DIVING INTO THE MACHINE ROOM

Lecture 2

MAL2, Spring 2025

Today's Goals:

Understanding the how's and the <u>why's</u> of building a good neural network.

DIVING INTO THE MACHINE ROOM

- How training a neural network works
- Activation functions
- Faster optimizers
- Learning rate scheduling
- Regularization
- General suggestions

GRADIENT DESCENT

gradient

weights & biases

$$\nabla \mathcal{L}(\theta) = \left(\frac{\partial \mathcal{L}}{\partial \theta_0}, \frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \ldots\ldots, \frac{\partial \mathcal{L}}{\partial \theta_n} \right)$$

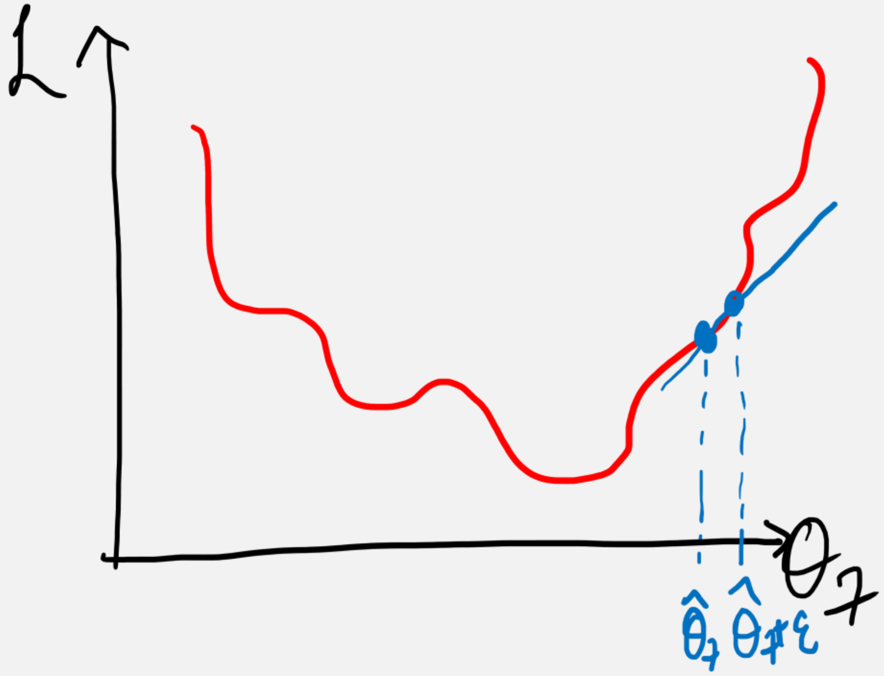
how do we calculate this?

loss function

$$\theta_{new} = \theta_{old} - \eta \nabla \mathcal{L}(\theta)$$

repeat until $\nabla \mathcal{L}(\theta) \approx 0$

GRADIENT DESCENT – IDEA #1



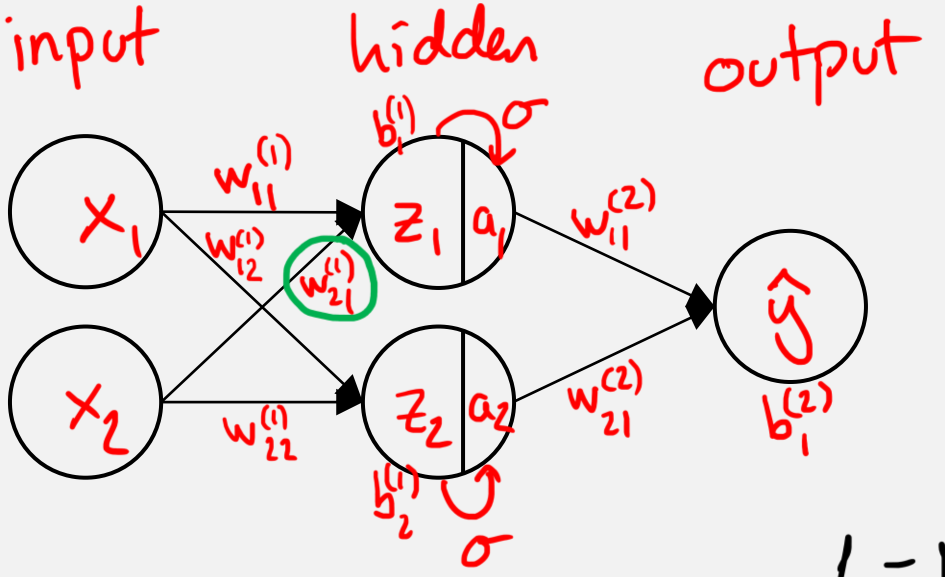
$$\frac{\partial L}{\partial \theta_7} \approx \frac{L(\theta_7 + \varepsilon) - L(\theta_7)}{\varepsilon}$$

INTRACTABLE

This procedure must then be repeated for all parameters at every single training step … and there may be **hundreds of thousands** of parameters!

GRADIENT DESCENT – IDEA #2

Use the chain rule



9 parameters

$\mathcal{L}$ is optimized in 9D space

$\mathcal{L} = MSE \sim (\hat{y} - y)^2$

we want $\frac{\partial \mathcal{L}}{\partial w}$ for all parameters in the model

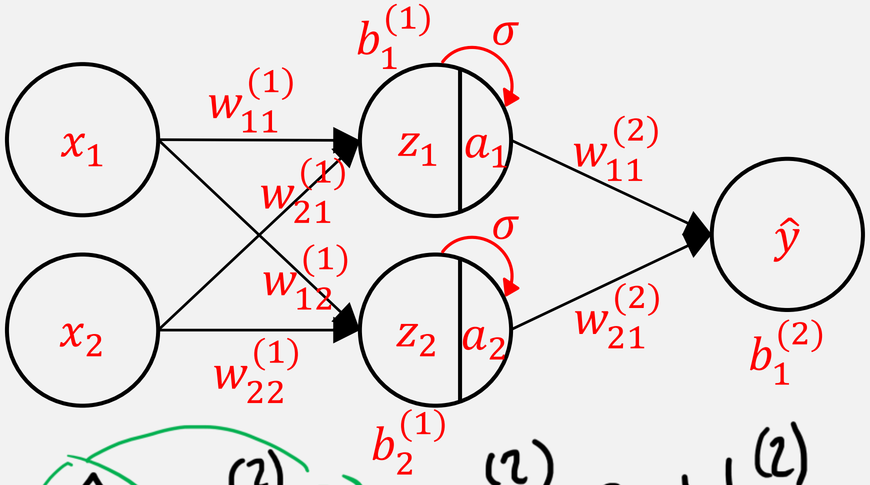
$$\frac{\partial \mathcal{L}}{\partial w_{21}^{(1)}}$$

It gets more complicated with larger networks and multiple "paths" to the same parameter – but the idea is there.

