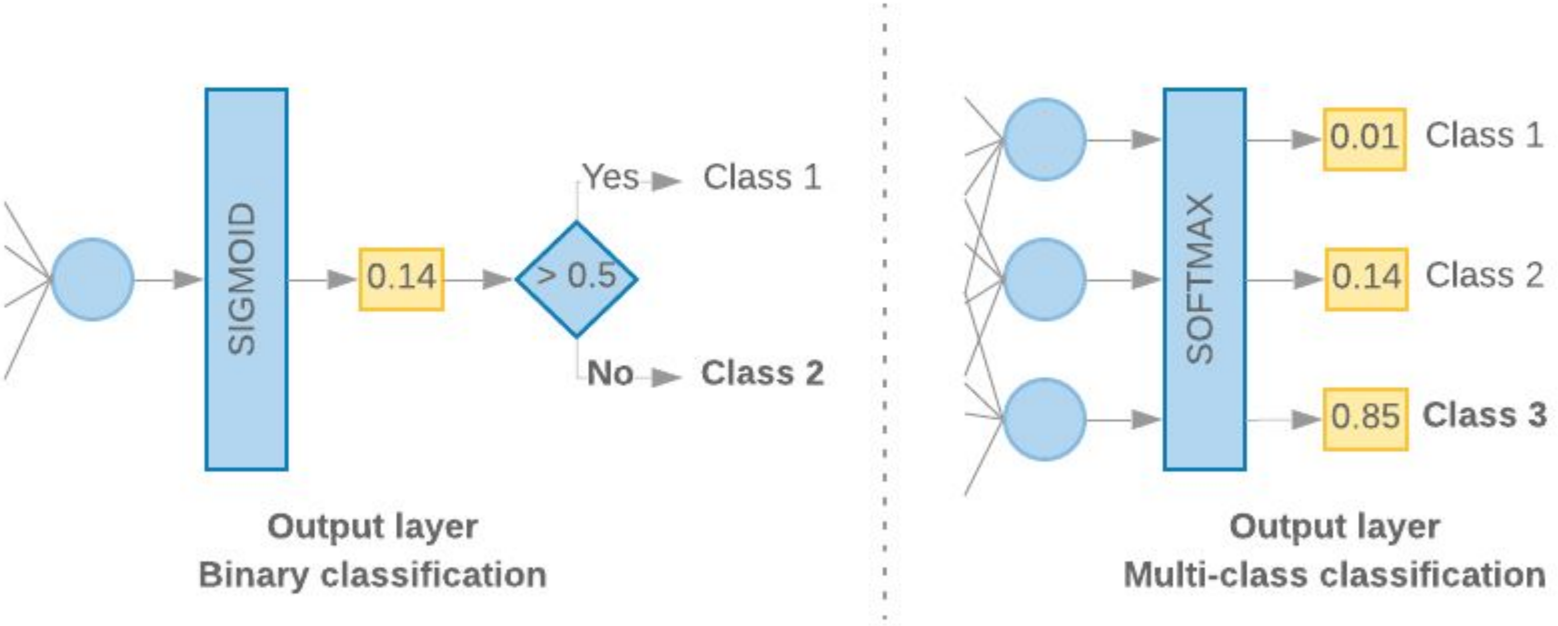
DL components
(complementary slides)

Activations and proper loss functions
Optimizers

Activation: Sigmoid vs Softmax



Typical NN architectures for different problems

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

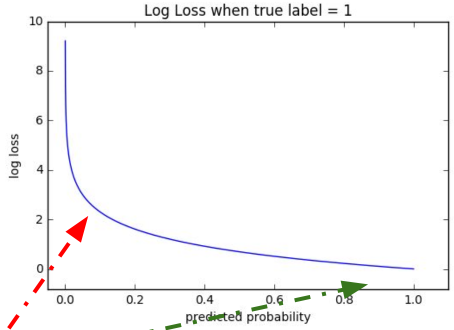
Logistic-Loss (cross-entropy) function for sigmoid\softmax activation layers

