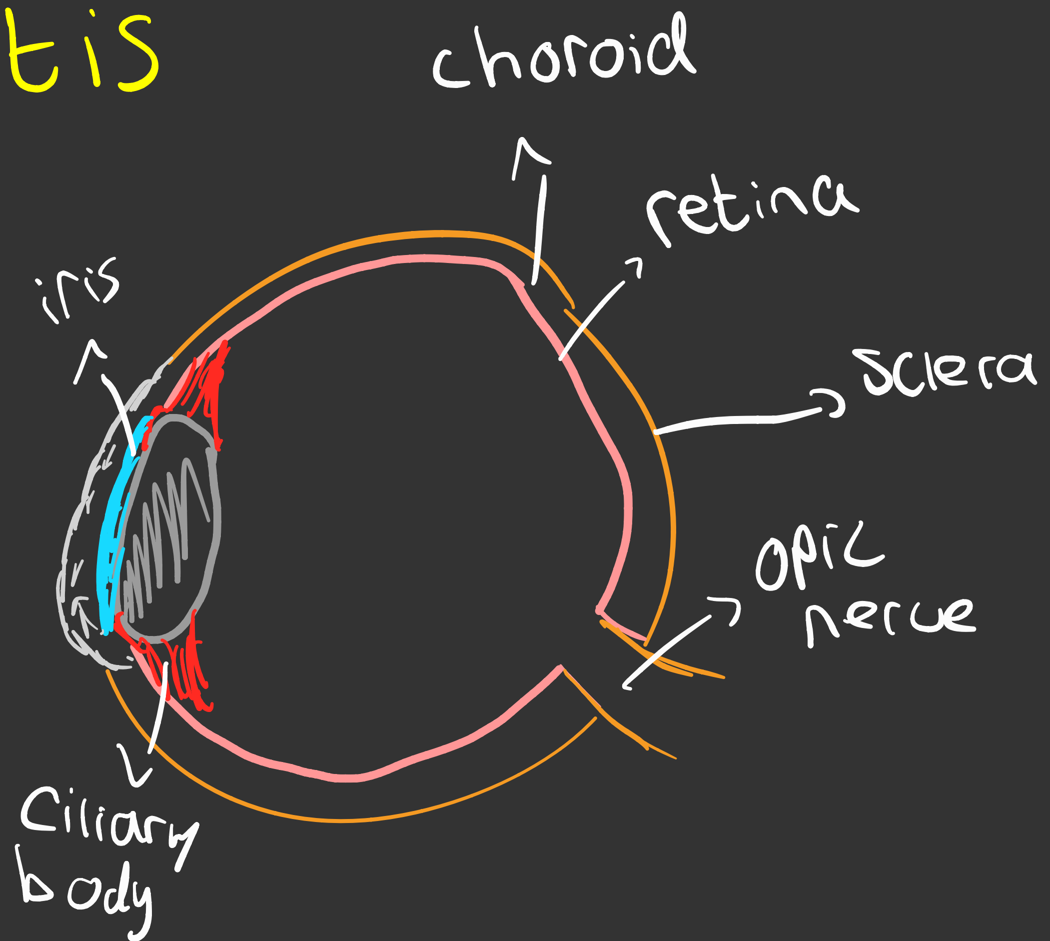
Uveitis

Uveitis is a form of eye inflammation that affects the middle layer of tissue in the eye, called the uvea. Different types of uveitis,

- Anterior, affects the inside of the front of your eye, also called iritis.
- Intermediate, affects the retina and blood vessels just behind the lens.
- Posterior, affects a layer on the inside of the back of the eye, either retina or Choroid.
- Panuveitis, when all layers of the uvea are inflamed, from the front to the back of the eye



uvea = Iris, ciliary body, Choroid

Vitreous = gel in the center of the eye.

Some inflammatory signs:

- macular edema, this is defined as an abnormal thickening of the macula associated with accumulation of fluid in the extracellur space of the outer plexiform layer.
- Optic disc hyperfluorescence, is an increased brightness of the optic nerve head during fluorescein angiography.
- retinal vascular staing and/or leakage (in posterior pole) are key indicators of a breakdown in the blood-retinal barrier.
- capillary leakage (in posterior pole), involves fluids, proteins, and cells leaking from retinal blood vessels.

  ↳ posterior pole: central back of the eye

macula

small yellowish-oval spot at the center of the retina in the back of the eye, responsible for sharp, detailed and central vision.

extracellular space

the area outside of cell membranes in a multicellular organism, filled with fluid, nutrients and signaling molecules

Outer Plexiform layer (OPL)

a crucial synaptic zone in the retina, located between the outer nuclear layer (photoreceptors) and the inner nuclear layer.

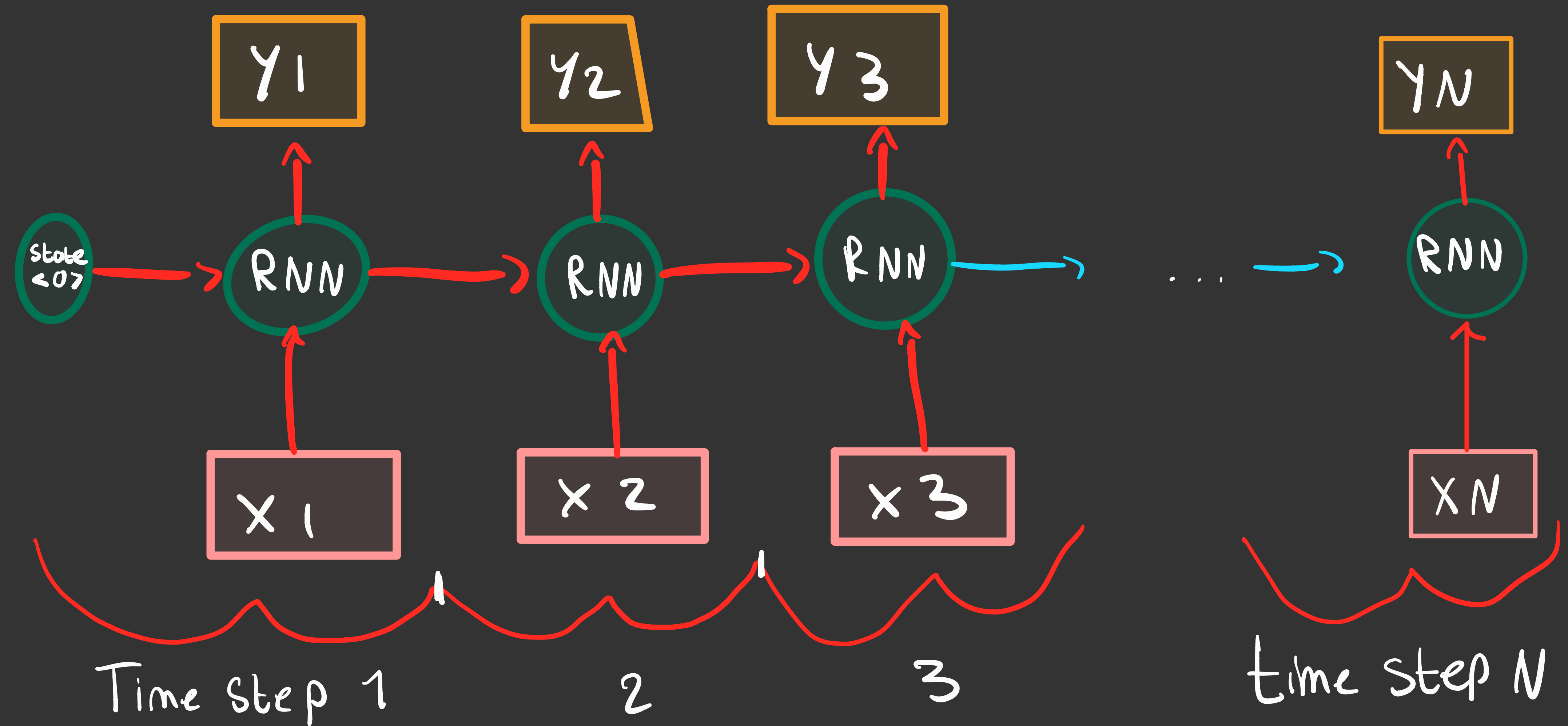
Optic nerve head (ONH)

structure at the back of the eye where retinal ganglion cell axons exit to form the optic nerve and ret. blood vessels enter.

Transformer

(attention is all you need)

- first there were Recurrent Neural Networks



- Worked fine but had some problems
  - -> Slow computations for long sequences
  - -> Vanishing or exploding gradients

$$\frac{dg}{dx} = \frac{dg}{df} \cdot \frac{df}{dx} =$$

$0.5 \cdot 0.5 \rightarrow \frac{1}{4} \rightarrow$ vanishing

$2 \cdot 2 \rightarrow 4 \rightarrow$ exploding

- -> difficulty in accessing information from a long time ago

Introducing the Transformer

2 macro blocks: Encoder and decoder, they are connected to eachother.

