DIVING INTO THE MACHINE ROOM

Lecture 2

MAL2, Fall 2025

Today's Goals:

Understand the how's and the why's 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

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

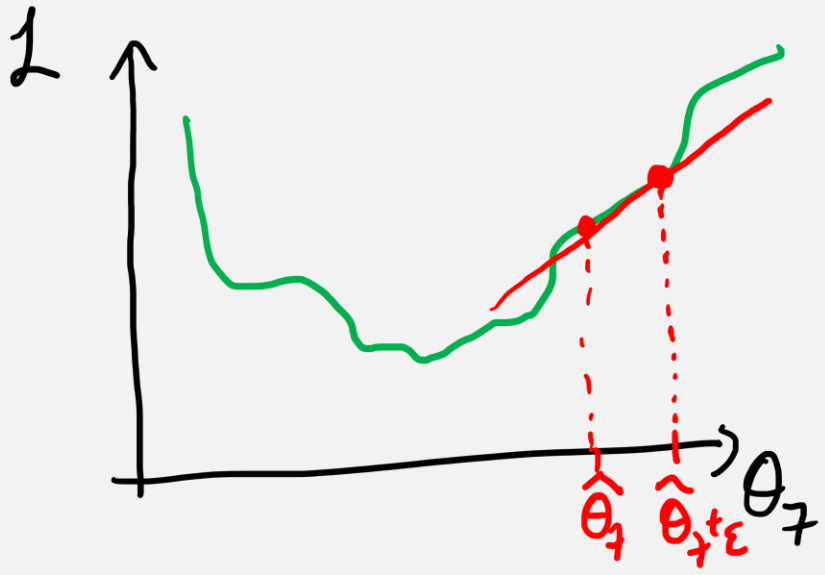
← how to calculate this

loss function

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

repeat until $\nabla \mathcal{L} \sim 0$

GRADIENT DESCENT – IDEA #1

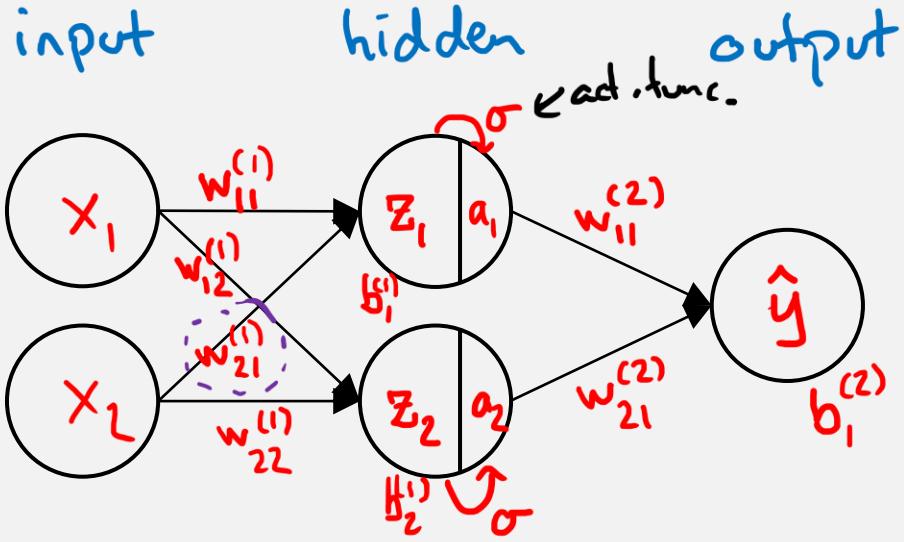


$$\frac{\partial L}{\partial \theta_7} \simeq \frac{L(\theta_7+\varepsilon) - L(\theta_7)}{\varepsilon}$$

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

INTRACTABLE

GRADIENT DESCENT – IDEA #2



Use the chain rule

9 parameters
1 optimized in 9D space

We want $\frac{\partial \mathcal{L}}{\partial \theta}$

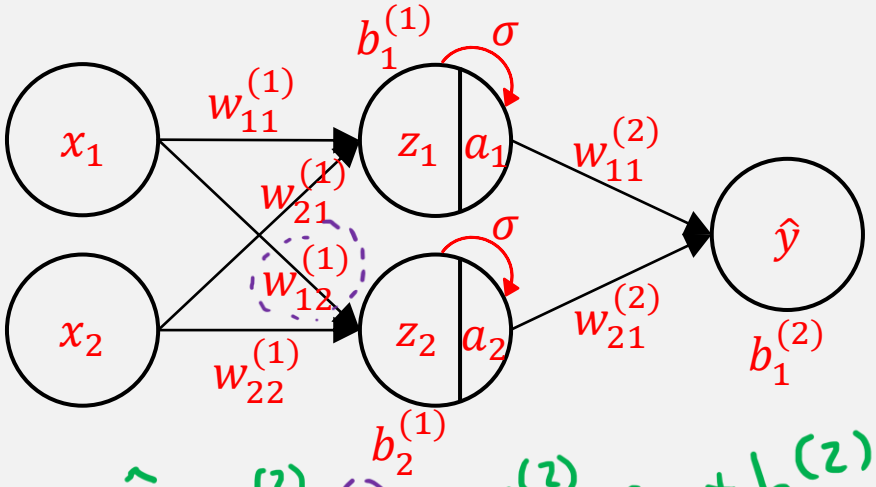
for all parameters

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

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

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

GRADIENT DESCENT – IDEA #2



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

chain rule

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

$2(\hat{y}-y)$ $w_{11}^{(2)}$ σ' x_2

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

$$a_1 = \sigma(z_1) \qquad a_2 = \sigma(z_2)$$

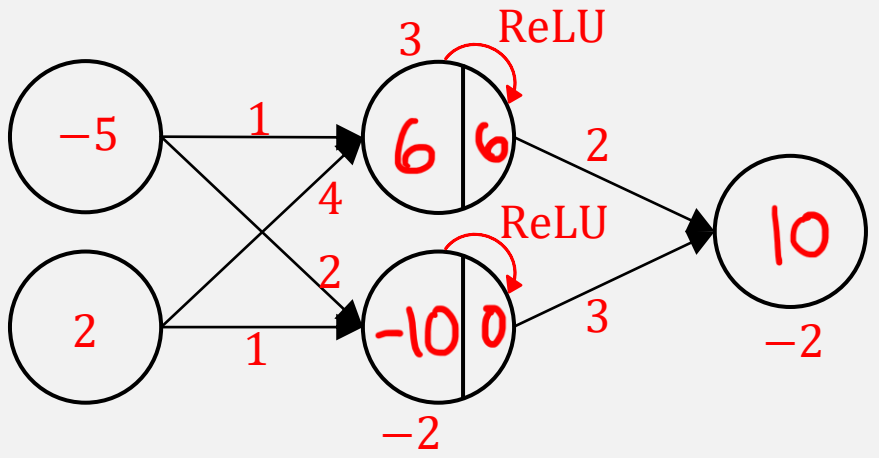
$$z_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + b_1^{(1)}$$