GRADIENT DESCENT – IDEA #2

$$\frac{\partial L}{\partial w_{21}^{(1)}}$$

$$L = (\hat{y} - y)^2$$



$$\frac{\partial L}{\partial w_{21}^{(1)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial w_{21}^{(1)}}$$

$$2(\hat{y}-y) \cdot w_{11}^{(2)} \cdot \sigma' \cdot x_2$$

$$\hat{y} = w_{11}^{(2)} \cdot a_1 + w_{21}^{(2)} \cdot a_2 + b_1^{(2)}$$

$$\sigma(z_1) \qquad \sigma(z_2)$$

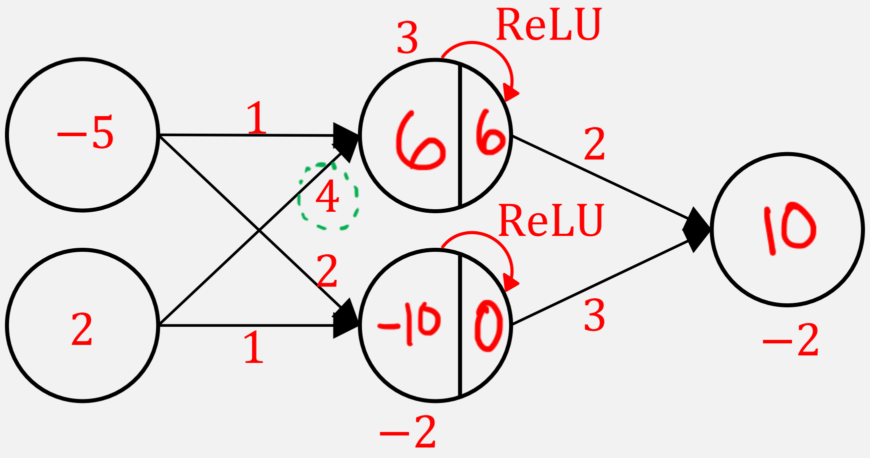
$$z_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + b_1^{(1)}$$

$$z_2 = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}$$

THE BACKPROPAGATION ALGORITHM

$y = 8$

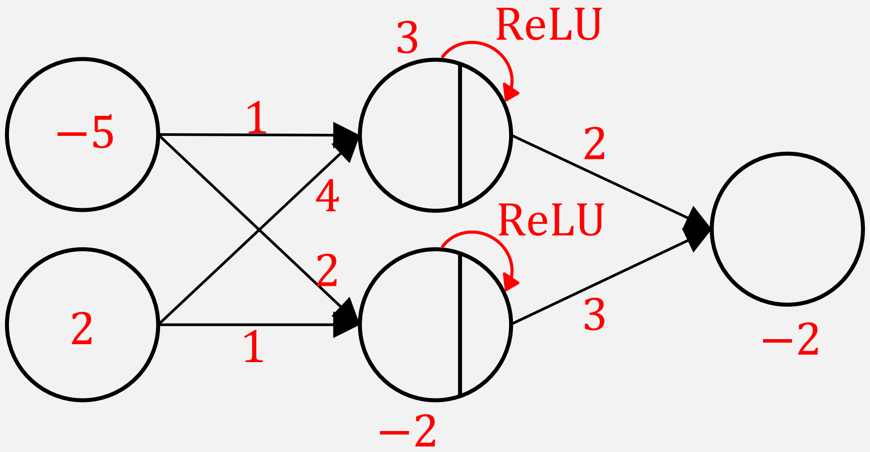
$\hat{y} = 10$



$$\nabla L = \begin{pmatrix} \partial L/\partial w_{11}^{(1)} \\ \partial L/\partial w_{21}^{(1)} \\ \partial L/\partial b_1^{(1)} \\ \partial L/\partial w_{12}^{(1)} \\ \partial L/\partial w_{22}^{(1)} \\ \partial L/\partial b_2^{(1)} \\ \partial L/\partial w_{11}^{(2)} \\ \partial L/\partial w_{21}^{(2)} \\ \partial L/\partial b_1^{(2)} \end{pmatrix} = \begin{pmatrix} -40 \\ 16 \\ 8 \\ 0 \\ 0 \\ 0 \\ 24 \\ 0 \\ 4 \end{pmatrix}$$

$$\frac{\partial L}{\partial w_{21}^{(1)}} = 2(10-8)\cdot 2 \cdot 1 \cdot 2 = 16$$

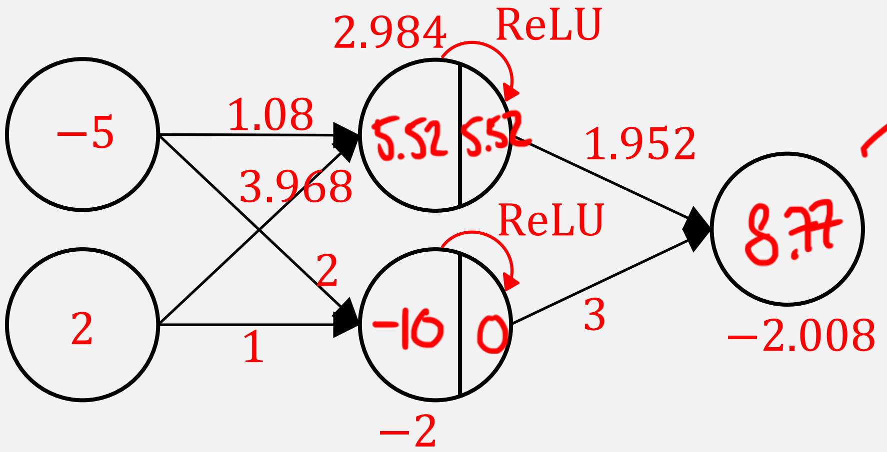
THE BACKPROPAGATION ALGORITHM



$$\nabla L = \begin{pmatrix} \partial L/\partial w_{11}^{(1)} \\ \color{red}{\partial L/\partial w_{21}^{(1)}} \\ \partial L/\partial b_{1}^{(1)} \\ \partial L/\partial w_{12}^{(1)} \\ \partial L/\partial w_{22}^{(1)} \\ \partial L/\partial b_{2}^{(1)} \\ \partial L/\partial w_{11}^{(2)} \\ \partial L/\partial w_{21}^{(2)} \\ \partial L/\partial b_{1}^{(2)} \end{pmatrix} = \begin{pmatrix} -40 \\ \color{red}{16} \\ 8 \\ 0 \\ 0 \\ 0 \\ 24 \\ 0 \\ 4 \end{pmatrix}$$

$$\begin{pmatrix} w_{11}^{(1)} \\ \color{red}{w_{21}^{(1)}} \\ b_{1}^{(1)} \\ w_{12}^{(1)} \\ w_{22}^{(1)} \\ b_{2}^{(1)} \\ w_{11}^{(2)} \\ w_{21}^{(2)} \\ b_{1}^{(2)} \end{pmatrix} = \begin{pmatrix} 1 \\ \color{red}{4} \\ 3 \\ 2 \\ 1 \\ -2 \\ 2 \\ 3 \\ -2 \end{pmatrix} - 0.002 \begin{pmatrix} -40 \\ \color{red}{16} \\ 8 \\ 0 \\ 0 \\ 0 \\ 24 \\ 0 \\ 4 \end{pmatrix} = \begin{pmatrix} 1.08 \\ \color{red}{3.968} \\ 2.984 \\ 2 \\ 1 \\ -2 \\ 1.952 \\ 3 \\ -2.008 \end{pmatrix}$$