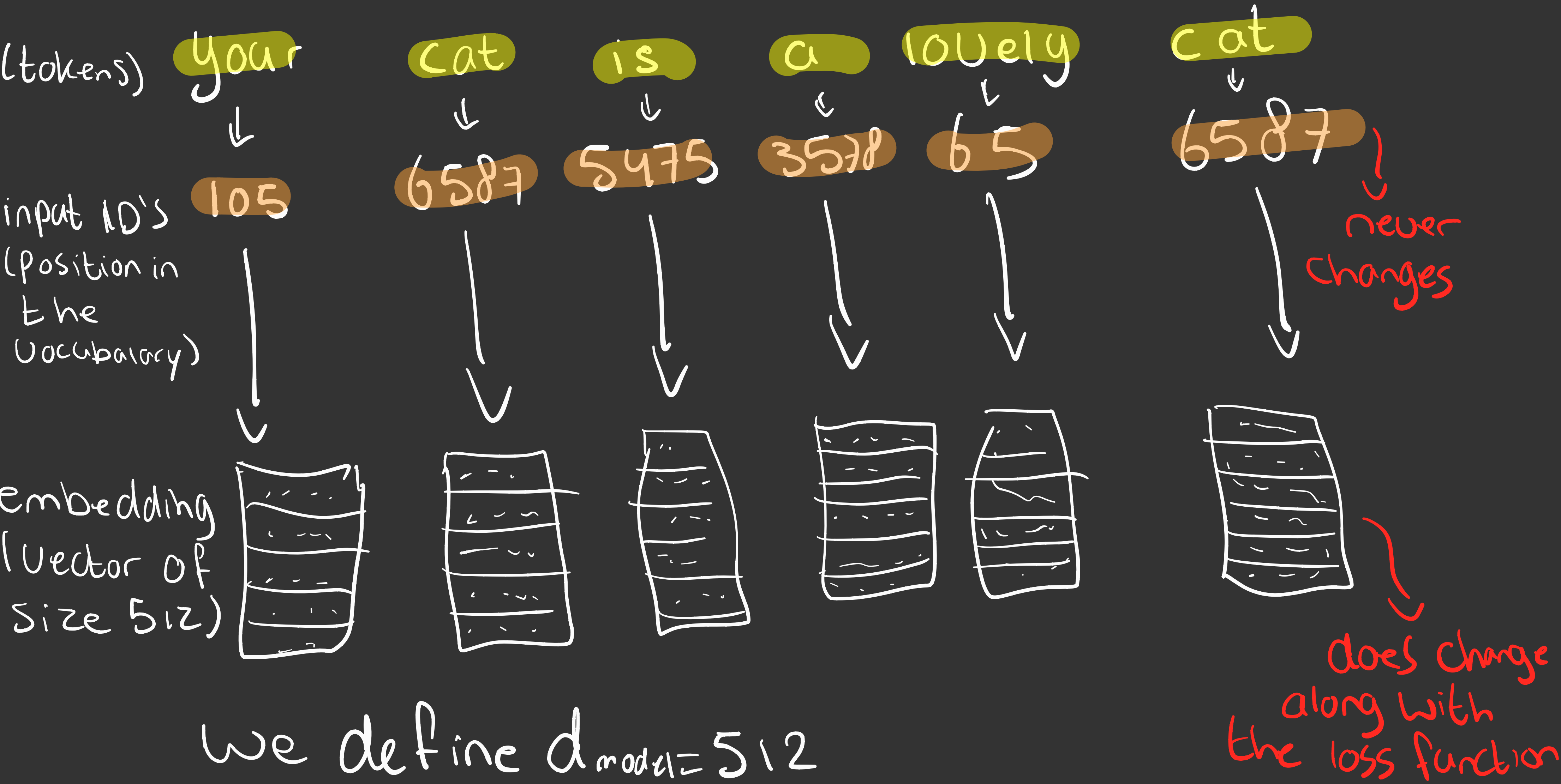
Input matrix (sequence, d_{model})

(6, 512) × (512, 6) = (6·6)

A, B, C, D, E, F (rows, 512 columns) × A B C D E F (columns, 512 rows) = AA AB AC ... AF

Encoder

Input embedding:



We define $d_{model} = 512$

Positional encoding

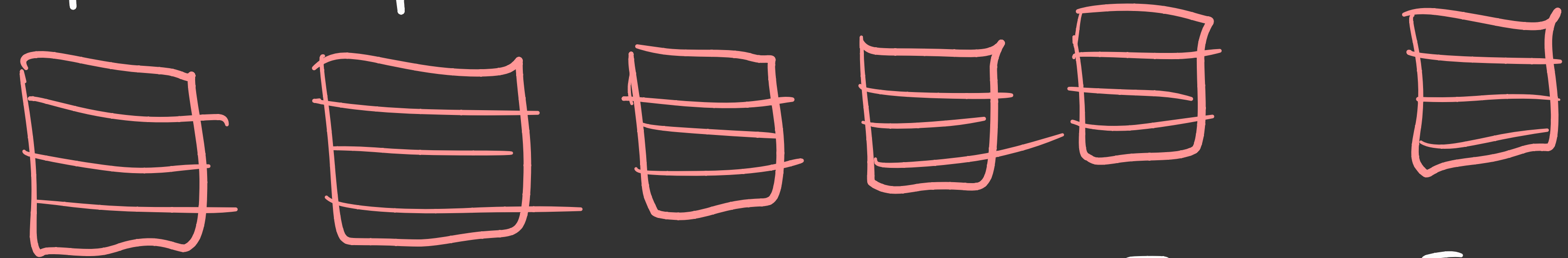
- We want each word to carry some information about its position in the sentence
- We want the model to treat words that appear close to each other as "close" and words that are distant as "distant"
- We want the positional encoding to represent a pattern that can be learned by the model

Sentence your cat is a lovely cat

embedding

Vector of size 512

Position embedding



Only computed once and reused for every sentence during training and inference

Vector of size 512

encoder input

even dimensions

$$PE(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

$$PE(pos, 2i+1) = \cos \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

uneven dimensions

We only need to compute the positional encoding once and then reuse them for every sentence, no matter if it is training or inference

Why trigonometric functions?

-> trig. functions like cos & sin naturally represent a pattern that the model can recognize as continuous, so relative positions are easier for the model.

Single-head attention

Self-attention!

- allows the model to relate words to eachother
- consider seq len. = 6 and $d_{model} = 512 = d_k$
- The matrices Q, K and V are just the input sentence

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right)V$$

$$\text{softmax}\left(\frac{\underset{(6,512)}{Q} \times \underset{(512,6)}{K^t}}{\sqrt{512}}\right) =$$

dot product

	your	cat	is	a	lovely	cat
your	0.268					
cat						
is						
a						
lovely						
cat						

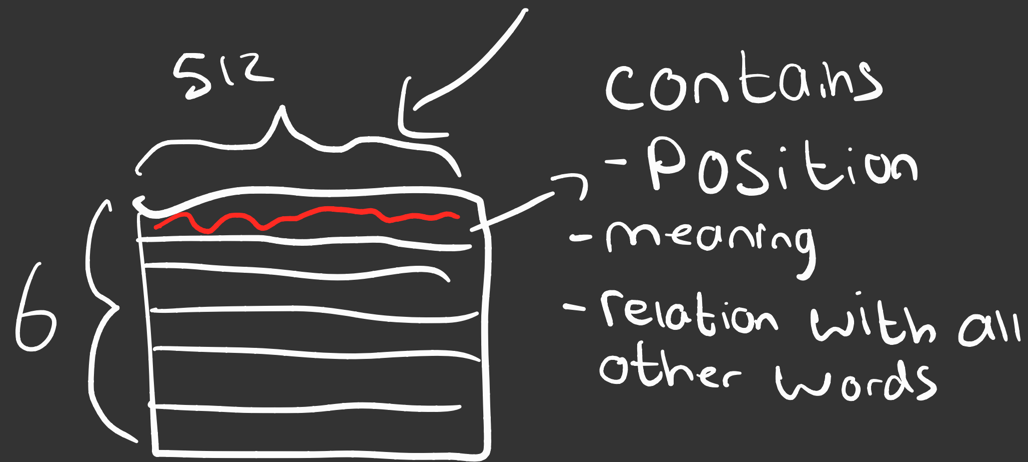
(6,6), values of all rows sum up to 1 (softmax

dot product

	Your	cat	is	a	lovely	cat
Your	0.268					
cat						
is						
a						
lovely						
cat						

(6,6), values of all rows sum up to 1
(softmax

× V (6, 512) = Attention (6, 512)