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

THE BACKPROPAGATION ALGORITHM

$y = 8$

$\hat{y} = 10$

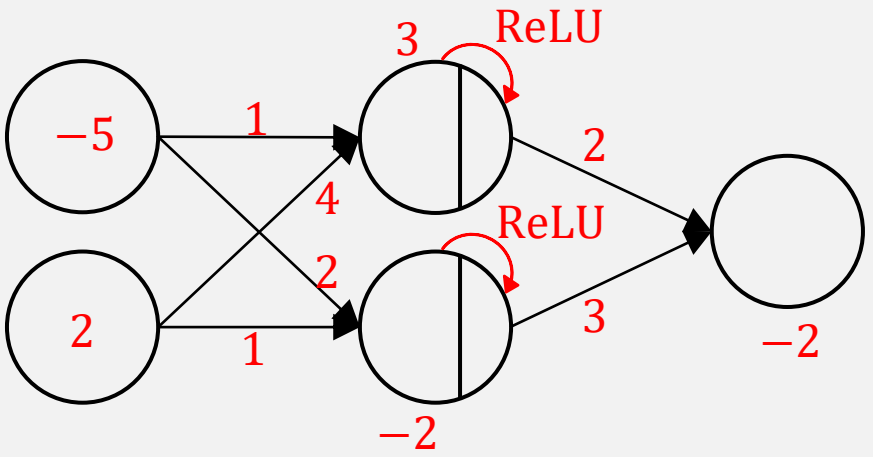


$$\frac{\partial L}{w_{21}^{(1)}} = 2(\hat{y}-y)w_{11}^{(2)}\sigma' x_2$$

$$= 2(10-8)\cdot 2\cdot 1\cdot 2 = 16$$

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

THE BACKPROPAGATION ALGORITHM

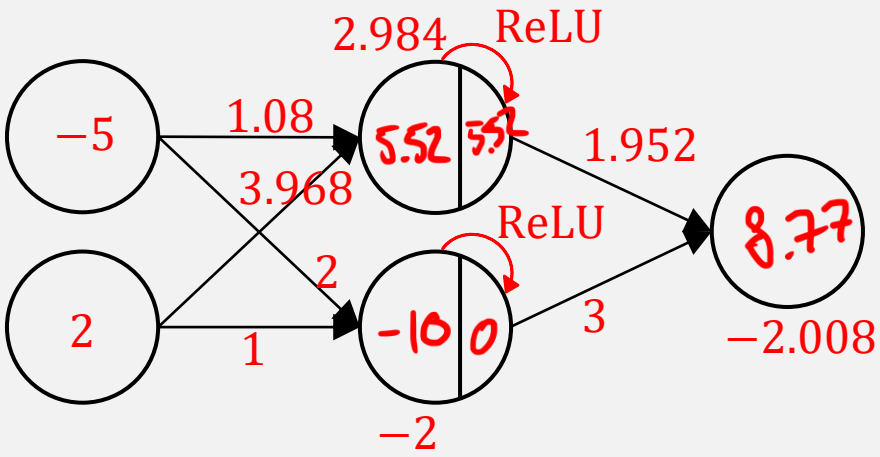


$$\begin{pmatrix} w_{11}^{(1)} \\ w_{21}^{(1)} \\ b_1^{(1)} \\ w_{12}^{(1)} \\ w_{22}^{(1)} \\ b_2^{(1)} \\ w_{11}^{(2)} \\ w_{21}^{(2)} \\ b_1^{(2)} \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 2 \\ 1 \\ -2 \\ 2 \\ 3 \\ -2 \end{pmatrix} - 0.002 \begin{pmatrix} -40 \\ 16 \\ 8 \\ 0 \\ 0 \\ 0 \\ 24 \\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

learning rate

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

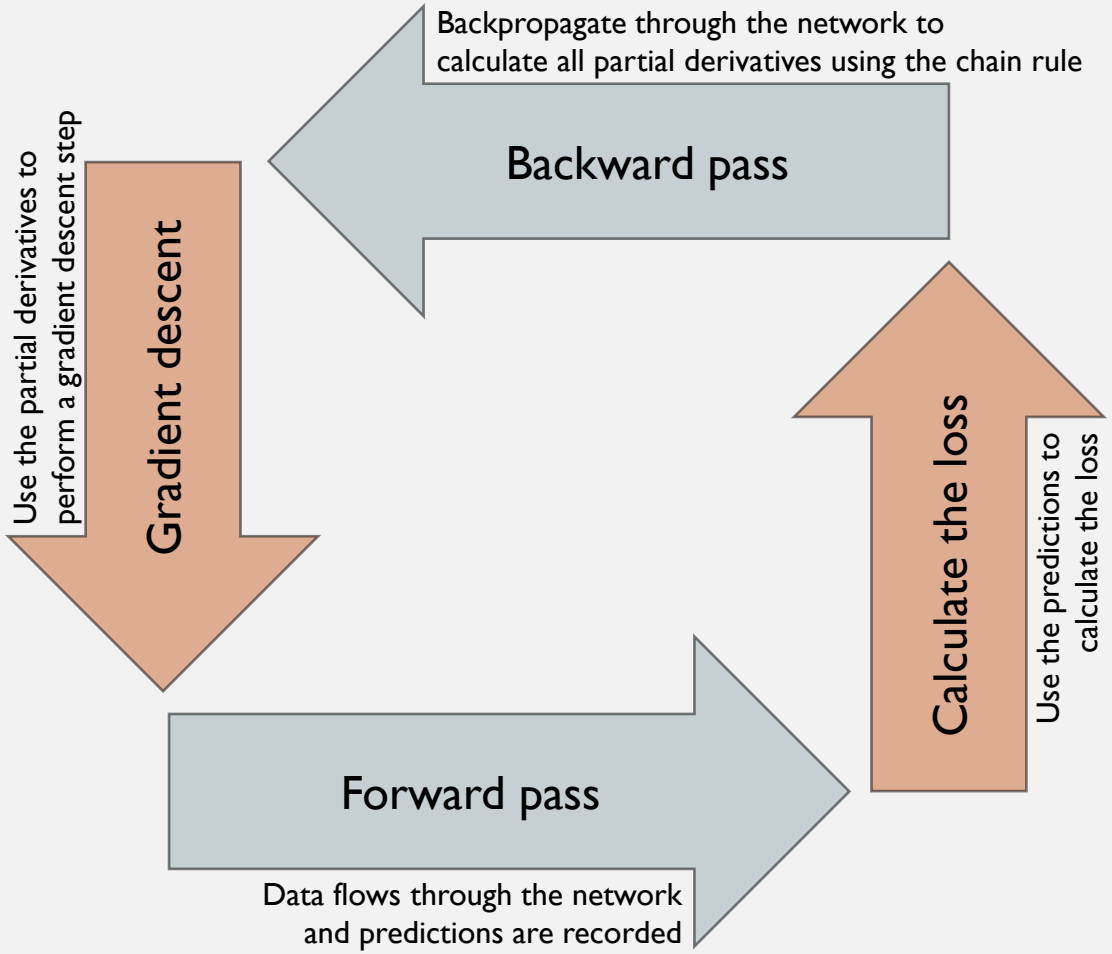
THE BACKPROPAGATION ALGORITHM



→ closer to actual value y=8 now reject

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

THE BACKPROPAGATION ALGORITHM



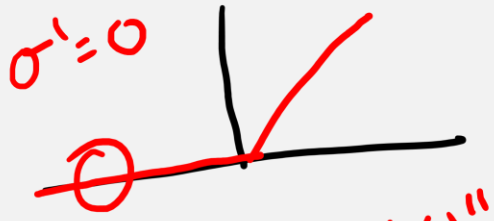
WHY DID WE JUST DO THIS?