THE BACKPROPAGATION ALGORITHM

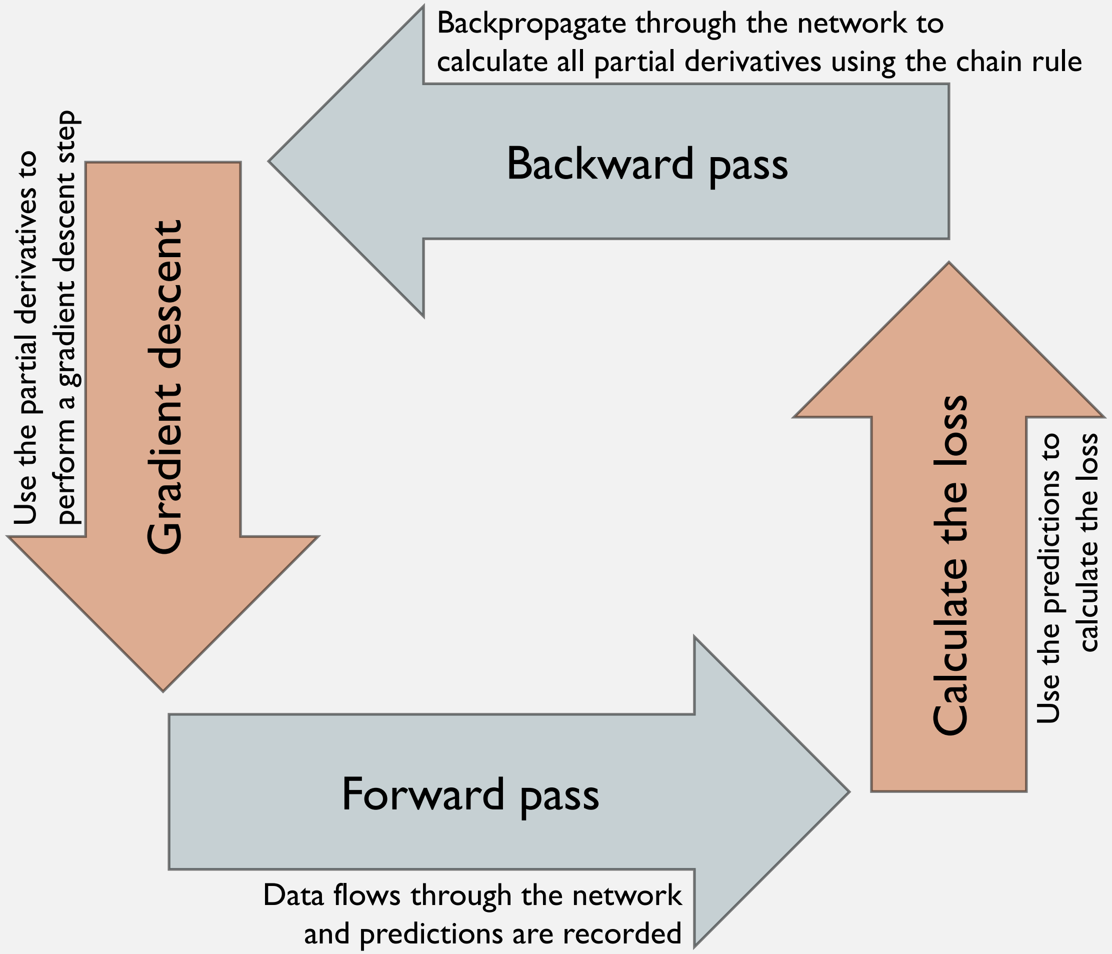


closer to 8

now repeat

$$\begin{pmatrix} w_{11}^{(1)} \\ w_{21}^{(1)} \\ b_1^{(1)} \\ w_{12}^{(1)} \\ w_{22}^{(1)} \\ b_2^{(1)} \\ w_{11}^{(2)} \\ w_{21}^{(2)} \\ b_1^{(2)} \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 2 \\ 1 \\ -2 \\ 2 \\ 3 \\ -2 \end{pmatrix} - 0.002 \begin{pmatrix} -40 \\ 16 \\ 8 \\ 0 \\ 0 \\ 0 \\ 24 \\ 0 \\ 4 \end{pmatrix} = \begin{pmatrix} 1.08 \\ 3.968 \\ 2.984 \\ 2 \\ 1 \\ -2 \\ 1.952 \\ 3 \\ -2.008 \end{pmatrix}$$

THE BACKPROPAGATION ALGORITHM



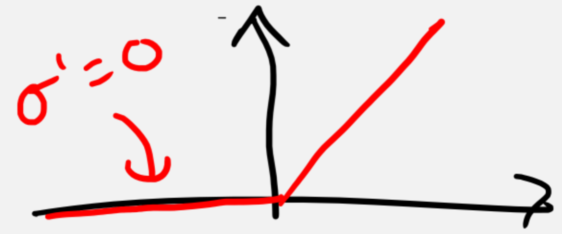
WHY DID WE JUST DO THIS?

$$\boxed{\frac{\partial L}{\partial w_{12}^{(1)}}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_2} \frac{\partial a_2}{\partial z_2} \frac{\partial z_2}{\partial w_{12}^{(1)}} = 2(\hat{y} - y) w_{21}^{(2)} \sigma' x_1 = \boxed{0}$$

derivative of ReLU

$$\nabla L = \begin{pmatrix} \partial L/\partial w_{11}^{(1)} \\ \partial L/\partial w_{21}^{(1)} \\ \partial L/\partial b_1^{(1)} \\ \boxed{\partial L/\partial w_{12}^{(1)}} \\ \partial L/\partial w_{22}^{(1)} \\ \partial L/\partial b_2^{(1)} \\ \partial L/\partial w_{11}^{(2)} \\ \partial L/\partial w_{21}^{(2)} \\ \partial L/\partial b_1^{(2)} \end{pmatrix} = \begin{pmatrix} -40 \\ 16 \\ 8 \\ \boxed{0} \\ 0 \\ 0 \\ 24 \\ 0 \\ 4 \end{pmatrix}$$

- why are these numbers 0?

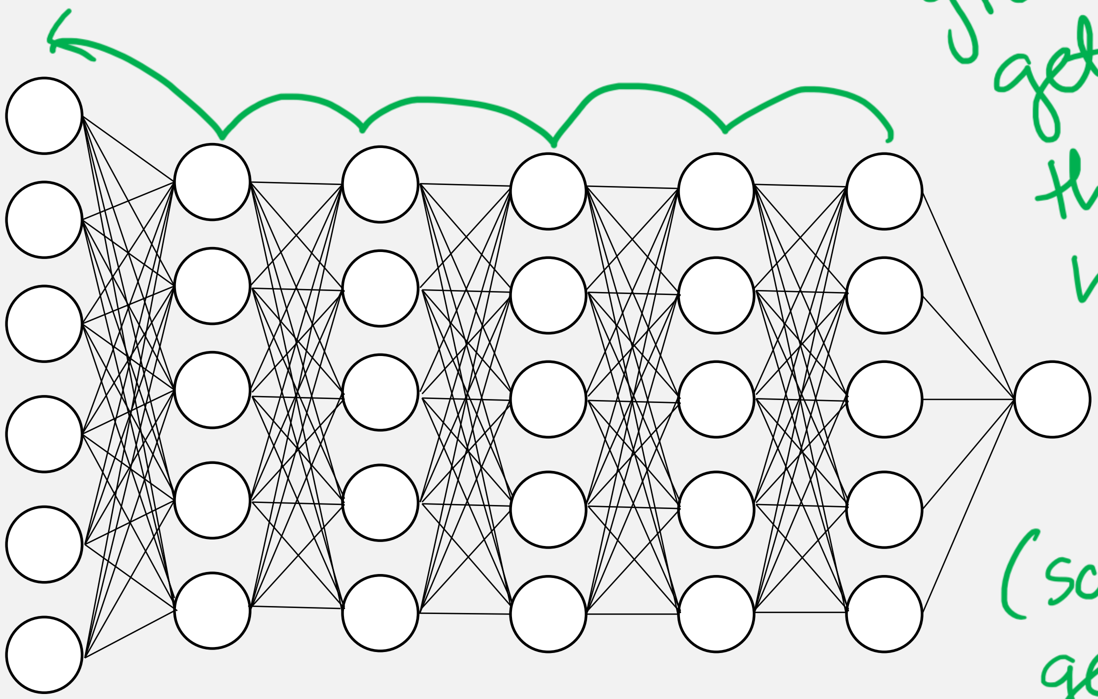


"dying ReLU"

one of many problems we face → keeps outputting zero, killing the neuron

In general: Gradients are unstable in deep neural networks!