$$\log Loss = \sum_{n} [-y \log(y') - (1-y)\log(1-y')]$$

y' = predicted ("probability")
y = label

Intuition: **y** $\neq$ **y'** ⇨ Big loss

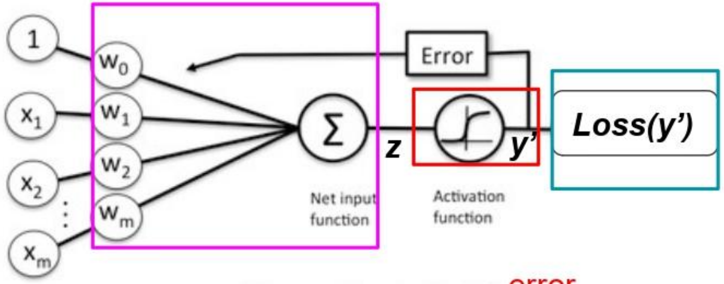
y	y'	logLoss
0	0.9	2.3
0	0.1	0.1
1	0.9	0.1
1	0.1	2.3



Update Rule for log loss minimization

Apply the chain rule

$$\frac{\partial Loss}{\partial W}=\boxed{\frac{\partial Loss}{\partial y'}}\boxed{\frac{\partial y'}{\partial z}}\boxed{\frac{\partial z}{\partial w}}$$



$$\boxed{\frac{\partial Loss}{\partial y'}}=\frac{-y}{y'}+\frac{1-y}{1-y'}$$

$$\boxed{\frac{\partial y'}{\partial z}}=y'(1-y')$$

$$\boxed{\frac{\partial z}{\partial w_i}}=x_i$$

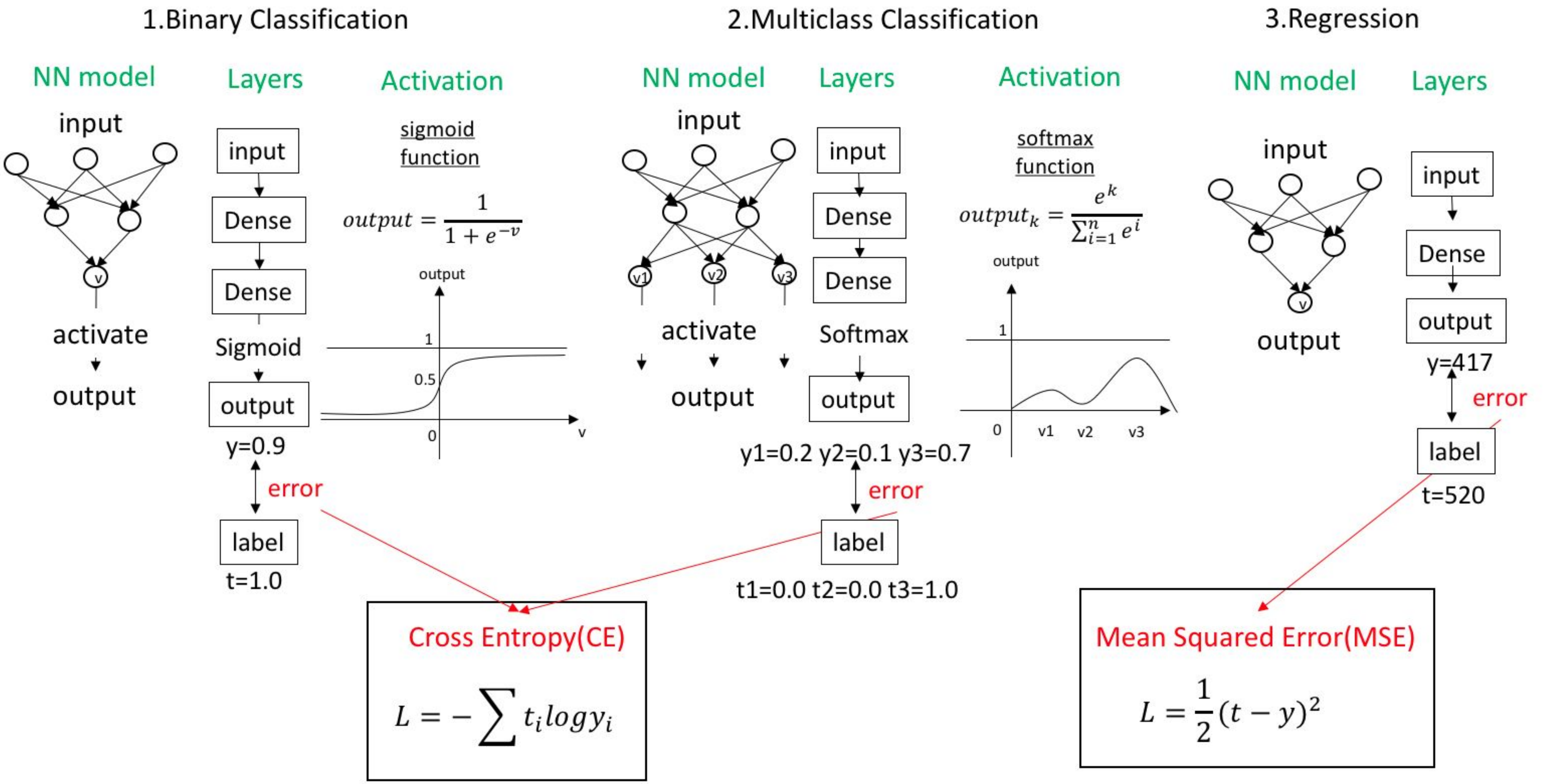
$$\frac{\partial Loss}{\partial W_i}=(y-y')x_i$$