multi-head attention

seq = Sequence length

d_{model} = Size of emb. V

h = N of heads

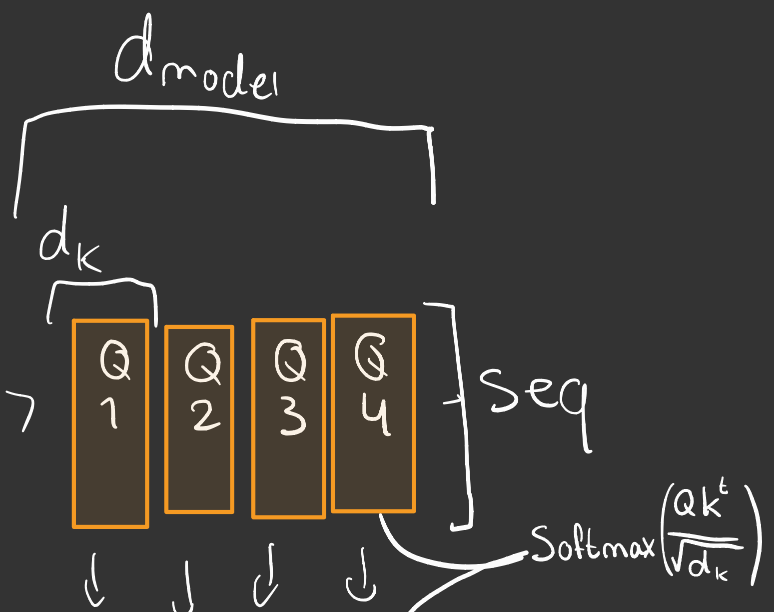
$d_k = d_v = d_{model}/h$

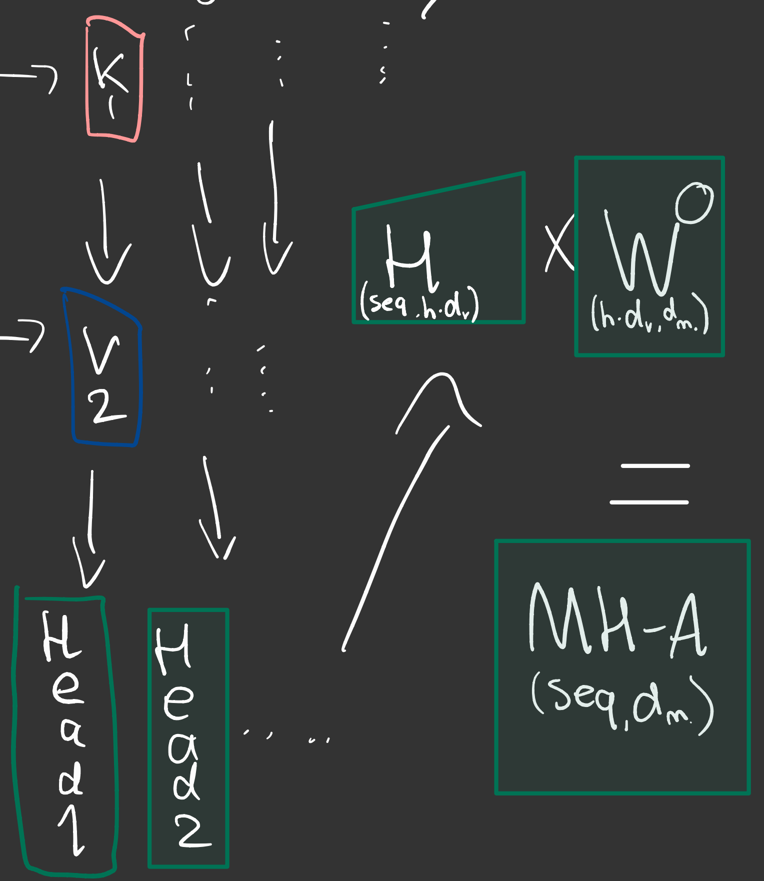
input (seq, d_{model})

Q (seq, dmodel) × W^Q (d_m, d_m) = Q' (seq, dmodel)

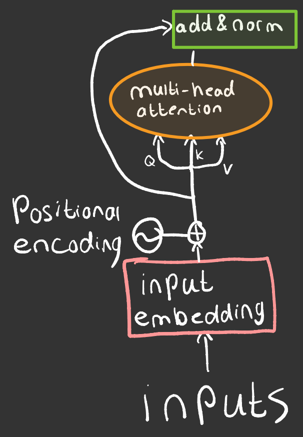
K (seq, dmodel) × W^K (d_m, d_m) = K' (seq, dmodel)

V (seq, dmodel) × W^V (d_m, d_m) = V' (seq, dmodel)





$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right)V$$



$$\text{multihead}(Q,K,V) = \text{Concat}(\text{head}_0, \dots \text{head}_h)W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

every heads learns to know "words" as different things, verbs and nouns for example.

WHY query, keys and values?

Linear Discriminant Analysis (LDA

LDA works by finding directions in the feature space that best separate the classes.
It does this by maximizing the difference between the class mean $\bar{x}$ while minimizing the spread within each class.

assume we have 2 classes with d-dimensional samples such as $x_1, x_2, \ldots, x_n$ where:

n_1 samples belong to class C_1

n_2 samples belong to class C_2

If x_i represents a data point, its projection onto the line represented by the unit vector $\vec{v}$ is $v^T x_i$.

Let the means of C_1 and C_2 before projection be μ_1 and μ_2 respectively.

After projection, the new means are $\hat{\mu}_1 = v^T \mu_1$ and $\hat{\mu}_2 = v^T \mu_2$.

Our aim is to normalize the difference $|\hat{\mu}_1 - \hat{\mu}_2|$ to maximize the class separation.

The scatter for samples of C_1 is calculated as:

$$s_1^2 = \sum_{x_i \in C_1} (x_i - \mu_1)^2$$

and C_2:

$$s_2^2 = \sum_{x_i \in C_2} (x_i - \mu_2)^2$$

goal is to maximize the ratio of the between-class scatter to the within-class scatter, which leads us to the following criteria:

$$J(v) = \frac{|\hat{\mu}_1 - \hat{\mu}_2|}{s_1^2 + s_2^2}$$

Break!

means = where each class is centered

scatter = how spread out each class is.

J(v) = a score saying "good separation, divided by bad overlap.

eigenvectors = the directions that optimize that score

What does "Project onto a direction v" mean?

suppose each data point is a vector
$x \in \mathbb{R}^d$,

Pick a directon v

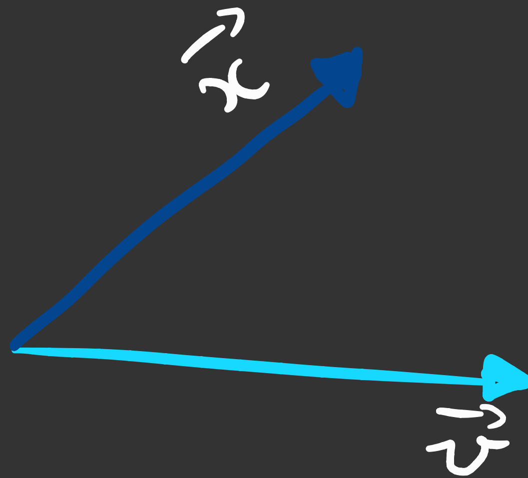
then the projection of x onto that direction
is $z = v^T x$

→ turns every d-dimensional point into a single number z, so instead of seeing two clouds in 2D or 10D, you see two groups of points on a line

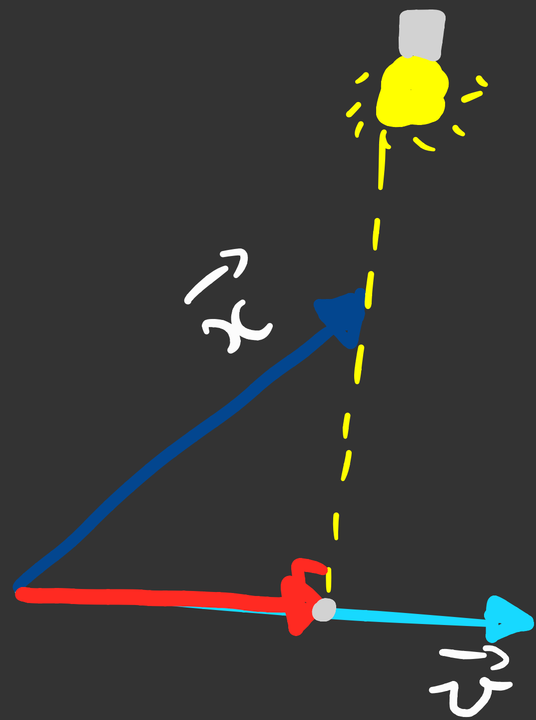
So LDA is asking:

"Which direction v makes those projected classes easiest to separate?"

Projection



the red vector is the projection of x onto v = $\text{proj}_{\vec{v}}\, x$



Vector Parallel to v with magnitude

$\frac{\vec{x} \cdot \vec{v}}{\|\vec{v}\|}$ in the direction of $\vec{v}$ is called Projection of $\vec{x}$ onto $\vec{v}$

$$\left[\text{Proj}_{\vec{v}} \vec{x} = \frac{\vec{x} \cdot \vec{v}}{\|\vec{v}\|^2} \vec{v} \right]$$

formula of red Vector

Scalar Projection

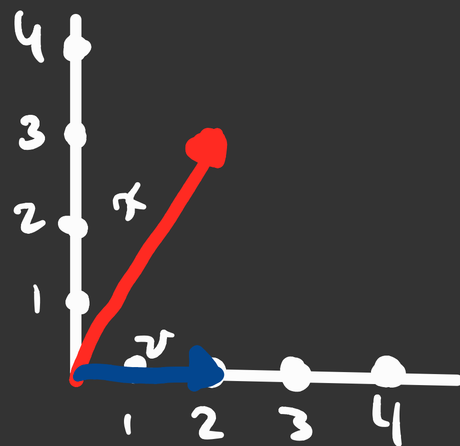
tells you: "how far along the direction u the point x lies." $\left[u^T x \right]$ formula

this is just a number

example

$x = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$, $v = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \rightarrow v^T x = \begin{bmatrix} 2 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} \rightarrow 4$,

but $\|v\| = 2$, so $\frac{4}{2} = 2$ scalar projection length



$\vec{a} = \begin{bmatrix} 2i \\ -j \\ k \end{bmatrix}$ $\quad \vec{b} \begin{bmatrix} i \\ -2j \\ k \end{bmatrix}$ $\quad 1$

$\begin{bmatrix} i & -2j & k \end{bmatrix} \begin{bmatrix} 2i \\ -j \\ k \end{bmatrix} \rightarrow 2+2+1 = 5$

exercise

$$A = \{ \overset{1 \;\; \frac{3}{\sqrt{2}}}{(1,2)},\ \overset{2 \;\; \frac{3}{\sqrt{2}}}{(2,1)},\ \overset{2 \;\; \frac{4}{\sqrt{2}}}{(2,2)} \}$$

$$B = \{ \overset{4 \;\; \frac{7}{\sqrt{2}}}{(4,3)},\ \overset{5 \;\; \frac{9}{\sqrt{2}}}{(5,4)},\ \overset{4 \;\; \frac{9}{\sqrt{2}}}{(4,5)} \}$$

$u_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ (horizontal)

$u_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ (diagonal)

$$\sqrt{3.5^2 + 3.5^2}$$

Okay, now that is out of the way,

once you project each class onto a line, each class becomes just a list of numbers

example

class A Projects to 1, 2, 2
class B Projects to 4, 5, 4

then the mean is its center on that line, and the mean of class B is its center on that line.

So the mean tells you:

"Where the average shadow of each class lands"

if those mean locations are far apart, that is good.

Since the projected data are just numbers, scatter means:

"How spread out are the projected points around their class mean?"

For one class with projected values $z_1, z_2, \ldots z_n$, and projected mean $\hat{\mu}$, the scatter is:

$$\left[\sum_i (z_i - \mu)^2\right]$$

this is the total squared spread around the center

So: small scatter = class shadows are tight
Large scatter = class shadows are smeared out

that is why scatter matters: even if class means are far apart, very large scatter can make the classes overlap.