VANISHING & EXPLODING GRADIENTS



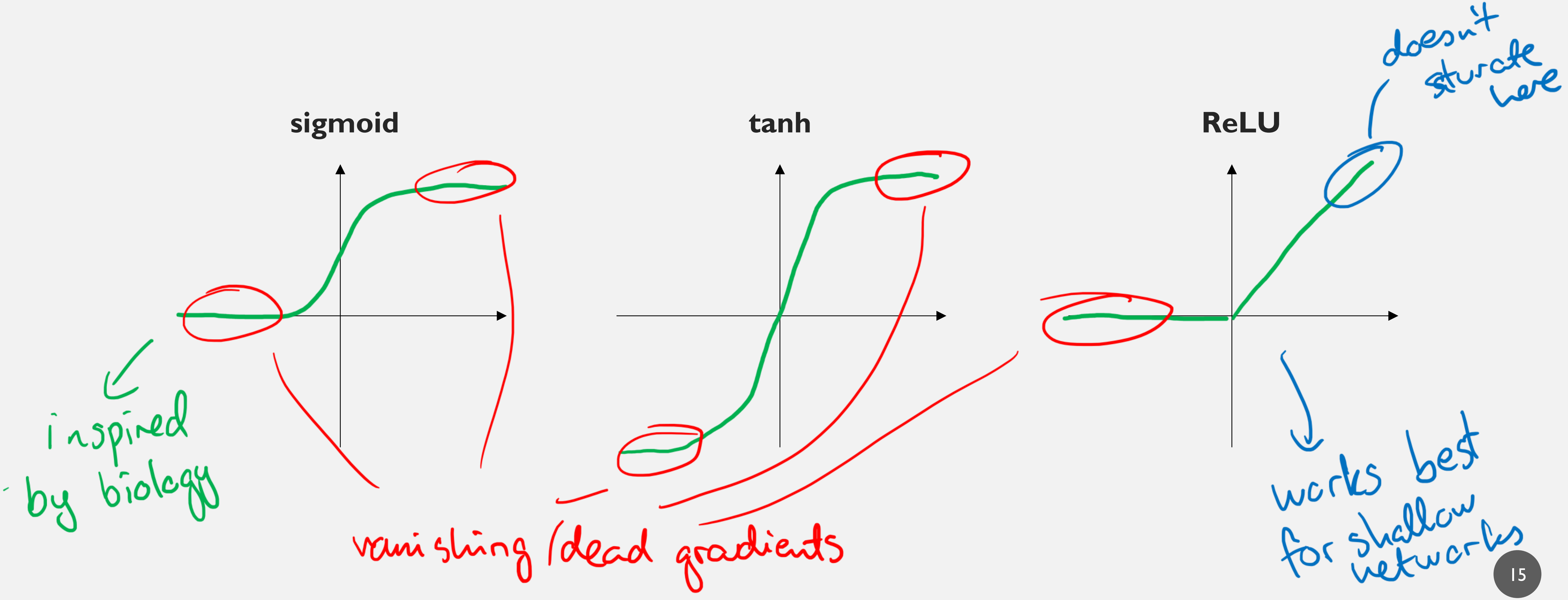
gradients tend to get smaller & smaller, the lower layers hardly get trained

(sometimes they get bigger & bigger, then training diverges)

DIVING INTO THE MACHINE ROOM

- How training a neural network works
- Activation functions
- Faster optimizers
- Learning rate scheduling
- Regularization
- General suggestions

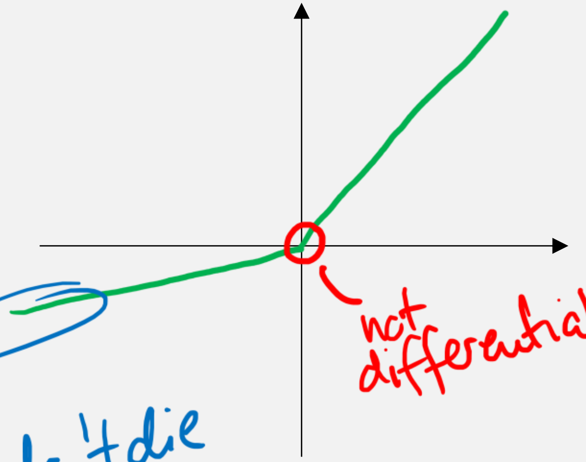
ACTIVATION FUNCTIONS



BETTER ACTIVATION FUNCTIONS

exponential linear unit

Leaky ReLU



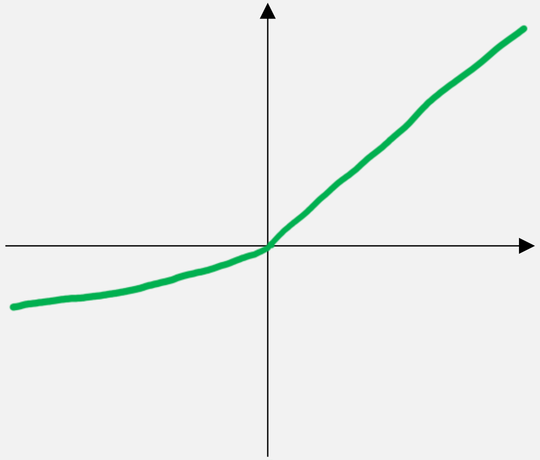
not differentiable

neurons don't die (they go in a coma)

$\max(\alpha z, z)$

α hyperparameter $0<\alpha<1$

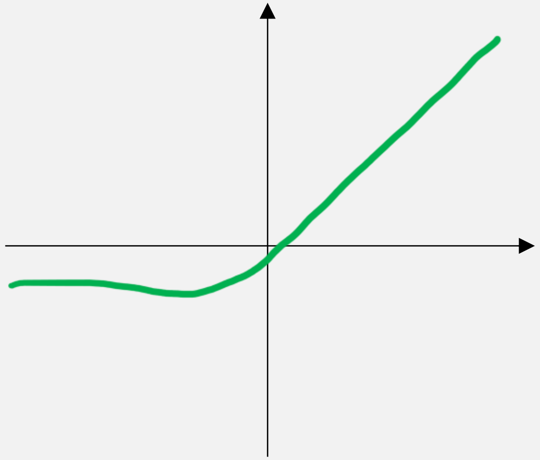
ELU



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

outperforms all variants of ReLU

Swish



$z \cdot \text{sigmoid}(z)$

outperforms everything for deep NNs

RECOMMENDATIONS

Try ReLU for shallow networks and Swish for deep networks

```
layer = Dense(100, activation="relu", kernel_initializer="he_normal")
layer = Dense(100, activation="swish", kernel_initializer="he_normal")
```

how the values of weights and biases are initialized

needs to match activation functions

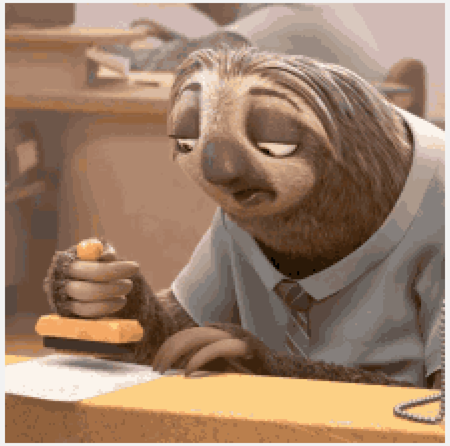
for reference

Initialization method	Activation function
Glorot (default)	tanh, sigmoid, softmax
He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish
LeCun	SELU

DIVING INTO THE MACHINE ROOM

- How training a neural network works
- Activation functions
- Faster optimizers
- Learning rate scheduling
- Regularization
- General suggestions

