero

- gradient direction carries more information than its norm
- adjust gradient step size

## Adam

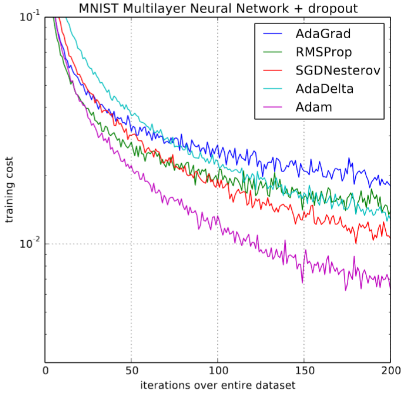
```
$\theta_0 \leftarrow$ init $v_0 \leftarrow 0$
$m_0 \leftarrow 0$
for random batch on step $t$ = 1..max_iter:
```

$$g_t = \nabla_\theta J(\theta_{t-1})$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

where $m_t$ - accumulated momentum
$v_t$ - accumulated squared components of gradient
$\beta_1, \beta_2$ - parameters
$\epsilon << 1$ - to prevent division by zero

## Adam

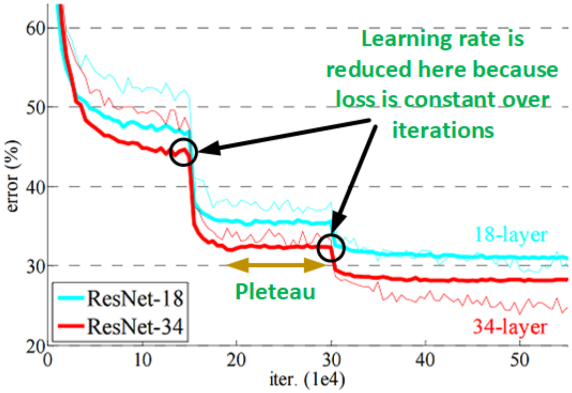
- essentially SGD with momentum + RmsProp
- corrections for $\hat{m}_t, \hat{v}_t$ are to make first optimization steps more stable. Because the calculation of $m_t, v_t$ can be seen as geometric series



## Reduce On Plateau

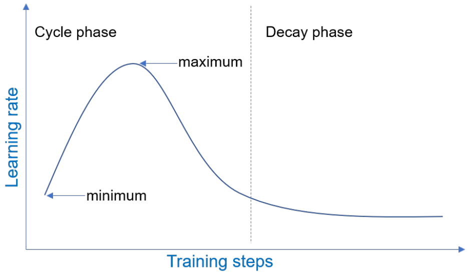
Reduce learning rate by some factor if loss is not decreasing enough.

- we can’t converge to exact local minima
- unfortunately, we increase "sensitivity" to more narrow local minima.



# Cycle LR

If weight initialization wasn't good enough, we try to increase learning rate at first few steps in a hope to jump into a better local minumum.



## Regularization

Most popular:

1. $L_2$ norm regularization through weight decay
2. Early stopping
3. Data augmentation. Create new samples from the same domain to increase size of your dataset. Remember generalization bounds.
4. Dropout. Drop random nodes in a layer with probability $p$
5. Batch Normalization

## Dropout

There are 2 interpretations for dropout:

- "Bagging" over neural networks
- Avoid feature coadaptation

Difference between bagging and dropout:

$$p(y|x) = \frac{1}{K}\sum_{i=1}^{K} p_i(y|x)$$

for bagging

$$p(y|x) = \sum_{\mu} p(\mu)p(y|x,\mu)$$

for dropout, where $\mu$ is mask on weights.
There is an exponential number of masks for fixed number of weights, that makes dropout more effective than explicit bagging.

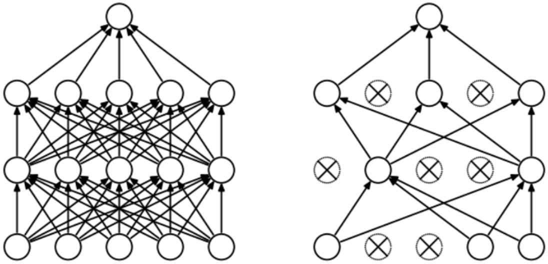
## Dropout

**On training:** On each batch randomly remove neurons in the previous layer with probability *p*.

**On inference:**

Ideally, sample all $2^n$ dropped-out networks and average predictions. In practice, approximate by using the full network with each node's output weighted by a factor of $1 - p$, so the expected value of the output of any node is the same as in the training stages.

=> Although it effectively generates $2^n$ neural nets, but at test time only a single network needs to be tested.



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

## BatchNorm

**On training:**
on every batch $t$:

$$\mu_t = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_t^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_t)$$

$$\hat{x}_i = \frac{x_i - \mu_t}{\sigma_t + \epsilon}$$

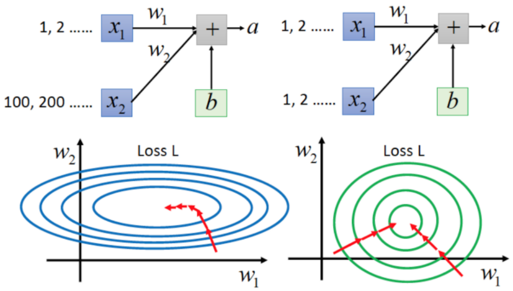
$$y_i = \gamma\hat{x}_i + \beta = BN_{\gamma,\beta}(x_i)$$

where $\mu_t$ - estimated batch mean
$\sigma_t^2$ - estimated batch variance
$\hat{x}_i$ - normalized input
$\gamma$ - learnable scale parameter
$\beta$ - learnable shift parameter

## BatchNorm

**On inference:** we can't compute $\mu_t, \sigma_t^2$. Instead, we use some running average over $\mu_t, \sigma_t^2$ that were observed during training.

- NN in theory can learn $\gamma, \beta$ to undo batch normalization. In practice, they usually don't
- BatchNorm stabilizes training by making surface of loss function more smooth


1, 2 ......
100, 200 ......
$x_1$
$x_2$
$w_1$
$w_2$
+
$a$
$b$
1, 2 ......
1, 2 ......
Loss L
$w_1$
$w_2$
Loss L