So LDA wants a direction $\vec{v}$ where:

distance between projected class means is large

and

within-class scatter is small

→ so it wants separation, but not a noisy one.

So now we also know why we want

$$J(v) = \frac{|\hat{\mu}_1 - \hat{\mu}_2|}{s_1^2 + s_2^2}$$

→ big as possible (numerator)

→ small as possible (denominator)

to be as big as possible, but how do we make this happen?

→ rewrite it as an optimalization problem and solve

$$v \propto S_W^{-1}(\mu_1 - \mu_2)$$

this "sort of" says:

"the best projection direction v is approximately the direction of the difference between the two class means, corrected by the spread within the classes."

v is the LDA-line direction

$\propto$ means it is parralel to something

$\mu_1 - \mu_2$ is the vector from one class center to the other

So $v \propto \mu_1 - \mu_2$ would already make some sense, but LDA is smarter.

S_w is the within class scatter matrix, so a description of how far the points within a class are spread.

$$S_w = S_A + S_B$$

$$S_A = \sum_{x_i \in A} (x_i - \mu_A)(x_i - \mu_A)^T$$

So first the class mean is calculated (μ_A).

After that, the deviatings are being calculated using $x_i - \mu_A$. but because we can't just square a vector, we use the outer product ($\vec{v} \cdot \vec{u}^T$) then adding all of those results up ends up being our S_A

We do the same for S_B and then obtain by adding up S_A and S_B

eventually we end up with something looking like this:

$$S_W = \begin{bmatrix} 4/3 & -1/3 \\ -1/3 & 8/3 \end{bmatrix}$$

Interpretation:

top left: total internal spread in x-direction

bottom right: total internal spread in the y-direction

off-diagonal: slight relationship between x- and y-deviations

so here the within class spread is bigger in y than in x.

That already tells LDA:

"Vertical differences are noisier than horizontal ones"

Why this matters for LDA:

When you later compute $v^T S_w v$,

you are asking: "If I project onto direction v, how much within-class spread do I see along that direction?"

so S_w is like a machine that knows the internal spread in every possible direction. Hence why it is a matrix and not just a number.

now it is time for a StatQuest video to help me further understand this, the basics are known now.

what I picked up from the video:

- How LDA creates a new axis:

2 criteria

1. maximize the distance between means.
2. minimize the scatter within each category

$$\frac{(\mu - \mu)^2}{s^2 - s^2}$$

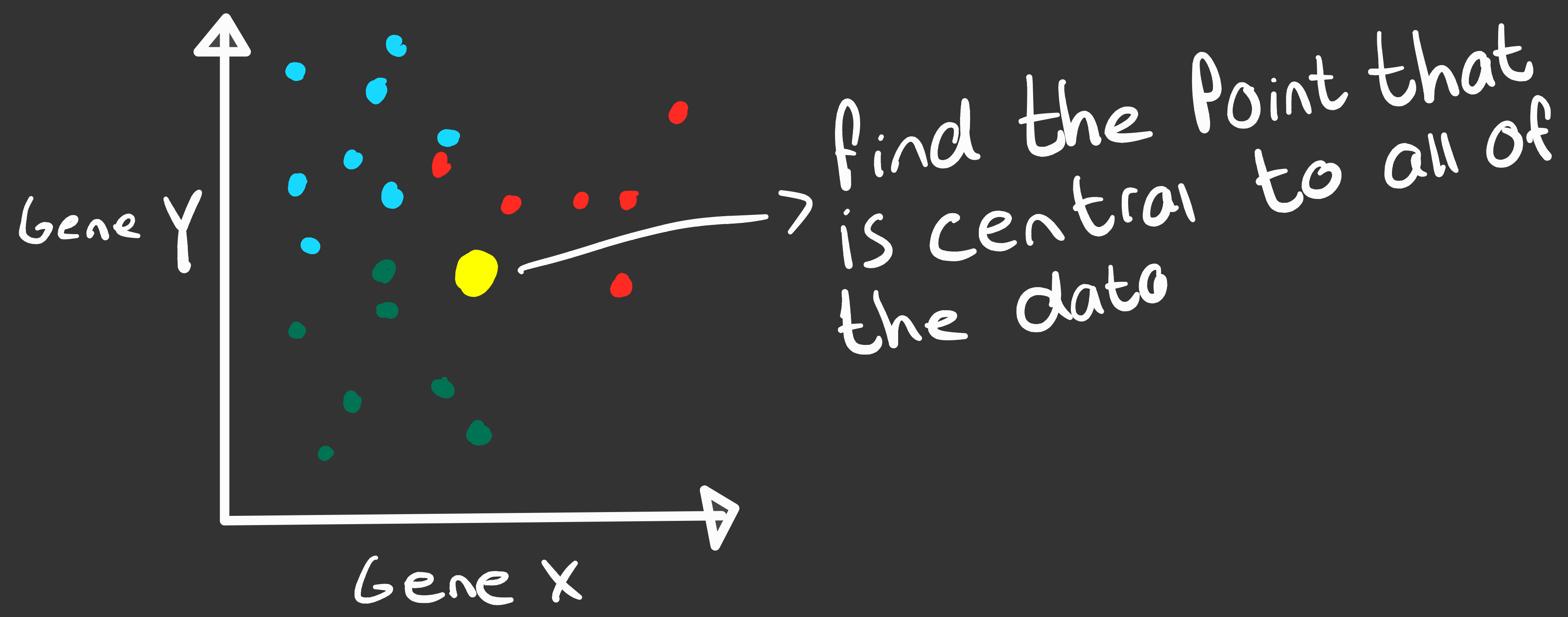
$(\mu - \mu)^2$ → ideally big → d^2

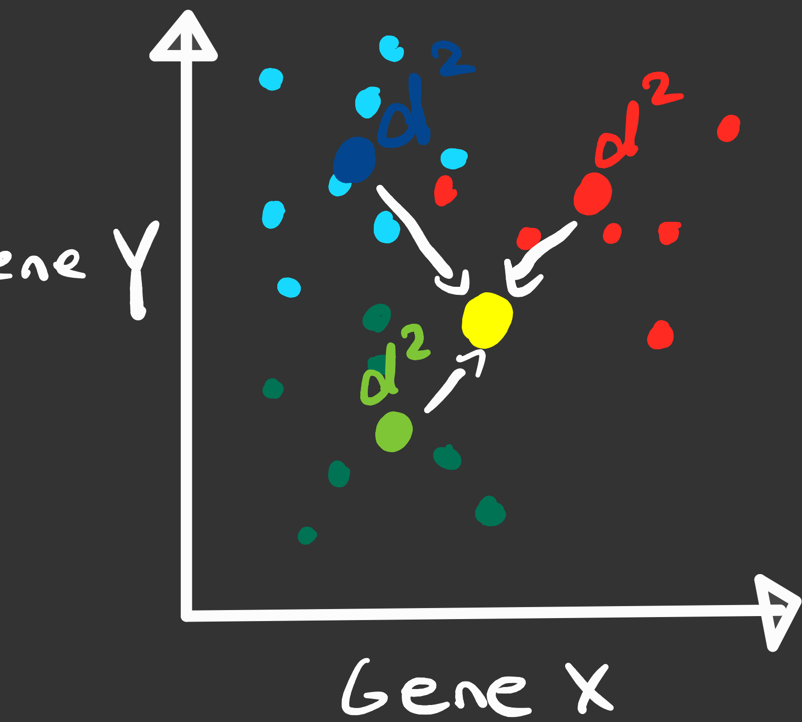
$s^2 - s^2$ → ideally small

its important we use both criteria for the best results

moving from 2 to 3 categories

- first difference is how you measure the distances among the means



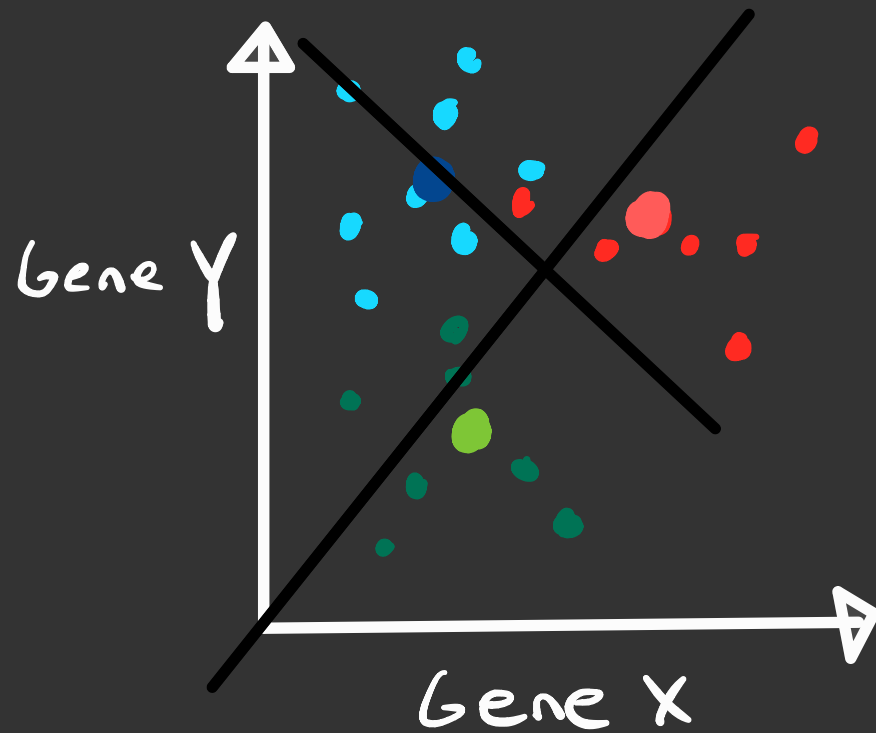


then measure the distances between a point that is central in each category and the main central point.

$$\frac{d^2 + d^2 + d^2}{s^2 + s^2 + s^2}$$

now maximize the distance between each category and the central point while minimizing the scatter for each category.