FASTER OPTIMIZERS



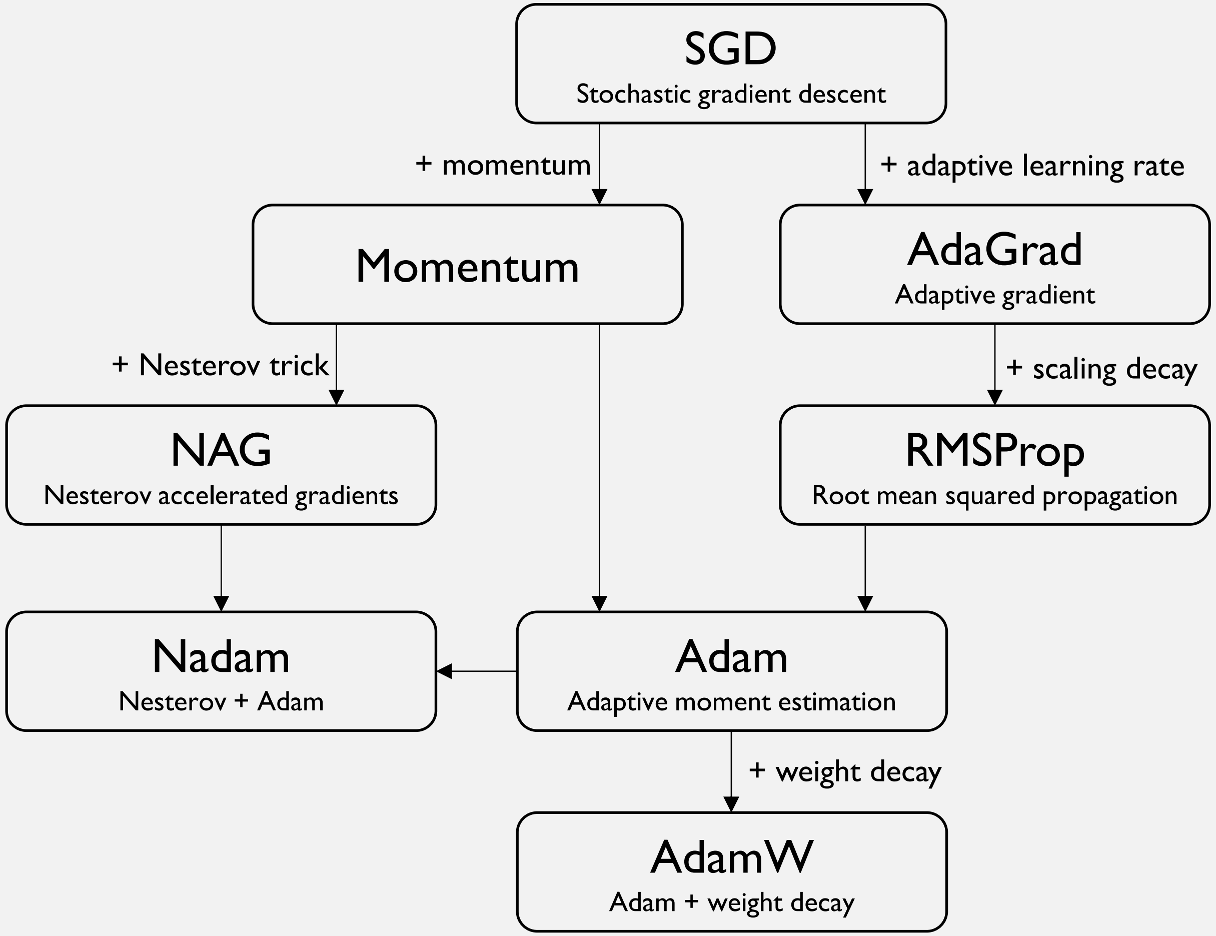
#GradientDescent
UpdatingParameters

#GradientDescent
FinishingAnEpoch

#GradientDescent
WhenItConverges

The point is: Gradient descent can be painfully slow

FASTER OPTIMIZERS

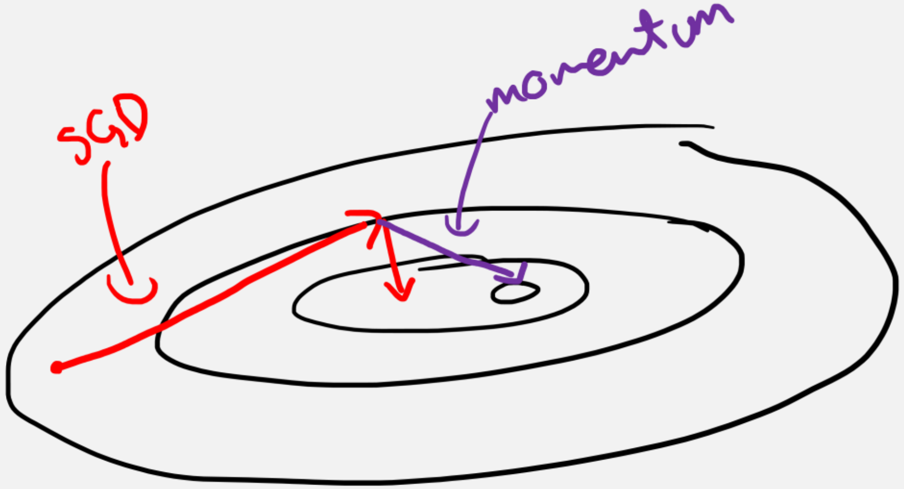


MOMENTUM

”the step we just took probably wasn’t a terrible idea”

Gradient descent

$$\theta \leftarrow \theta - \eta \nabla L(\theta)$$



Momentum

momentum

$$m \leftarrow \beta m - \eta \nabla L(\theta)$$

$$\theta \leftarrow \theta + m$$

```
optimizer = SGD(learning_rate=0.001, momentum=0.9)
```

The trick works because $\boldsymbol{m}$ generally points in the right direction, so the gradient here is slightly more accurate

THE NESTEROV TRICK

Momentum

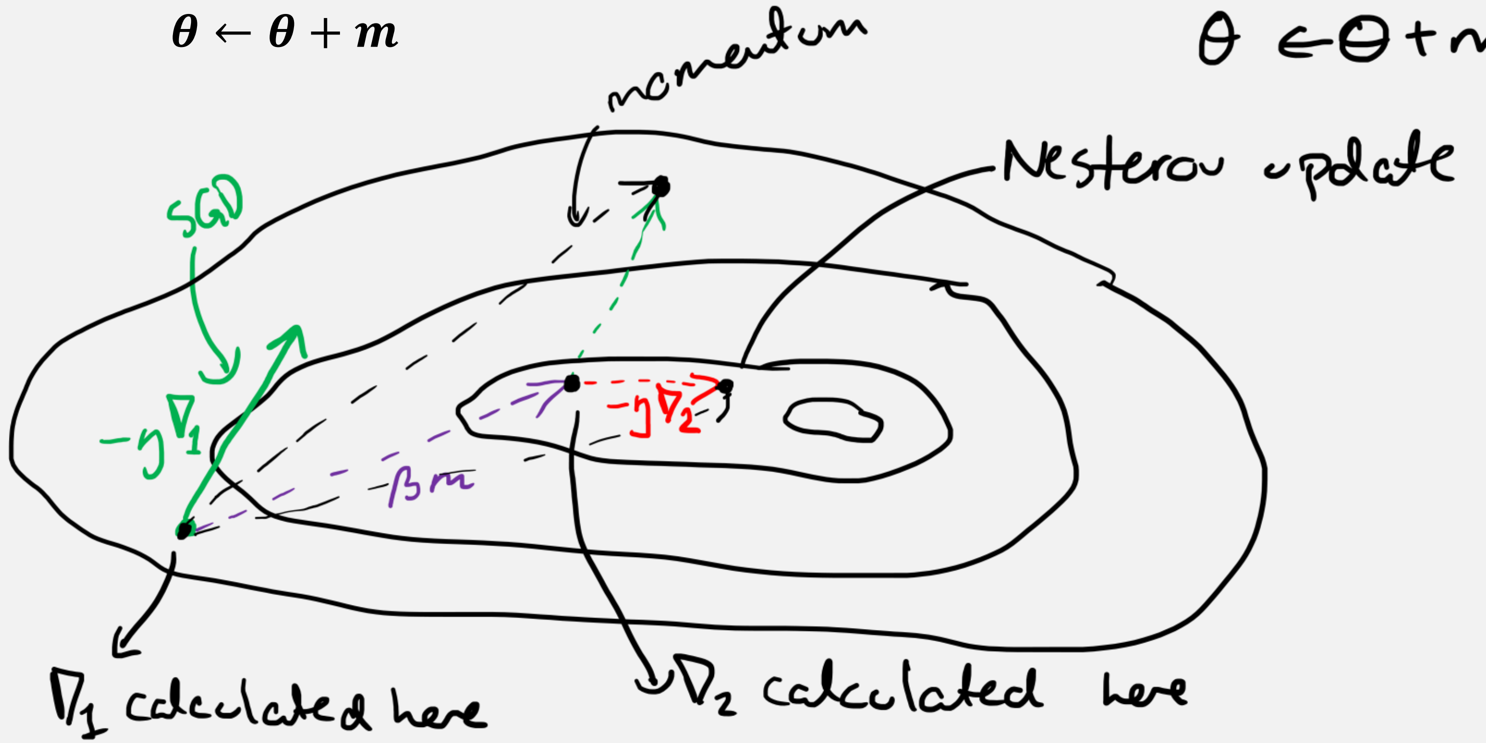
$$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla \boldsymbol{L}(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$$