Adaptation step

error

$$w_i \leftarrow w_i-\alpha\frac{\partial Loss}{\partial W_i}=w_i-\alpha(y'-y)x_i$$

Algorithm looks identical to linear regression!



1.Binary Classification
2.Multiclass Classification
3.Regression
NN model
Layers
Activation
input
activate
output
v
input
Dense
Dense
Sigmoid
output
y=0.9
error
label
t=1.0
sigmoid function
$output = \frac{1}{1 + e^{-v}}$
output
1
0.5
0
v
NN model
Layers
Activation
input
v1
v2
v3
activate
output
input
Dense
Dense
Softmax
output
y1=0.2 y2=0.1 y3=0.7
error
label
t1=0.0 t2=0.0 t3=1.0
softmax function
$output_k = \frac{e^k}{\sum_{i=1}^{n} e^i}$
output
1
0
v1
v2
v3
NN model
Layers
input
v
output
input
Dense
output
y=417
error
label
t=520
Cross Entropy(CE)
$L = -\sum t_i log y_i$
Mean Squared Error(MSE)
$L = \frac{1}{2}(t - y)^2$

The parameter which minimize loss function

$$\frac{\partial L}{\partial w_{jk}}$$

	Binary Classification		Multiclass Classification		Regression
Activation	Sigmoid		Softmax		
Loss	MSE	CE	MSE	CE	MSE
Equation No.	1	2	1	2	3

1. $\frac{\partial L}{\partial w_{jk}} = \underbrace{(t_k - y_k)}_{A\ differential\ of\ L} \times \underbrace{y_k \times (1 - y_k)}_{A\ differential\ of\ Activate} \times x_j$

cancel

2. $\frac{\partial L}{\partial w_{jk}} = \underbrace{\frac{t_k - y_k}{y_k \times (1 - y_k)}}_{A\ differential\ of\ L} \times \underbrace{y_k \times (1 - y_k)}_{A\ differential\ of\ Activate} \times x_j = (t_k - y_k) \times x_j$