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

derivative of ReLU

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

why are these numbers zero?

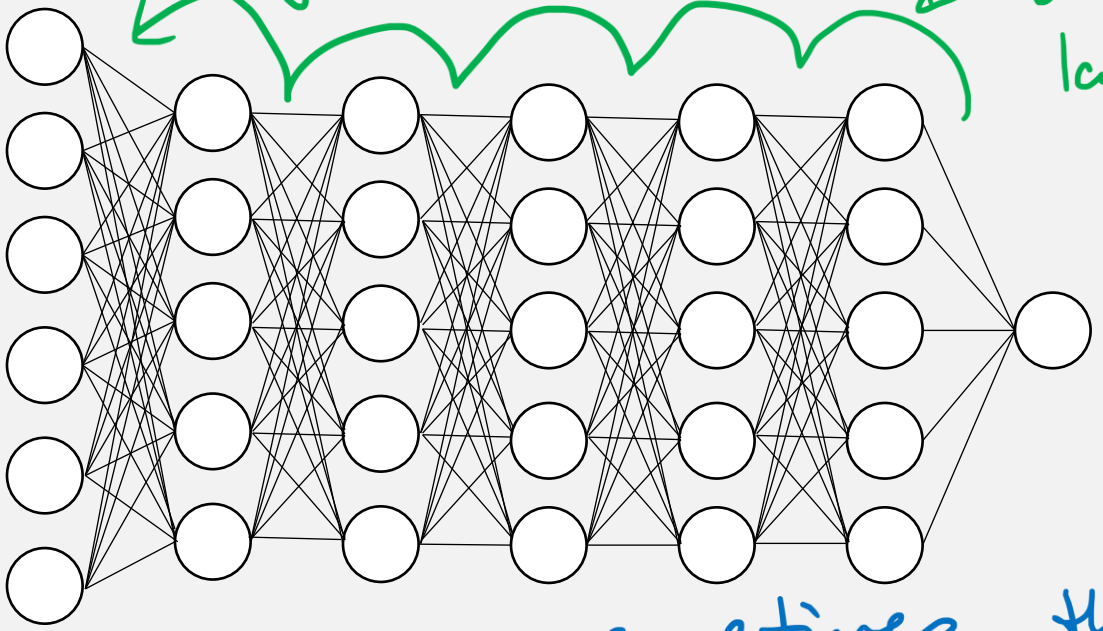


"dying ReLU"

one of many problems faced during training → keeps outputting 0, "kills" the neuron

In general: Gradients are unstable in deep neural networks!

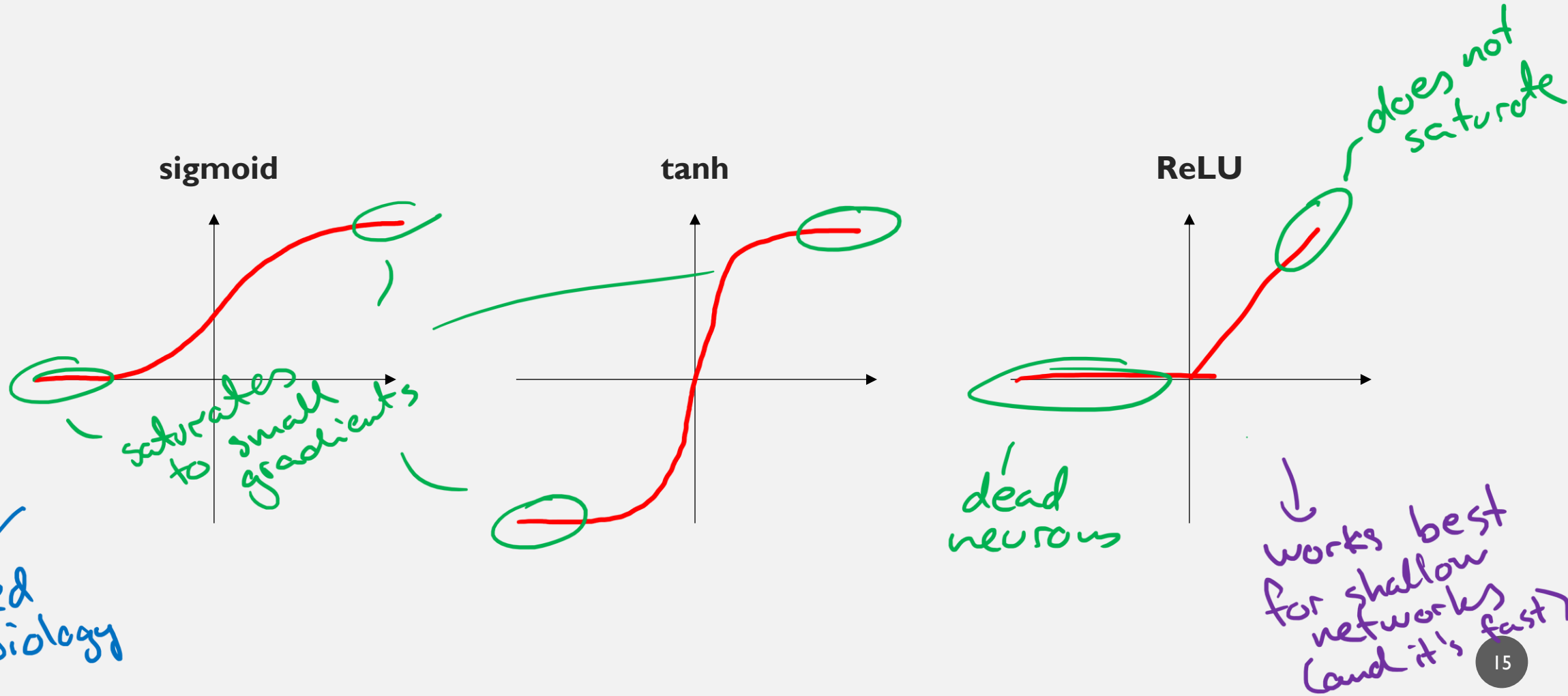
VANISHING & EXPLODING GRADIENTS



DIVING INTO THE
MACHINE ROOM

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

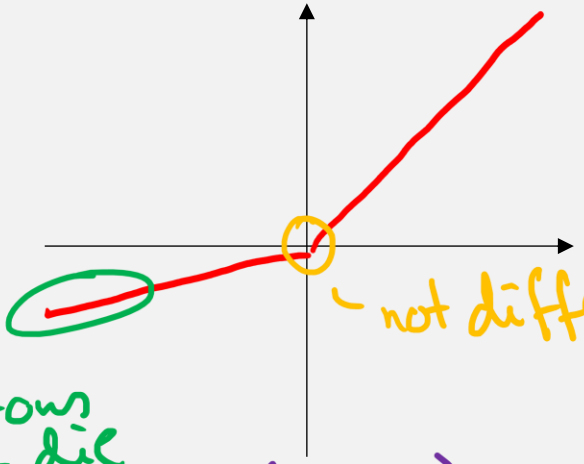
ACTIVATION FUNCTIONS



BETTER ACTIVATION FUNCTIONS

"exponential linear unit"