- Second difference, is that LDA creates 2 axes to separate the data. This is because the 3 central points for each category define a plane.



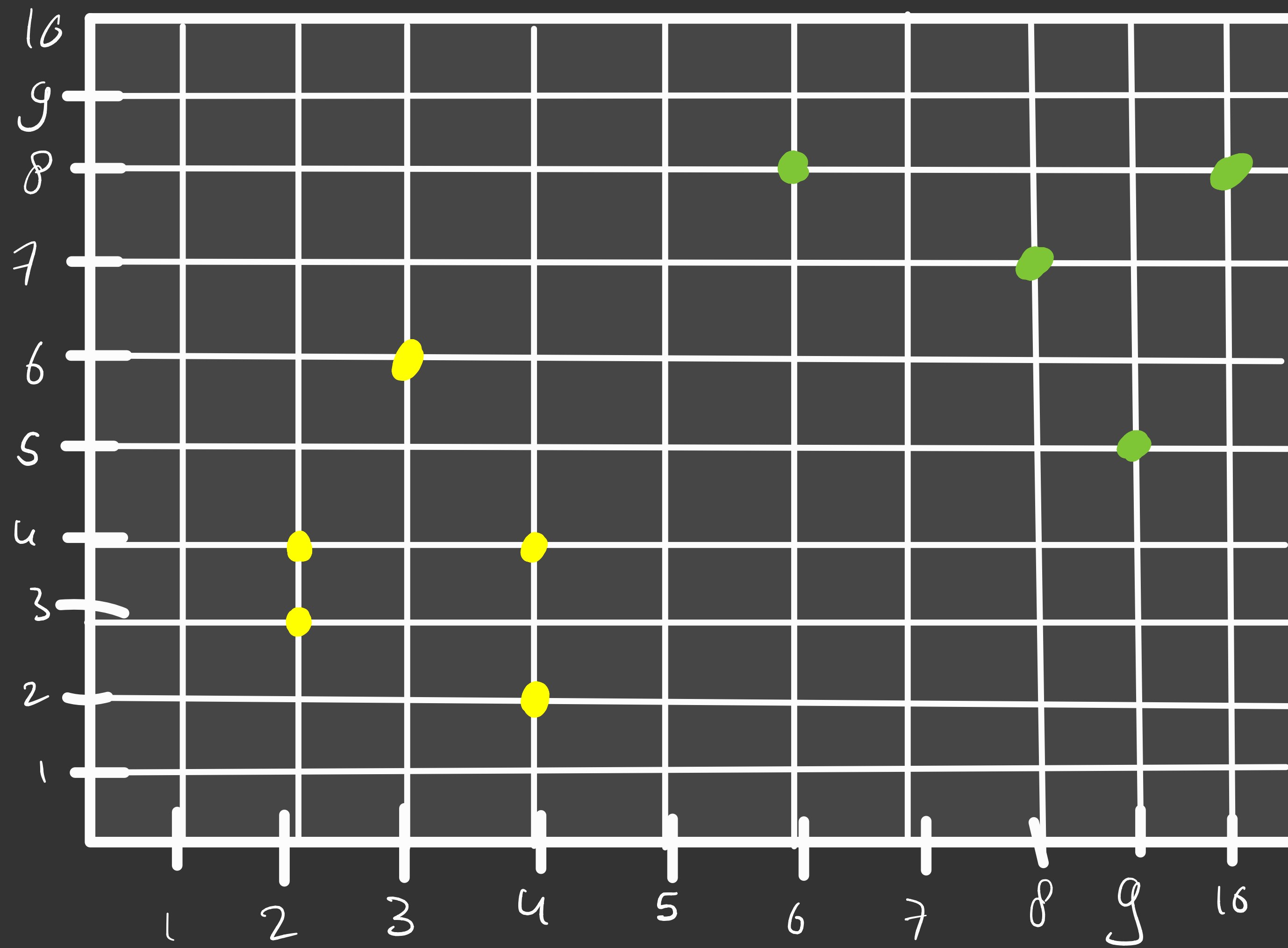
With three points, we can draw two lines to optimize separation

example

class ω_1: $X_1 = (x_1, x_2) = \{(4,2), (2,4), (3,6), (4,4)\}$

class ω_2: $X_2 = (x_1, x_2) = \{(9,10), (6,8), (9,5), (8,7), (10,8)\}$

We can plot these points into the following graph:

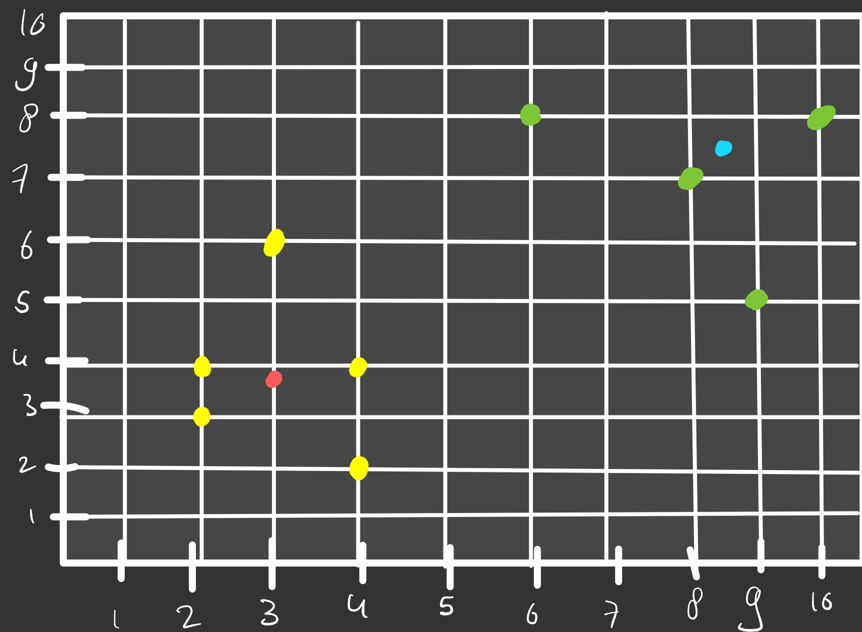


how can we convert this into a new 1D axes?

1. calculate the means

$$\mu_1 = \begin{pmatrix} 3 \\ 3.8 \end{pmatrix}$$

$$\mu_2 = \begin{pmatrix} 8.4 \\ 7.6 \end{pmatrix}$$



2. calculate covariance matrix

$$S_1 = \sum_{x \in w_1} \frac{(x-\mu_1)(x-\mu_1)^T}{N-1} \rightarrow = \begin{pmatrix} 1 & -\frac{1}{4} \\ -\frac{1}{4} & 2.2 \end{pmatrix}$$

$$\left[\begin{pmatrix}4\\2\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]\left[\begin{pmatrix}4\\2\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^T$$

mat. mul.

$$\begin{pmatrix}1\\-1.8\end{pmatrix} \cdot \begin{pmatrix}1 & -1.8\end{pmatrix} = \begin{pmatrix}1 & -1.8\\-1.8 & 3.24\end{pmatrix}$$

$$S_2 = \begin{pmatrix}2.3 & -0.05\\-0.05 & 3.3\end{pmatrix}$$

$$S_w = S_1 + S_2 = \begin{pmatrix}3.3 & -0.3\\-0.3 & 5.5\end{pmatrix} = S_w$$

3. Calculate between class scatter matrix

$$S_B = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$$

$$= \begin{pmatrix} 29.16 & 20.52 \\ 20.52 & 14.44 \end{pmatrix}$$

4. find the eigen Values

$$\left[S_W^{-1} S_B w = \lambda w \right] \rightarrow S_W^{-1} S_B - \lambda I = 0$$

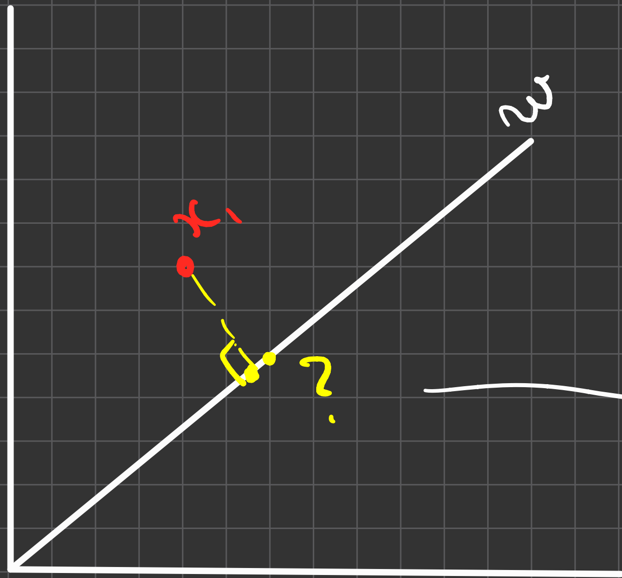
the eigen Values is a number that tells you:

"If I apply this transformation in a special direction, how much does it stretch or Shrink that direction?"

for now it is somewhat clear, it might be time to see what this would look like in our 384D embedding space. With n_bins amount of classes and using different types of foundation models.

things I should get straight

- Final research question + Primary hypothesis
- Picking metric(s), Spearmann correlation?
- Clear inclusion/exclusion



$$\to \text{Proj}_w(x) = \frac{x^T w}{w^T w} w$$

$$\frac{11}{2}[1 \ 1] = [5\tfrac{1}{2}, 5\tfrac{1}{2}]$$

to get the point on the 1D line, first normalize w to $u \to u = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}$ and then $x^T u \to [4 \ 7][\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}] = \frac{11}{\sqrt{2}}$ on the 1D line

- get random results.
- explore recent papers on encoding. temporal information. dont reinvent the wheel.
- iterating is a good method, for better control.
- validate with 2 sequenced things.
- more sanity check.