3. $\frac{\partial L}{\partial w_{jk}} = \underbrace{(t_k - y_k)}_{A\ differential\ of\ L} \times x_j$

Recommend Setting

1. Binary Classification
   Sigmoid + CE
2. Multi Classification
   Softmax + CE
3. Regression
   MSE

Problem
- Sigmoid + MSE
- Softmax + MSE

Learning speed will be decreased by $y_k \times (1 - y_k)$ term

Optimizers

- We have considered approaches to gradient descent which vary the number of data points involved in a step.
- However, they have all used the standard update formula:

$$\mathrm{W} := W - \alpha \cdot \nabla J$$

- There are several variants to updating the weights which give better performance in practice.
- These successive “tweaks” each attempt to improve on the previous idea.
- The resulting (often complicated) methods are referred to as “optimizers”.

Momentum

Gradient Descent Update Rule

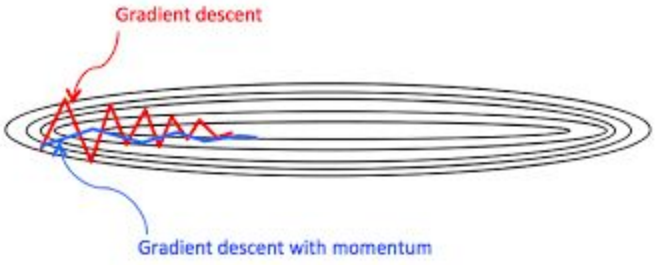
$$w_{t+1} = w_t - \eta \nabla w_t$$

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

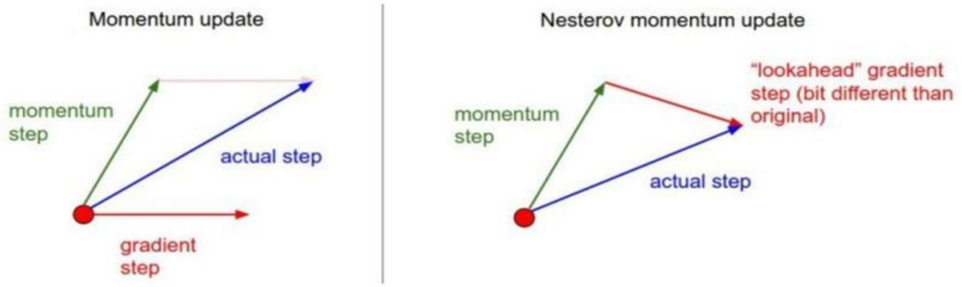
Like a low-pass-filter: smooth the GDS trajectory



Nesterov

Look Ahead as Well as Back (Nesterov)

- Momentum-based gradient descent only uses the current estimated gradient and (a filtered version) of the past estimates. Why not also look ahead to future values?



Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

AdaGrad

- Idea: scale the update for each weight separately.
- Update frequently-updated weights less
- Keep running sum of previous updates
- Divide new updates by factor of previous sum

$$W := W - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla J$$

RMSProp

- Quite similar to AdaGrad.
- Rather than using the sum of previous gradients, decay older gradients more than more recent ones.
- More adaptive to recent updates

adam

Idea: use both first-order and second-order change information and decay both over time.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla J$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla J$$