Leaky ReLU



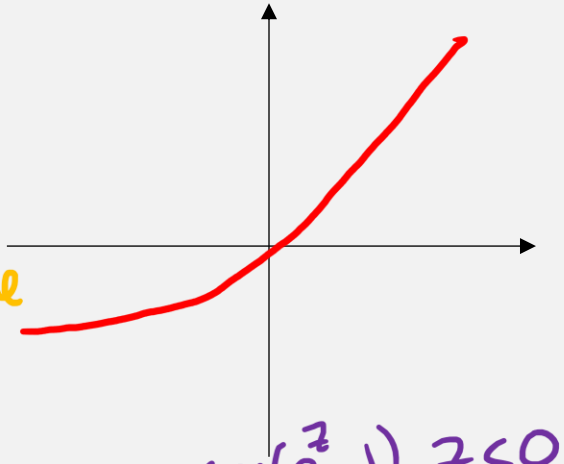
neurons don't die

not differentiable

$\max(\alpha z, z)$

↓ hyperparameter $0 < \alpha < 1$

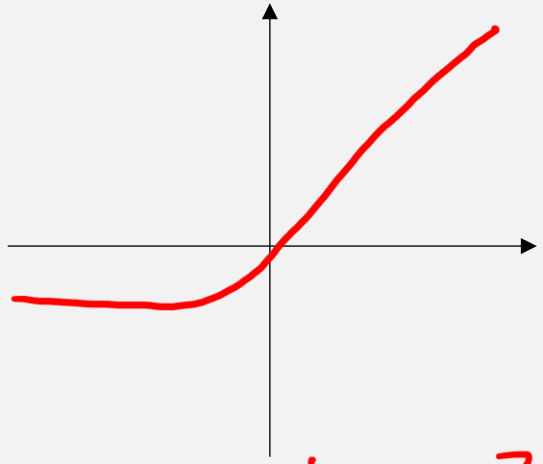
ELU



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

outperforms all variants of ReLU in training time for deep n.w.

Swish



$$z \cdot \frac{1}{1+e^{-z}} = z \cdot \text{sigmoid}(z)$$

outperforms everything else for deep networks

RECOMMENDATIONS

Try ReLU for shallow networks and Swish for deep networks

a good starting point

Leaky ReLU also a good choice

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

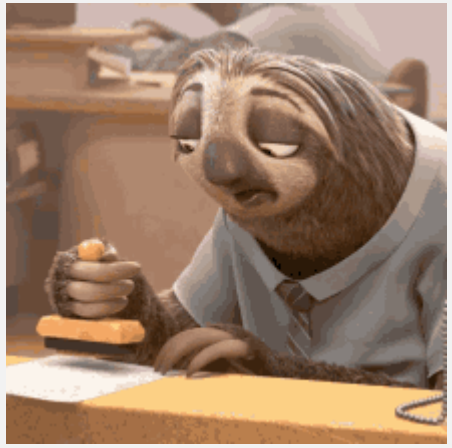
how the values of weights and biases are initialized