André's idea

PCA Tables

RETFound_Green (384D)

coverage	n_pc	AUC	AP
95%	111	0.78	0.49
99%	235	0.78	0.50

→ overfitting, hits ceiling at 0.78

→ overfitting at epoch 30

Baseline raw

Seed	AUC	AP
42	0.81	0.53
1	0.86	0.50
2	0.79	0.49
69	0.81	0.52

→ Baseline

AUC = 0.8021 (±0.0074)

AP = 0.5108 (±0.0185)

dinov2 Large

Seed	AUC	AP	
42	0.7675	0.4712	240 Npc
1	0.7669	0.4552	
2	0.7538	0.4495	
42	0.7482	0.4479	594 Npc
1	0.7544	0.4533	
2	0.7640	0.4549	

Baseline

AUC = 0.798 (± 0.0078)

AP = 0.0.471 (± 0.0612)

RETFound mAE

Baseline

AUC = 0.7655
AP = 0.4464

I want to know in what direction in the embedding space the different HyperE type embeddings are pushed as of right now.

I assume they are not "well separated". I want to see if adding a certain time based $\vec{v}$ to the embedding dim would help separate the examinations

! Interesting idea: See where all the different HyperE type embeddings reside at the beginning of an examination, see where they might join into each other and where they end up and detect the outliers to see why they could behave like that

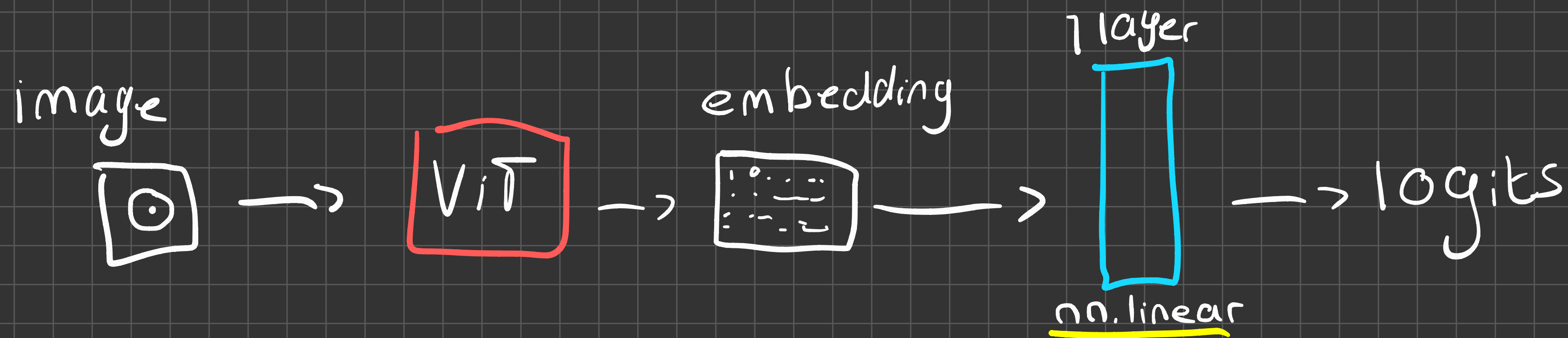
1. get the $\vec{\mu V}$ for every class, $\{\mu_1, \mu_2, \dots, \mu_5\}$. based on only frames in the first minute.
2. calculate the euclidean distance between all of the different class means.
3. See which frames differ the most from their class μ and inspect them visually.
4. find direction in which I could push the first minute frames into a certain direction, to Create a better distinguishment.

do the same for the other minutes
so it is visible where the most "separability" lies

! then calculate the class μ only based on training data and see if it is still possible to detect outliers.

-> this could be a useful way to decide which frames to use in an exam.

Linear probing



nn.linear(in_dim, num_classes) = $y = xW^T + b$

x = the embedding, RETFound Green has 384D

W = the weights, matrix in the form [num_classes, in_dim]

b = the bias, simple shift

y = raw scores (logits)

One VS. Rest

W is a vector (sort of arrow), during the training, the model learns in what direction this "arrow" should point, so it goes to where the most points of the class lay.

The calculation that nn.linear does, measures for a new image simply: to what degree does the the image embedding point allign with the **W** vector?

Multiclass (K=5)

W is now a collection of 5 different vectors, who'm all point to different directions based on their class.

The nn.linear layer measures for new image how well they fit to each of these 5 vectors

How to calculate how well the points fit to a vector direction?

Dot product

$$y = (x_1 \cdot w_1) + (x_2 \cdot w_2)$$

example

5D embedding

$x = [2, 1, -1, 0, 2]$

$W = [1, -1, 2, 0, 1]$

$b = 0.5$

$y = xW^T + b$ -> 1.5, because it is positive, the point points to the "right direction".

But, to be able to calculate metrics, we need a number between 0 and 1

With one vs. rest (binairy) we use the Sigmoid σ

$p = \frac{1}{1+e^{-y}}$, if $y = 1.5$ then $p = 0.818. \rightarrow 0.82\%$

With multi-class we use softmax

$$p_i = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}$$

- raw y score for that class
- raw y scores for all other classes summed up

Partial Derivatives

$p = 3b + 10t$

p = total price, bread = €3, taart = €10

derivative with respect to amount of bread.

"If I freeze amount of taarten I buy, how much does my total price change, if I add one extra bread (b)?

so t becomes a constant, since it is frozen.

most important rule:

everything that is not the letter that you're looking at, at that moment, gets treated as a constant.

$J = 3m^2 + 7K \rightarrow \frac{\partial J}{\partial m} = 6m$

$m = 1 \rightarrow 3$
$m = 2 \rightarrow 12$
$m = 3 \rightarrow 27$

examples

$$z = 4w + b \rightarrow \frac{\partial z}{\partial w} = 4$$

$$\rightarrow \frac{\partial z}{\partial b} = 1$$

$$z = 5w \rightarrow \frac{\partial z}{\partial w} = 5$$

$$c = (z-10)^2 \rightarrow \frac{\partial c}{\partial z} = 2(z-10)$$

so, $\frac{\partial c}{\partial w} = 2(z-10) \cdot 5$

$\downarrow$

$10(z-10)$

let w be 3

$z = 15$

$c = (15-10)^2 \rightarrow (15-10)(15-10) = 225 - 150 - 150 + 100 = \underline{25}$

$\frac{\partial c}{\partial w} = 10(15-10) = \underline{50}$ → so if $w\uparrow$, $c\uparrow$, so w needs to go down

$LR = 0.01$

$$w_{new} = \underbrace{w_{old} - (LR \cdot 50)}$$

$$3 - (0.01 \cdot 50) = 2.5$$

$w = 2.5$

$z = 12.5$

$c = 6.5$

$$\frac{\partial c}{\partial w} = 10\left(12\tfrac{1}{2} - 10\right) = \underline{25}$$

medisch netwerk (trained by hand)

bloedwaarde 1: $x_1 = 2$

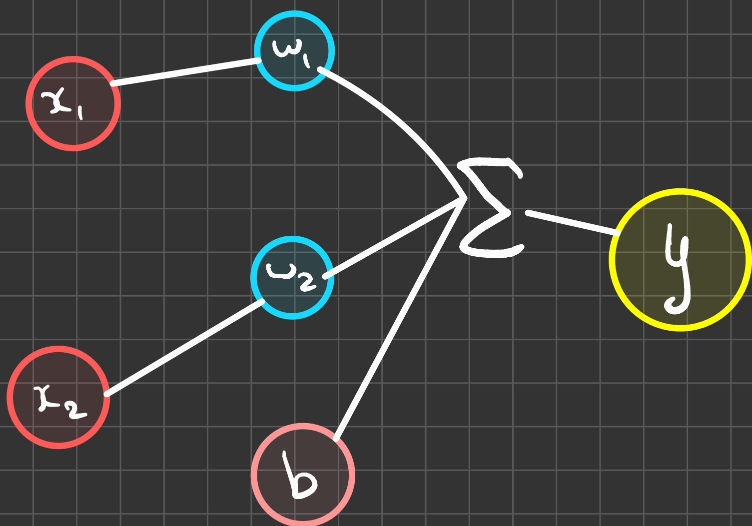
bloedwaarde 2: $x_2 = 3$

$y = 12$ (werkelijke gezondheidsscore.

$w_1 = 2$

$w_2 = 1$

$b = 1$



$$c = (z - y)^2$$

$$\frac{\partial c}{\partial z} = 2(z - y)$$

$$z = (x_1 \cdot w_1) + (x_2 \cdot w_2) + b$$

$$\frac{\partial z}{\partial w_1} = x_1$$

$$\frac{\partial z}{\partial w_2} = x_2$$

forward pass

$$z = (2 \cdot 2) + (3 \cdot 1) + 1 = \underline{8}$$

$$c = (8 - 12)^2 = \underline{16}$$

$$\frac{\partial c}{\partial w_1} = 2x_1(z - y) \rightarrow 4(8 - 12) = \underline{-16}$$

$$\frac{\partial c}{\partial w_2} = 6(8 - 12) = \underline{-24}$$

$$w_{1new} = w_{1oud} - (0.01 \cdot -16) = \underline{3.6}$$

$$w_{2new} = w_{2oud} - (0.01 \cdot -24) = \underline{3.4}$$

forward pass

$$z = 7\tfrac{1}{5} + 10\tfrac{1}{5} + 1 = 18\tfrac{2}{5}$$

$$c = (18\tfrac{2}{5} - 12)^2 = 38,44\ldots$$

$$h = x \cdot w_h$$
$$z = h \cdot w_z$$
$$c = (z - y)^2$$

$$\rightarrow \frac{\partial c}{\partial w_h} = \frac{\partial c}{\partial z} \frac{\partial z}{\partial h} \frac{\partial h}{\partial w_h}$$

$$\frac{\partial c}{\partial z} = 2(z-y)$$
$$\frac{\partial z}{\partial h} = w_z$$
$$\frac{\partial h}{w_h} = x$$

$$\rightarrow 2 w_z x (z-y) = \frac{\partial c}{\partial w_h}$$

$wh = 3$, $w_z = 2$, $y = 10$, $x = 2$

$\rightarrow c = 4$, $z = 12$

$\rightarrow 2 \cdot 2 \cdot 2 = 8(12-10) = 16$, so $w_h \downarrow$

Convolution

From Pixels to recognizing:

1. Convolution (the filter)
2. Pooling (zooming in)
3. Architecture (ResNet)

1. Convolution

Imagine we have a **scan** and we only want the **vertical lines** to be found.