Nesterov

$$m \leftarrow \beta m - \eta \nabla L(\theta + \beta m)$$

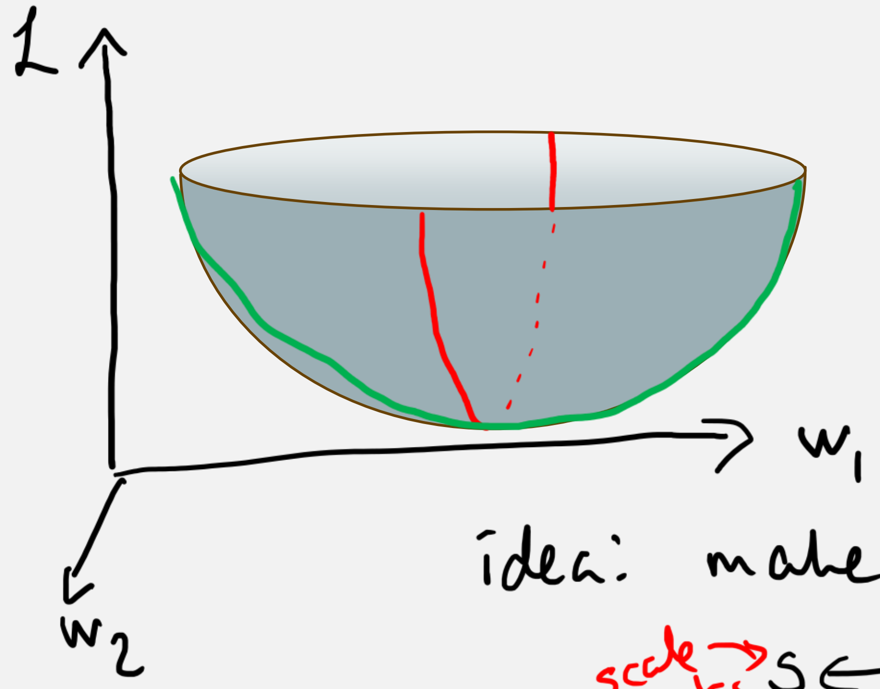
$$\theta \leftarrow \theta + m$$




```
optimizer = SGD(learning_rate=0.001, momentum=0.9, nesterov=True)
```

ADAPTIVE LEARNING RATES

Observation: Loss functions often resemble elongated bowls



with the same η

idea: make η smaller in the steepest directions

scale factors → $s \leftarrow s + \nabla L(\theta) \otimes \nabla L(\theta)$

$\otimes$ = elementwise multiplications

AdaGrad → $\theta \leftarrow \theta - \eta \cdot \nabla L(\theta) \oslash \sqrt{s + \varepsilon}$ → avoid $\bar{0}$ (10^{-10})

↳ η scaled down so much that training stops

SCALING DECAY

add decay to s so it doesn't explode

AdaGrad

$$s \leftarrow s + \nabla L(\theta) \otimes \nabla L(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla L(\theta) \oslash \sqrt{s + \varepsilon}$$

RMSProp

$$s \leftarrow \underbrace{\rho}_{\sim 0.9} s + (1-\rho) \nabla L(\theta) \otimes \nabla L(\theta)$$