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



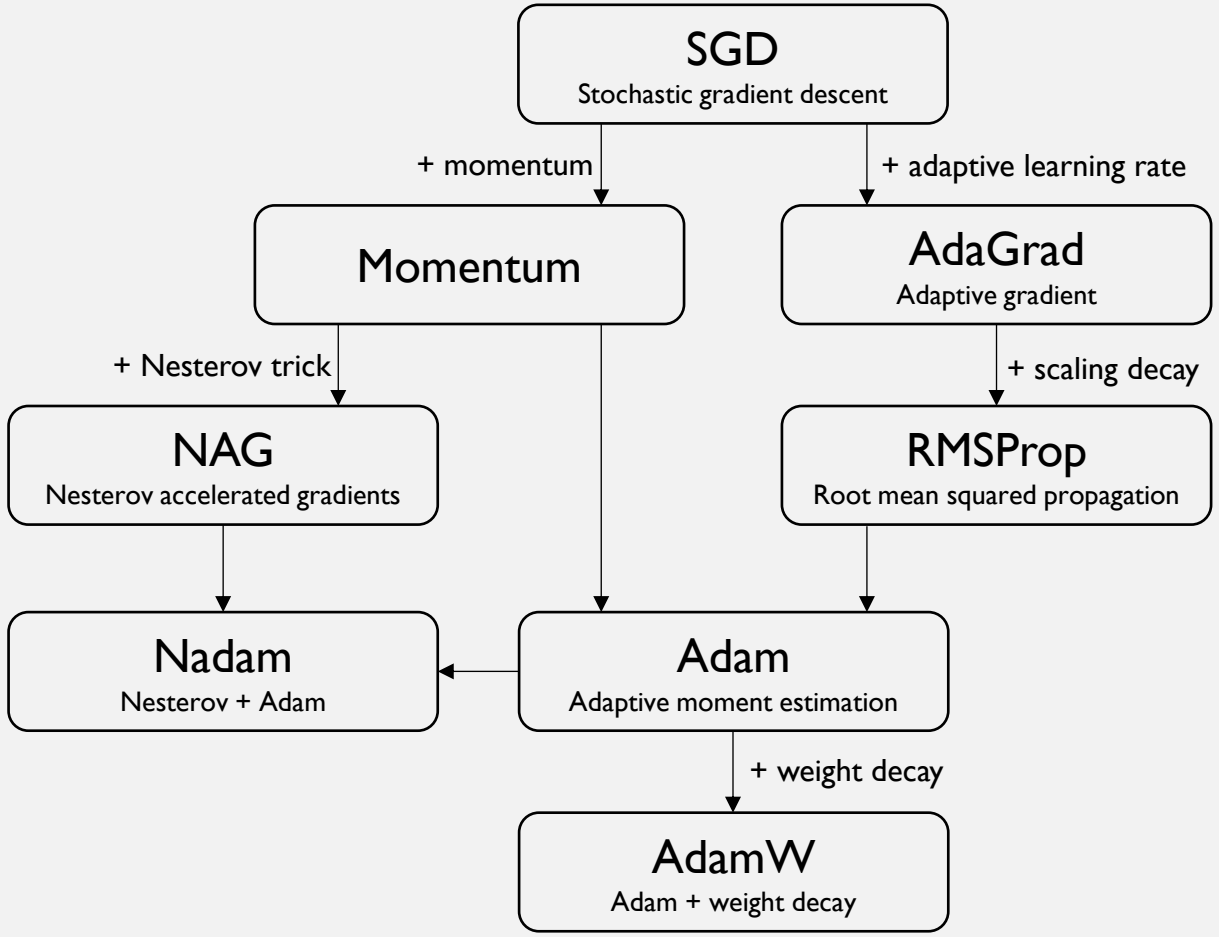
#GradientDescent
UpdatingParameters

#GradientDescent
FinishingAnEpoch

#GradientDescent
WhenItConverges

The point is: Gradient descent can be painfully slow

FASTER OPTIMIZERS



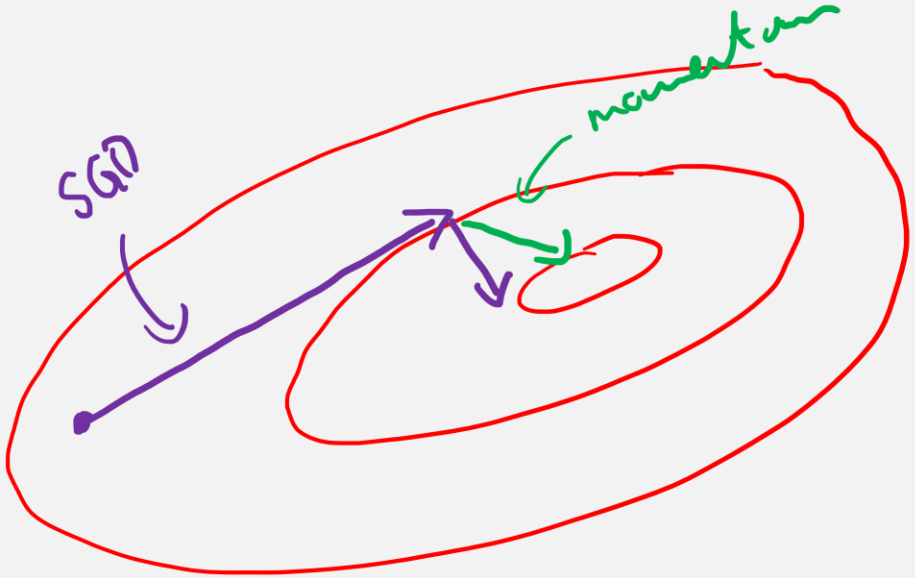
momentum
Nesterov trick
adaptive learning rate
scaling decay
weight decay

MOMENTUM

"the step we just took probably wasn't a terrible idea"

Gradient descent

$$\theta \leftarrow \theta - \gamma \nabla J(\theta)$$



Momentum

$$m \leftarrow \beta m - \gamma \nabla J(\theta)$$

momentum

hyperparameter

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

linear combination of current gradient & all previous steps

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

The trick works because m generally points in the right direction, so the gradient here is slightly more accurate