For this, we use a small roster of 3x3 numbers, called a **kernel**. This roster moves across the entire scan and continuosly makes calculations.

the **calculation** is simple; we mutiply the numbers from the roster with the pixels that lay underneath it and sum them up.

If the pattern in the scan looks like the pattern in the filter, we get a high number.

-1	0	1
-1	0	1
-1	0	1

→ example of such a filter

0	255	255
0	255	255
0	255	255

→ pixels (piece of picture)

0 x -1 = 0　　0 x 255 = 0　　1 x 255 = 255
0 x -1 = 0　　0 x 255 = 0　　1 x 255 = 255
0 x -1 = 0 +　　0 x 255 = 0 +　　1 x 255 = 255 +
0　　0　　765

0 + 0 + 765 = 765 → Written on feature map.
Vertical boundary = found.

How to calculate this cost?

- the network takes all numbers from the last feature maps and squishes them flat to one long list of numbers.
- At the end, there are a few normal neurons, that say "based on these edges en shapes that I'm seeing, I'm giving a score of 0.9 to 'bleeding' and 0.1 for 'healthy'"
- If it was a bleeding,

$y = 1$, $C = (0.9 - 1)^2 = 0.01$

$z = 0.9$

- Then the network uses that 0.01 C to calculate back to the 3x3 filter

2. Pooling

- pooling is the summarizer of the network, the goal is to comprimize, while maintaining the most important information.
- most common pooling method is **Max pooling**
  - dividing the feature map in small blocks of i.e. 2 x 2 pixels
  - for each blocks we're looking at the 4 numbers
  - Then throwing away the 3 smallest numbers en only keep the biggest (**Max**) number
- ultimately results in smaller grids.

Rnn

- a standard neural network is "forgetful". Once an image has been processed, the network starts clean on the next image.
- the RNN does something different. It has a loop σ, the outcome of the calculation at step 1 is getting sent back to step 2.

How the math works

- let x_1, x_2, x_3 be a sequence of embeddings. Instead of only looking at x, the network keeps track of a Hidden State (h), this is the "memory"

- at every step, the network:

1. grabs the new input (x)
2. grabs the old memory (h_{old})
3. Throws them together in a formula to make h_{new}

$$h_{new} = \text{activation}(\underline{W} \cdot x + \underline{U} \cdot h_{old} + b)$$

$\underline{W}$ and $\underline{U}$ are the weights.

W determines how heavy the new information weighs.
U determines how heavy the "history" weighs.

example: heart rate

$$h_t = (W \cdot x_t) + (U \cdot h_{t-1}) + b$$

$x_1 = 80$
$x_2 = 100$
$h_0 = 0$
$W = 0.1$
$U = 0.5$
$b = 0$

first update is h_1

$$h_1 = (0.1 \cdot 80) + (0.5 \cdot 0) + 0 = 8$$
$$h_2 = (0.1 \cdot 100) + (0.5 \cdot 8) + 0 = 14$$

you can see that the old memory will keep on being multiplied by 0.5.
If there is a 20 frame sequence, this gradient vanishes.
This is where the GRU comes in, with its gates.

GRU

Gated Recurrent Unit

- a GRU has gates, that decide what information is important enough to keep, and what can go away.

→ this solves the vanishing gradient problem

1. The "gatekeepers": Reset (r) and update (z)

- In a GRU there are 2 controlnumbers (between 0-1), Which are calculated with the Sigmoid function σ.

- [ ] Update Gate (z_t): determines how much of the "old" memory is kept vs. how much new information is being added.
- [ ] Reset Gate (r_t): determines how much of the "old" memory w

2. the candidate-status $\tilde{h}_t$

- before updating the definitive memory, we make a concept version of the memory.
- here we use the **Reset Gate**. If r_t is null, we ignore the past entirely and we look at the new input only.

3. The finale mix h_t

- at the end, the old state and the new one are being mixed based on the update gate.

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$$

$$\tilde{h}_t = \tanh(\underbrace{W_h \cdot x_t}_{\text{the influence of the new input}} + U_h \cdot \underbrace{(r_t \cdot h_{t-1})}_{\text{the old memory, filtered by the reset gate}})$$

explain learning process

$W_1 = 0.4$

$W_2 = 0.6$

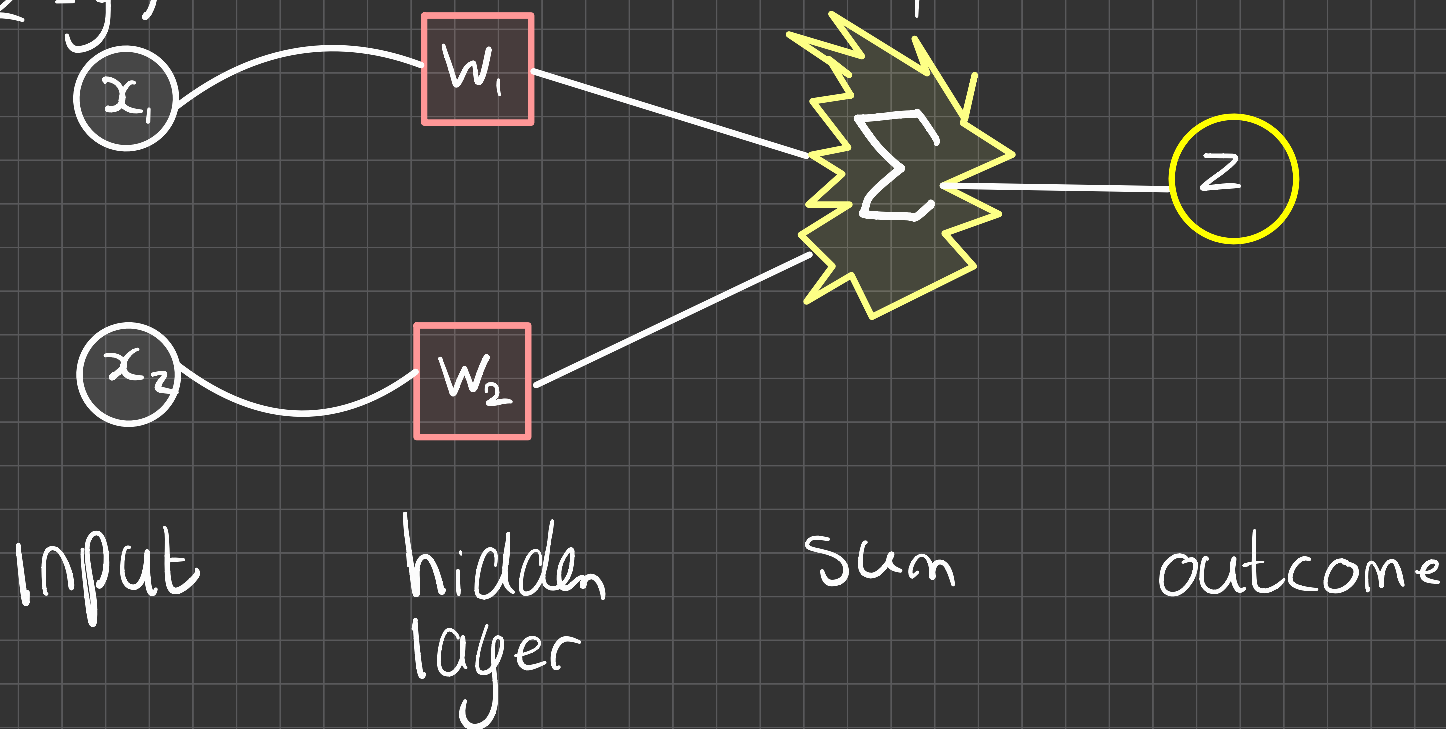
$x_1 = 8$ sleep

$x_2 = 9$ hours studied

→ $y = 7.5$

$C = (z - y)^2$

$z = (x_1 \cdot W_1) + (x_2 \cdot W_2)$



$$8 \cdot 0.4 = 3.2$$
$$9 \cdot 0.6 = 5.4$$
$$\sum \rightarrow \overset{z}{8.6}$$

$$c = (7.5 - 8.6)^2 = 1.21$$

$$\frac{\partial c}{\partial z} = 2(z-y)$$

$$\frac{\partial z}{\partial W_1} = x_1$$

$$\rightarrow 2x_1(z-y)$$

$$2 \cdot 8(8.6 - 7.5)$$

$$16(1.1) \rightarrow 17.6 \cdot 0.01$$

$$0.4 - (17.6 \cdot 0.01)$$

$$0.4 - 0.176 = W_{1\,new} = 0.224$$

Computer Vision (general)

- Works mostly by looking at the RGB pixel values.
- this approach doesn't work for identifying features larger than one pixel. So patches (parts of the picture) are needed.

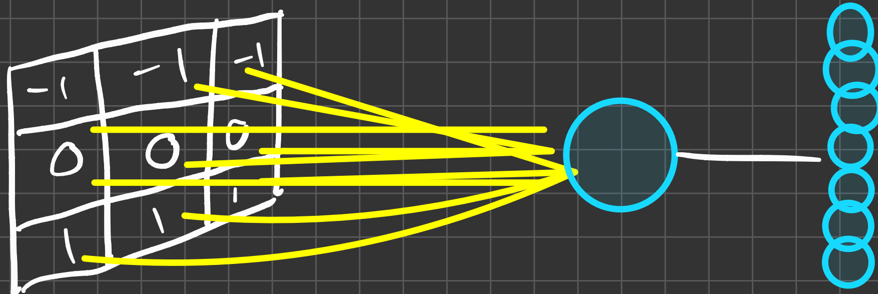
1	0	-1
1	0	-1
1	0	-1

→ Kernel that moves over a patch of pixels, When applying this kernel, we call it convolution.