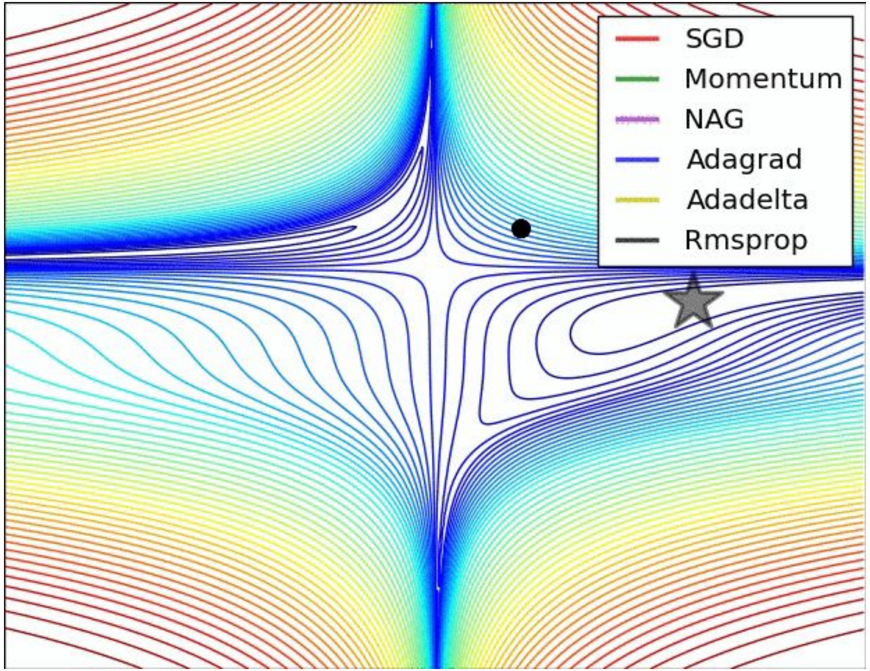
$$m_t = \frac{m_t}{1 - \beta_1^t}$$

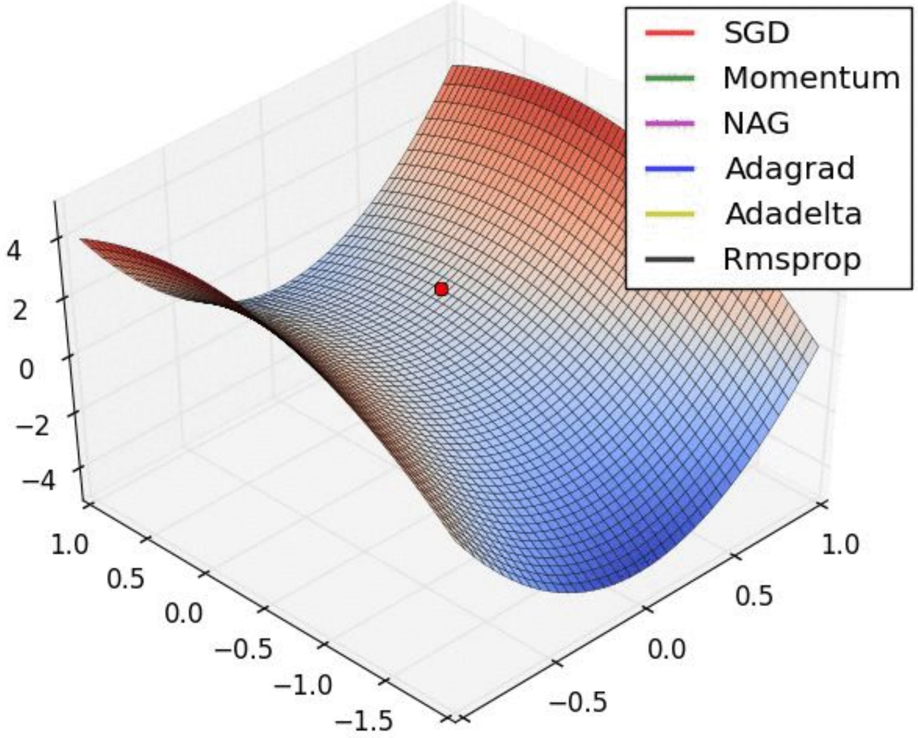
$$v_t = \frac{v_t}{1 - \beta_1^t}$$

$$W := W - \frac{\eta}{\sqrt{v_t} + \epsilon} \odot m_t$$

Which one should I use?!

- RMSProp and Adam seem to be quite popular now.
- Difficult to predict in advance which will be best for a particular problem.
- Still an active area of inquiry.


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5