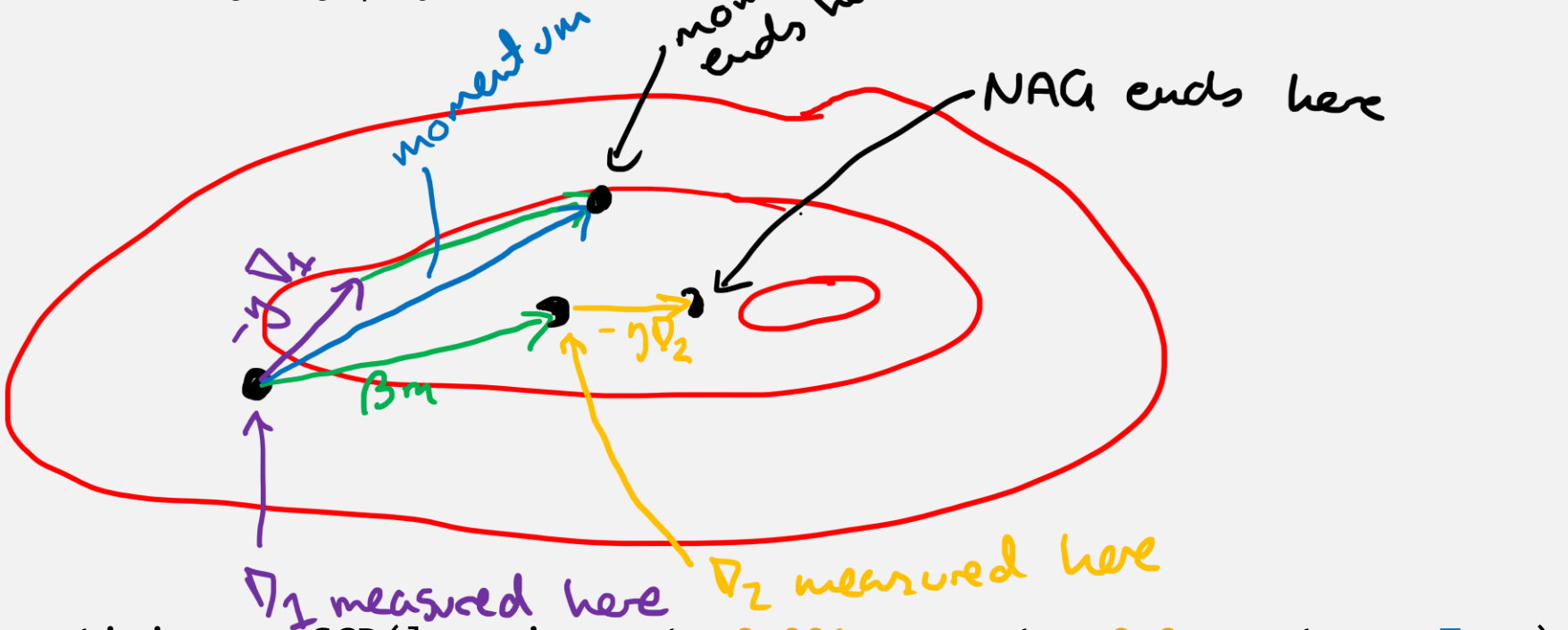
THE NESTEROV TRICK

Momentum

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

Nesterov

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

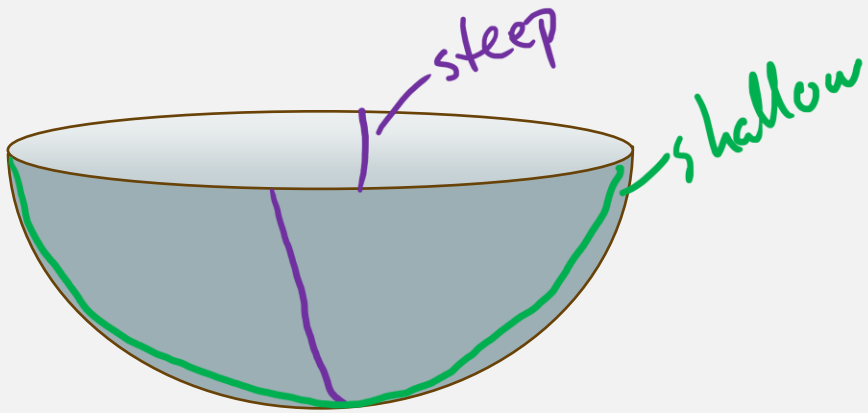


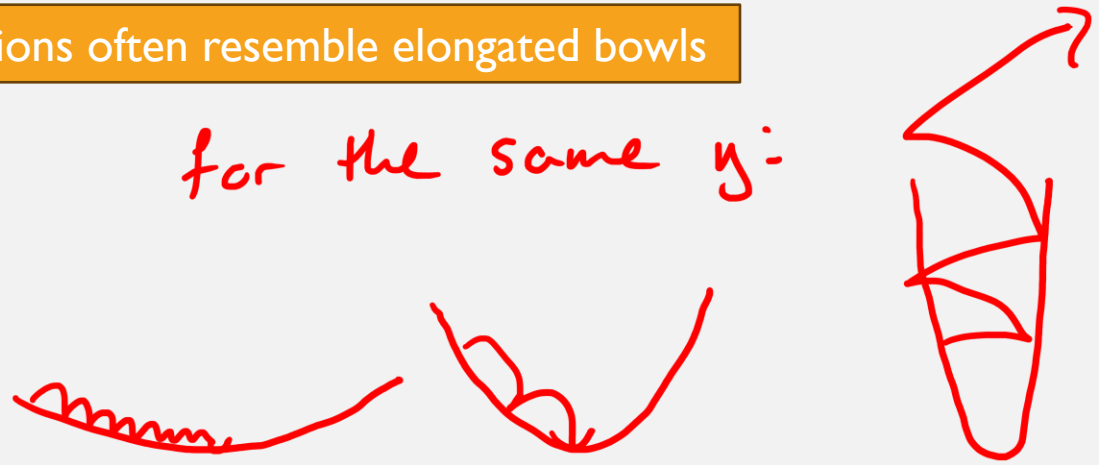

```
optimizer = SGD(learning_rate=0.001, momentum=0.9, nesterov=True)
```

momentum
Nesterov trick
adaptive learning rate
scaling decay
weight decay

ADAPTIVE LEARNING RATES

Observation: Loss functions often resemble elongated bowls





IDEA: make y smaller in the steepest directions

scale factor → $s \leftarrow s + \nabla_\theta L(\theta) \otimes \nabla_\theta L(\theta)$ → elementwise multiplications

$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta) \oslash \sqrt{s+\varepsilon}$ → $\varepsilon \sim 10^{-10}$ to avoid div by zero

AdaGrad ← η scaled down too much, so training tends to stop

momentum
Nesterov trick
adaptive learning rate
scaling decay
weight decay

SCALING DECAY

add decay to scale factor so it doesn't explode & stop training

AdaGrad

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \nabla \boldsymbol{L}(\boldsymbol{\theta}) \otimes \nabla \boldsymbol{L}(\boldsymbol{\theta})$$

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

RMSProp

$$s \leftarrow \rho s + (1-\rho) \nabla L(\theta) \otimes \nabla L(\theta)$$

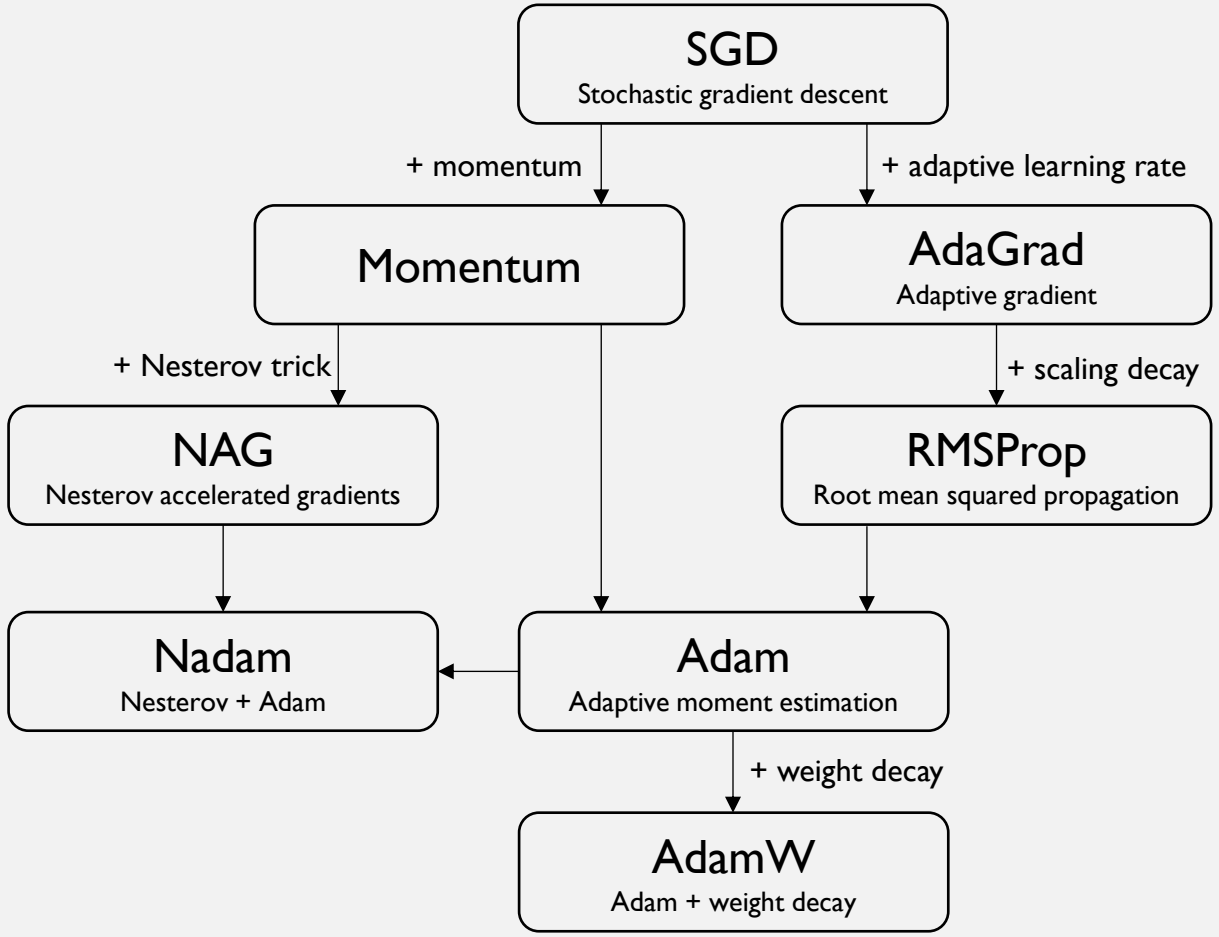
$$\theta \leftarrow \theta - \eta \nabla L(\theta) \oslash \sqrt{s+\varepsilon}$$