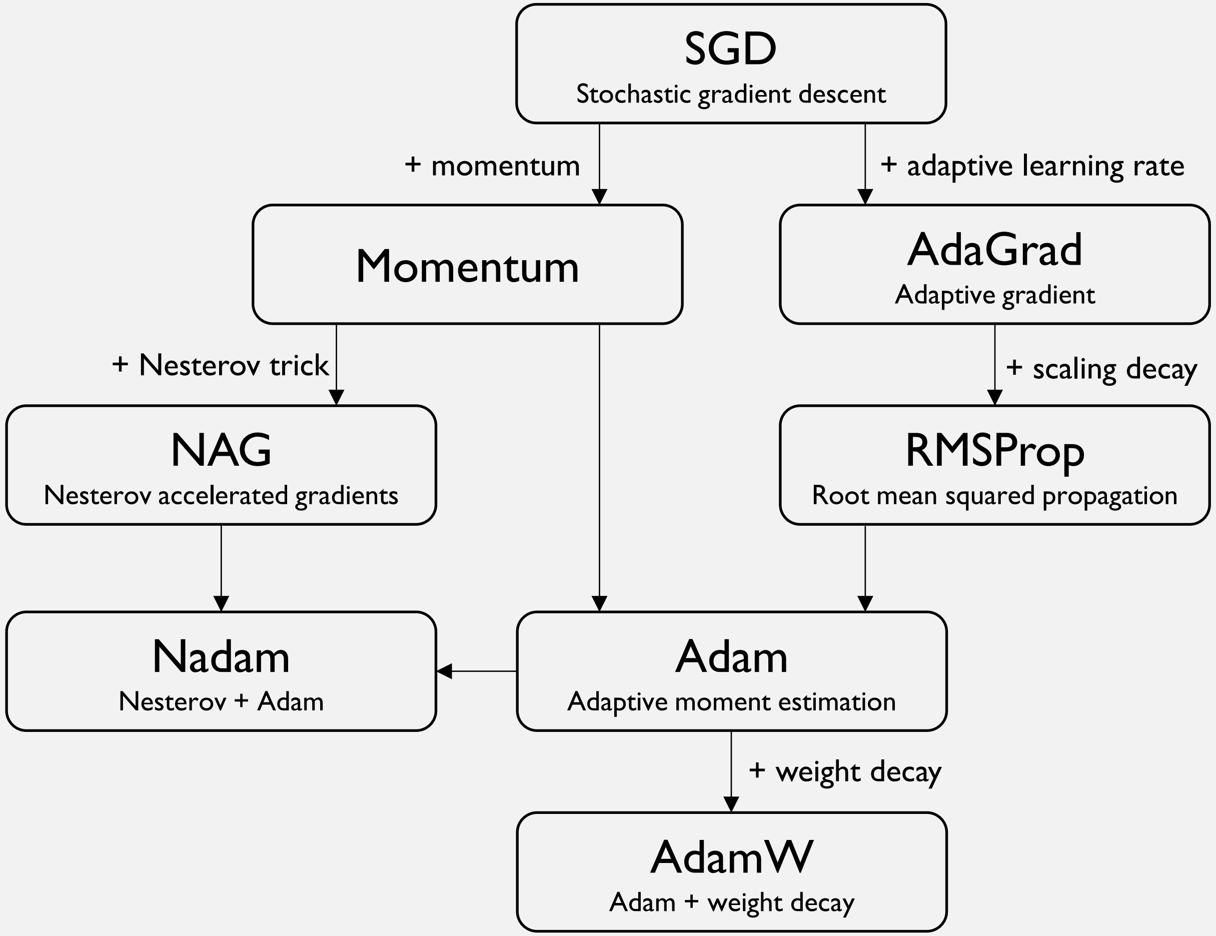
same thing

WEIGHT DECAY

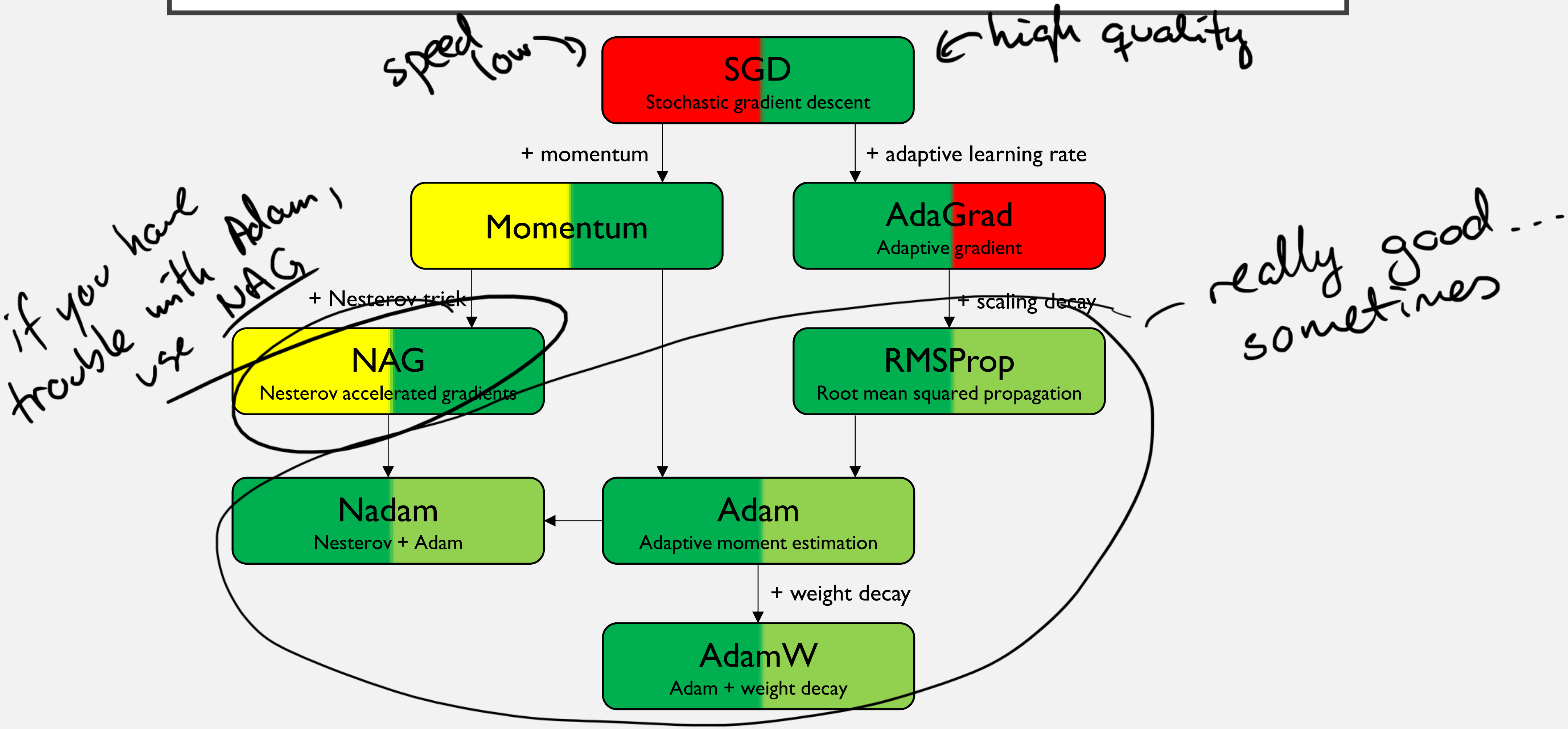
At each training step, multiply all weights by, say 0.95

→ built-in regularization

RECOMMENDATIONS



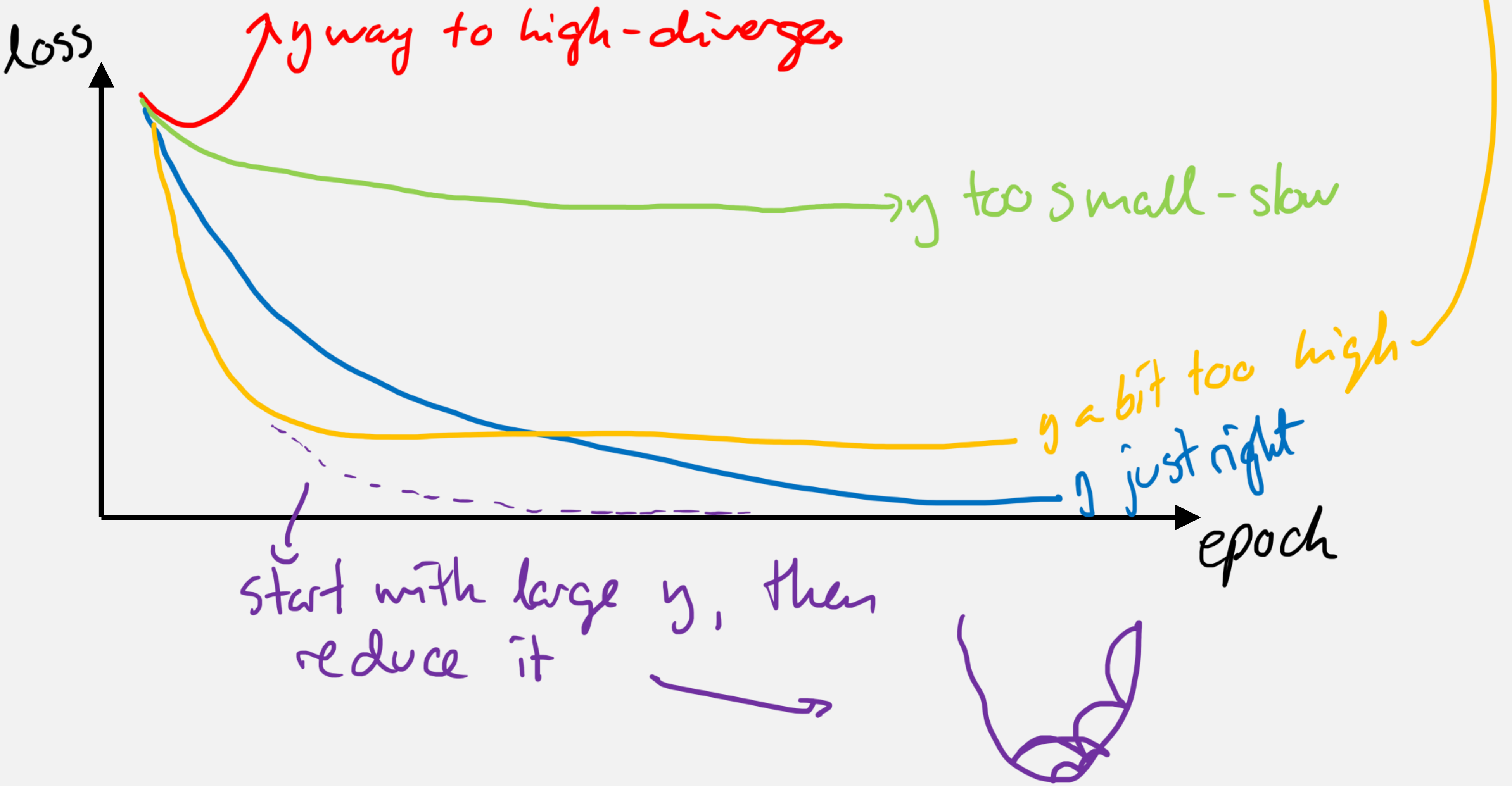
RECOMMENDATIONS



DIVING INTO THE
MACHINE ROOM

- How training a neural network works
- Activation functions
- Faster optimizers
- Learning rate scheduling
- Regularization
- General suggestions

LEARNING RATE SCHEDULING



LEARNING RATE SCHEDULING

Power

$$\eta(t) = \frac{\eta_0}{1 + t/s}$$

iteration #

after s steps: $\eta_0/2$

2s $\eta_0/3$

3s $\eta_0/4$

Exponential

$$\eta(t) = \eta_0 \, 0.1^{t/s}$$

reduces by a factor of 10 every s steps

Piecewise constant

$\eta = 0.1$ for 5 epochs

$\eta = 0.01$ for 10

0.001 for 50

. . . . . .