↳ this one could detect vertical edges. prewitt operator is a kind of kernel

- different combined kernels can detect different features.

the convolution: the kernel weights are the input values.



neural networks change the kernel weights to optimize them for the task

Vision Transformer

necessary foundation:

- layer normalization
- self attention (Q, K, V)
- multi-head attention
- positional encoding
- Linear projection / flattened patches
- CLS token
- Attention is all you need paper

→ After these are understood, I can move on to ViT.

Layer Normalization.

Layer Norm is a technique to make the training of nn more stable and faster.

It is a way to keep the flow of numbers restrained within a model.

In the context of ViT's, this is happening on individual embedding level

example Imagine the picture is being divided into patches, and every patch is converted to a list with numbers (**vector**)

- Without normalization: Some numbers can be really big (500) and others small (0.01), makes it hard for the model to learn. **because** small changes in weights may have drastic effects.
- **With** layernorm: per patch, you're looking at all numbers in the vector, you calculate the mean and the variance 0 ± 1.0

difference

There is a big difference between Batch- and Layer normalization
(Batch → cnn)

1. <u>Batch</u>
   looks "vertically" through all the images for one specific feature, it is dependent on what else is in this batch.

2. <u>Layer</u>
   looks "horizontally" within one patch embedding, so the result is not dependent on what is in patch B or a different image

Why?

Layer Norm is ideal for Transformers, because the order and the content of the sequences of patches can vary per example. Without the model losing track by the rest of the batch.

Self-attention

description

self attention is the mechanism where a model learns which parts of an input are the most important in relation to each other.

Instead of looking at every piece of data (like a pixel or word) isolated, the model looks at the entire context.
It brings up the question: "Which other part of this input helps me to better understand this specific part better?

example

Imagine looking at an image of a bird sitting on a branch. To really understand the bird, the model should also pay attention to the "branch" it is sitting on, but less to the sky. Self-Attention calculates these relevant connections mathematically.

intuition

With transformers, We are using 3 roles **Query, Key, Value**

- **Query (Q)**

This is where you're Searching for. "I need information about flying birds"

- **Key (K)**

This is the label on the back of each book in the library.
The library scans all labels to See which ones fit to the question

- **Value (V)**

This is the actual content of the book, after you have decided which Keys are most relevant for the query, you use the information from those specific **Values** (books)

Linear Transformations

Let X be the "input matrix", consisting of a sequence of patch-embeddings.

to obtain the Q, K & V, we multiple X with 3 weight matrices: W^Q, W^K, W^V

$Q = X \cdot W^Q$

$K = X \cdot W^K$

$V = X \cdot W^V$

, these weight matrices are being trained by the model. So the model learns the best way to convert pixels in the best Q, K and V on its own.

The score : dot Product

Now that we have our matrices, we want to know how good a Query matches to a Key.

We do this by using the **dot product**. Mathematically, this is the center of "attention"

$$Score = Q \cdot K^T$$

In a matrix multiplication, the model compares every patch (Query) with **all** the other patches (keys).

If the vectors of a Query and a Key point in the same direction in the "space", the result in the result matrix becomes really big → high attention

Normalization

The scores from the dot-product can be really big or small. To make useable weights out of them, we use the soft max-function

The Result

For the final step, we mutipliy these weights with the Value matrix (V).

The Patches with the most attention deliver the most information to the final output of that layer

Positional Encoding

In a transformer, the final vector that is entering the attention layer is the sum of the content and position

$$Input = C + P$$

When we calculate the Query and the Key for two different patches (i and j), we do the following

- $Q_i = (C_i + P_i) W^Q$
- $K_j = (C_j + P_j) W^K$

When the model calculates the attention score via $Q_i \cdot K_j^T$, you get 4 interactions.

1. content-content : does patch i look like patch j ?
2. content-position : does the content of patch i fit with the position of patch j ?
3. position-content : does the pos. of patch i fit the content of patch j ?
4. position - position : do they exist in a logical distance

Full Architecture

Patches (196)

Vectors (196)

embeddings (196)

Frame

224x224

16x16x3

RGB

768 D Vector

W^E

embedding

Patching

flattening each patch

linear projection on W^E

add the positional encoding to each patch embedding

Results in weights

Self attention (single head)

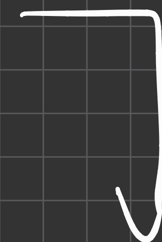
Patch	score
1	0.05
2	0.40
3	0.35
4	0.10
5	0.05
6	0.05

$$\frac{Q_i \cdot K^T_{other}}{\sqrt{d_k}} = \text{score}$$

embedding Position aware = PE + embedding

↳ This isn't just done once - it's done parallel 8 times (or more) → each with different learned projections.

each head has its own learned projection matrices W^Q, W^K, W^V



the results from all heads are concatenated and linearly transformed, enriching the presen-tation.

Applying Transformers in our project.

1.

Let T be our sequence of frames (e.g. 16 frames in exam)

Each frame went through RETFound Green, so is an embedding of dimension D (384)

Input matrix $X \in \mathbb{R}^{T \times D}$

2.

make positional encoding vector P

$X' = X + P$

3.

Temporal self attention $Q = X'W^Q$, $K = XW^k$, $V = XW^v$

Work in progress

embeddings through GRU

now that I understand how the embeddings are being made through the ViT, it is now time to look at the next step in the pipeline; GRU. How do we go from a sequence of embeddings to a classification.

1. Intuition : Why a GRU
2. Maths : dismantling de Gates
3. Data flow : Dimensions & Tensors
4. Finale : Classification Structure

1. Intuition

1. What is a sequence model?

not just 1 input, but a row:

$x_1, x_2, x_3, \dots x_T, \quad x_T \in \mathbb{R}^{384}$

a sequence model reads that embedding one for one in time order,

the goal is not only to look at each frame independently, but to internally keep a "summary" of what it has seen so far. We call this summary the hidden state h_t

So x_t is what you're seeing now and h_t is what the system has remembered upon till now

2. Hidden state normall Rnn

With a vanilla hidden state, the following is happening at every point in time:

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$$

The logic behind this is simple:

the new state h_t depends on 2 things

1. new input x_t
2. previous state h_{t-1}

So the model is saying: "give me what I see now plus what I already knew, and I'll make a new summary.

3. What do the symbols mean?

let Hidden state size $H = 128$
then the dimensions are:

$x_t \in \mathbb{R}^{384}$

$h_{t-1} \in \mathbb{R}^{128}$

$W_x \in \mathbb{R}^{128 \times 384}$

$W_h \in \mathbb{R}^{128 \times 128}$

$b \in \mathbb{R}^{128}$

$h_t \in \mathbb{R}^{128}$

$1 \times 384 \quad 384 \times 128$

$1 \times 5 \quad 5 \times 2$

But, this is kind of dumb, since it is randomly picked what should be forgotten

1. What is the GRU solving?

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$$\tilde{h}_t = \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

The GRU wants to decide:

- how much of the history stays relevant
- how much the new information may change the memory
- if part of the old memory should be shut of to make a new candidate state.

2. Update Gate z_t

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

Sigmoid(σ) turns values between 0 and 1

$z_t \in \mathbb{R}^H$ $H=128$

close to 0 : maintain the old memory
close to 1 : replace strongly with new information.

it works per dimension.

the update gate determines how much of the new x_t may enter

3. reset Gate r_t

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$r_t \in \mathbb{R}^H$

this gate is used for building the candidate state $\tilde{h}_t$

close to 0: ignore pieces of the old memory when building the new candidate

close to 1: mainly use the old memory

It says: "how much of the old memory may participate while I'm thinking of what the new candidate should be?"

4. Candidate hidden state $\tilde{h}_t$

$$\tilde{h}_t = \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h)$$

here originates the proposal for a new hidden state

symbol $\odot$ means element-wise multiplication.

so $r_t \odot h_{t-1}$ means that the research gate will mute or almost turn off some components in the old memory before they end up in the candidate update.

so $\tilde{h}_t$ says = "what would be a good new state, given the current input en the allowed parts of the past?"

5. final hidden state h_t

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

7. Code-Wise

per frame $x_t \in \mathbb{R}^{384}$

but GRU gets a sequence $x_1, x_2, \dots, x_T$

So $X \in \mathbb{R}^{T \times 384}$

When working in batches $X \in \mathbb{R}^{B \times T \times 384}$

$B = 8$, $T = 12$ → $[8, 12, 384]$

→ The GRU reads the tensor alongside the time dimension
at every new frame, new hidden state $h_1, h_2, h_3, \dots, h_T$

LoRA (Low Rank Adaptation)

As medical imaging models scale toward Alot of parameters, Full fine tuning all model weights for specific clinical tasks has become computationally and financially prohibitive.

LoRA offers a paradigm shift for the image embeddings, reducing trainable parameters by up to 10.000 times.

Unlike previous "adapter" methods, LoRA introduces zero inference latency, as the adaption matrices can be merged with the original Weights during deployment.

Limitations traditional FT

- storing really large fine tuned models
- Updating weights that won't benefit

The LoRA Methodology

LoRA is based on the hypothesis that the change in weights during task adaptation has a low "intrinsic rank".

Instead of updating the full-rank pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times K}$, LoRA represents the update ΔW through a low-rank decomposition:

$$W = W_0 + \Delta W = W_0 + BA$$

Where:

W_0 is the frozen pre trained weight matrix

$B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ are trainable parameters.

r is the rank ($r \ll \min(d,k)$)

d and k are the dimensions of the originally trained weight matrix W_0

r stands for the rank, this is a hyperparameter that you chose yourself and is substancially lower than the original dimension

example