momentum
Nesterov trick
adaptive learning rate
scaling decay
weight decay

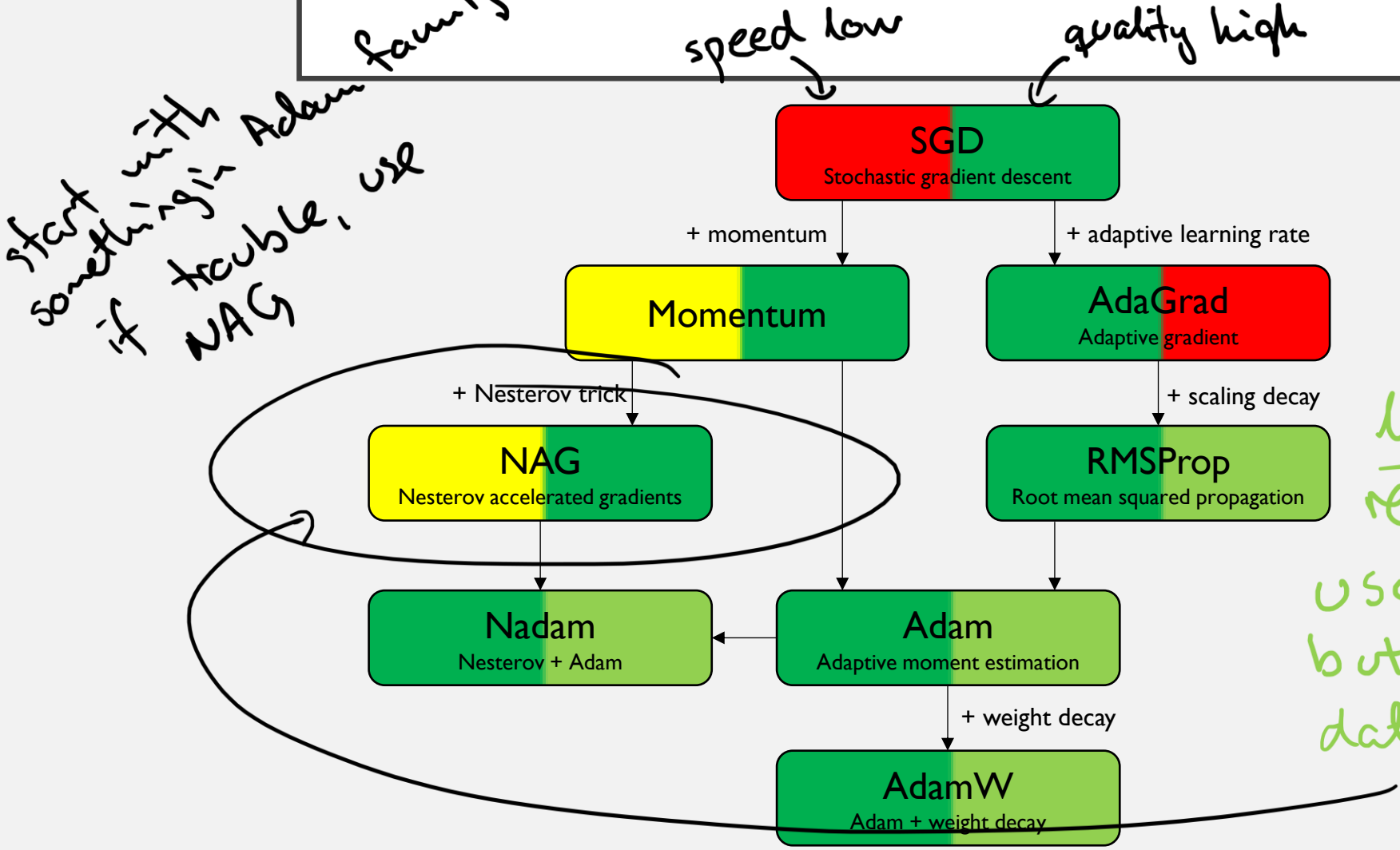
WEIGHT DECAY

At each training iteration, multiply all weights by say 0.99 - a form of built-in regularization

RECOMMENDATIONS



RECOMMENDATIONS

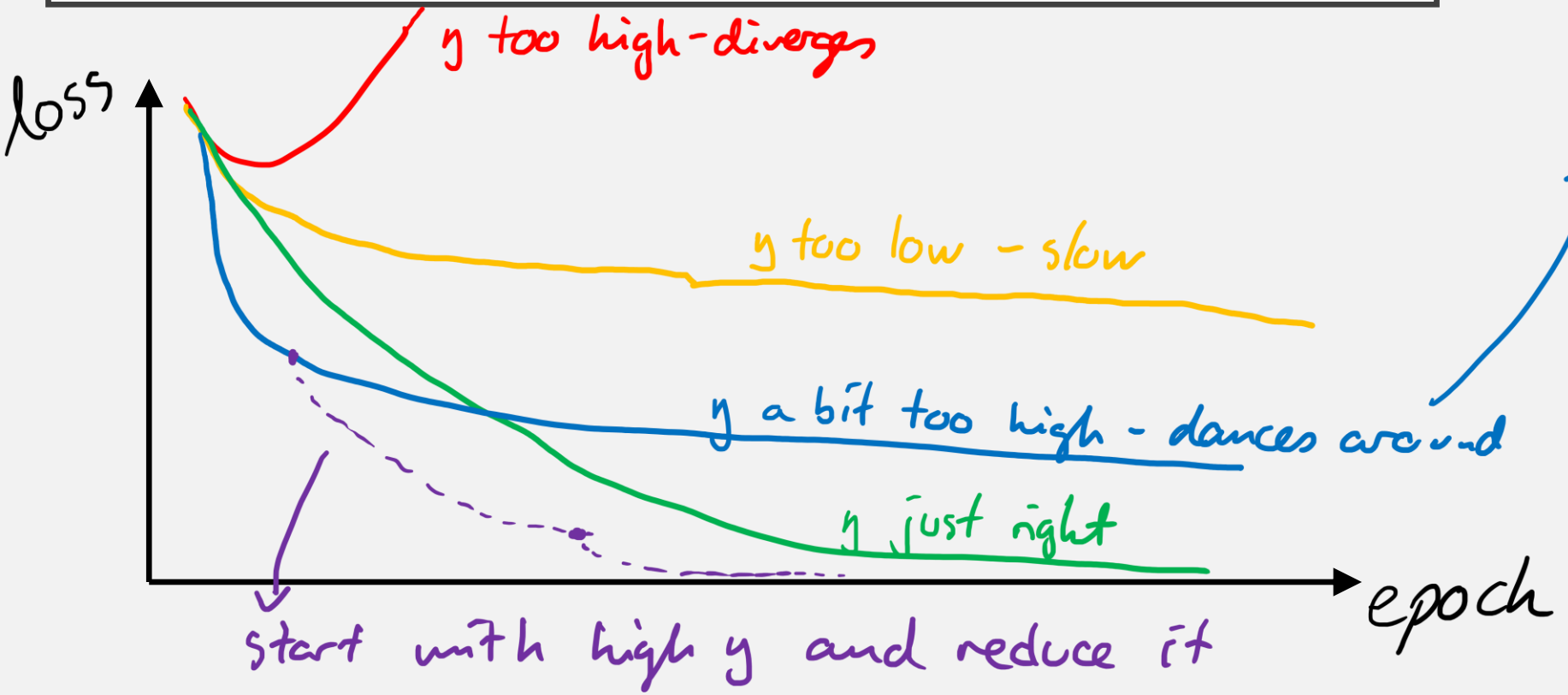


light green: really good... sometimes usually a good choice but "allergic" to some datasets

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

LEARNING RATE SCHEDULING

Power

hyper-parameters

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

iteration no.