Performance

measure valid. error every N steps, divide η by λ when error stops dropping

All of them work pretty well – performance scheduling is a good default

CODING A LEARNING RATE SCHEDULE

```
#power scheduling
optimizer = SGD(learning_rate=0.01, decay=1e-4)

#exponential scheduling
def exp_decay_fn(epoch):
    return 0.01 * 0.1 ** (epoch/20)

lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exp_decay_fn)
history = model.fit([...], callbacks = [lr_scheduler])

#piecewise constant scheduling
def piece_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        [...]
        and so on - use LearningRateScheduler callback with piece_fn

#performance scheduling
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(factor = 0.5, patience = 5)
history = model.fit([...], callbacks = [lr_scheduler])
```

DIVING INTO THE
MACHINE ROOM

- How training a neural network works
- Activation functions
- Faster optimizers
- Learning rate scheduling
- Regularization
- General suggestions

L1 AND L2 REGULARIZATION

tools to avoid overfitting

Remember Lasso and Ridge regression?

penalty for large coefficients

$\hookrightarrow \quad \alpha \sum_i \theta_i^2 \quad (L2)$

$\alpha \sum_i |\theta_i| \quad (L1)$

never use with Adam-types - use AdamW instead

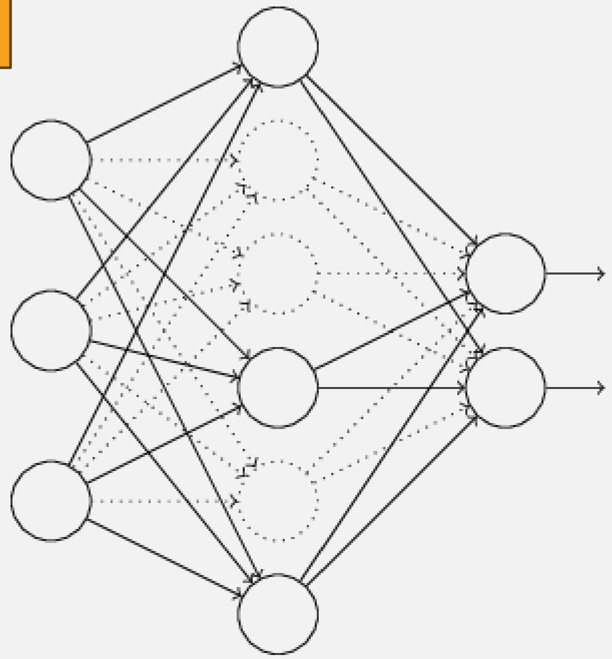
α

```
layer = Dense([...], kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

DROPOUT REGULARIZATION

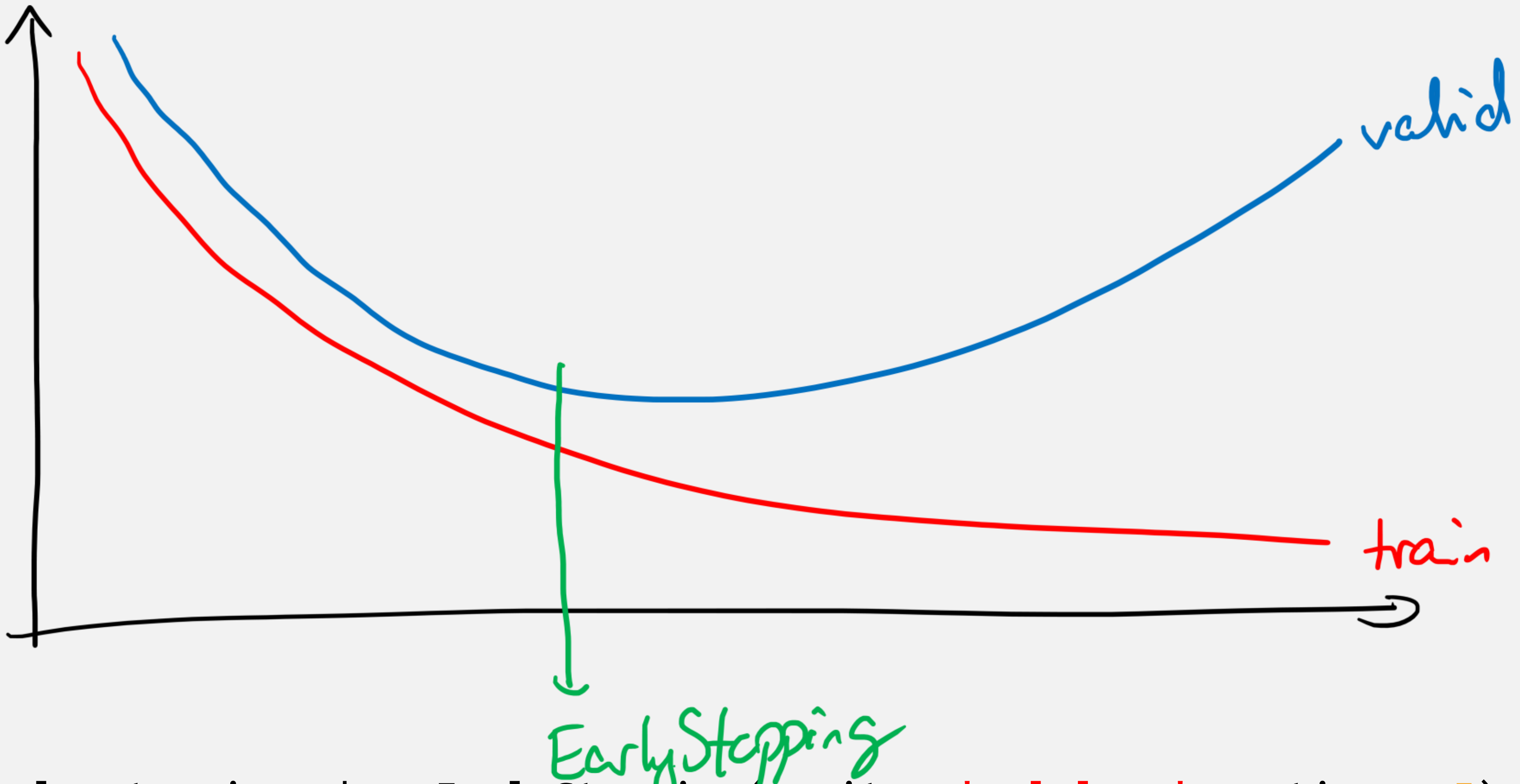
At every training step, every neuron has a probability p of being entirely ignored!

So the model can't rely too much on any particular neuron, making it more robust



```
tf.keras.layers.Dropout(0.2)
```

BUT THE BEST WAY TO PREVENT OVERFITTING IS USUALLY …




```
early_stopping_cb = EarlyStopping(monitor='val_loss', patience=5)
history = model.fit([...], callbacks = [early_stopping_cb])
```

DIVING INTO THE MACHINE ROOM

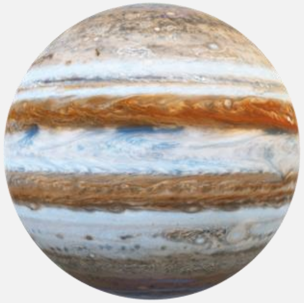
- How training a neural network works
- Activation functions
- Faster optimizers
- Learning rate scheduling
- Regularization
- General suggestions

GENERAL SUGGESTIONS

Hyperparameter	Recommendations
Activation function	ReLU if shallow. Swish if deep (Leaky ReLU as a faster alternative).
Optimizer	Start with Adam or AdamW, but switch to NAG if it doesn't work out. Never use SGD or AdaGrad.
Learning rate schedule	Performance scheduling is pretty good and requires very little hyperparameter tuning, but others may do better if you do it right.
Regularization	EarlyStopping and weight decay will usually do the trick, but look at others if you can't get rid of overfitting. Never use L1/L2 with Adam-type optimizers.

TWO TASKS

Find your neural network from last week



and make it better now that you are cleverer

Scan this QR code

and tell me about something you are still unsure about

You have 20 minutes