After s steps: $\frac{\eta_0}{2}$

After $2s$ steps: $\frac{\eta_0}{3}$

⋮

Exponential

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

reduces by a factor of 10 every s steps

Piecewise constant

e.g.

$\eta = 0.1$ for 5 epochs

$\eta = 0.01$ for 10 epochs

$\eta = 0.005$ for 50

...

Performance

measure validation loss every N steps, divide η by λ when loss stops decreasing

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

Remember Lasso and Ridge regression?

TOOLS TO AVOID OVERFITTING

ADD Penalty for large coefficients

↳ $\alpha \sum_i \theta_i^2$ [L2]

$\alpha \sum_i |\theta_i|$ [L1]

TO LOSS FUNC.

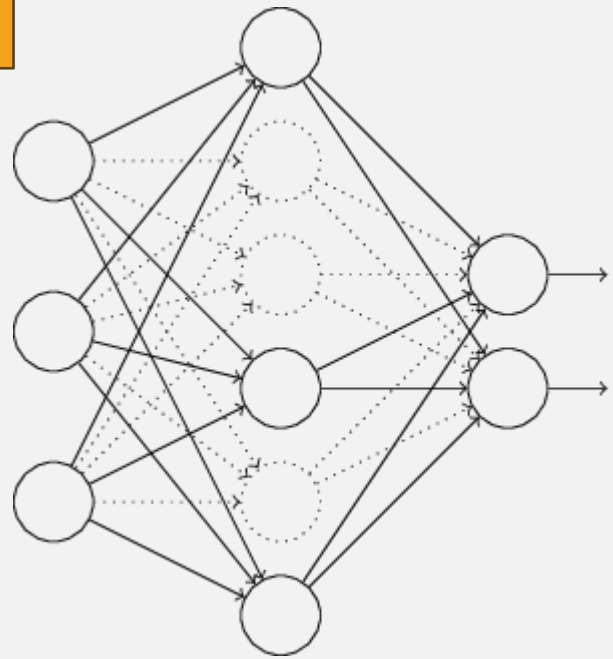
NEVER DO THIS WITH ADAM-TYPE OPTIMIZERS (use AdamW instead)

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

DROPOUT REGULARIZATION

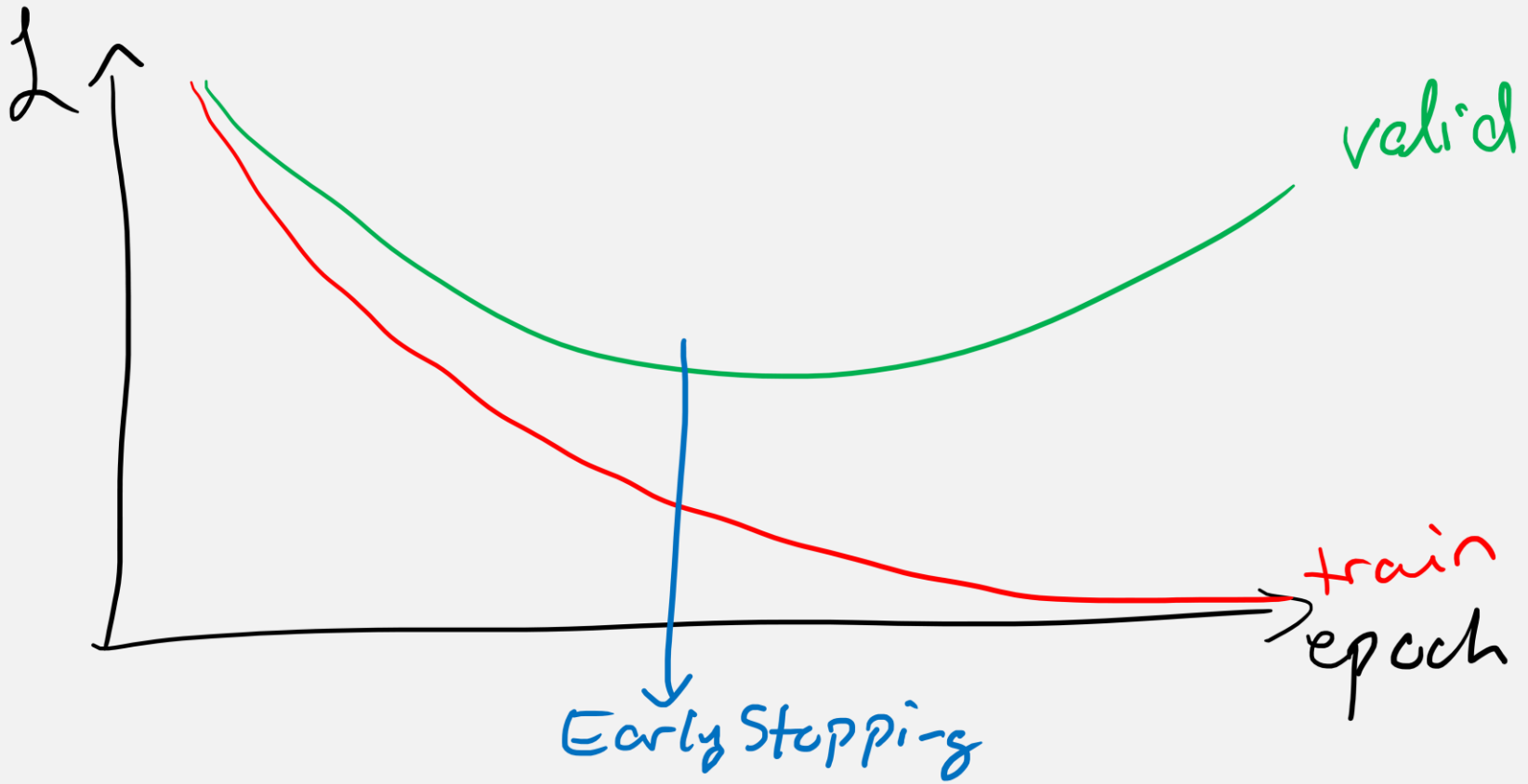
At every training step, every neuron has a probability p of being entirely ignored!

and 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

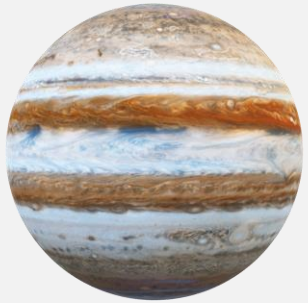
- How training a neural network works
- Activation functions
- Faster optimizers
- Learning rate scheduling
- Regularization
- General suggestions

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