

VISION TO LANGUAGE · VISION I

Vision Transformer · optional labs and references

Detailed arithmetic, PyTorch implementation, transfer learning and interpretation.

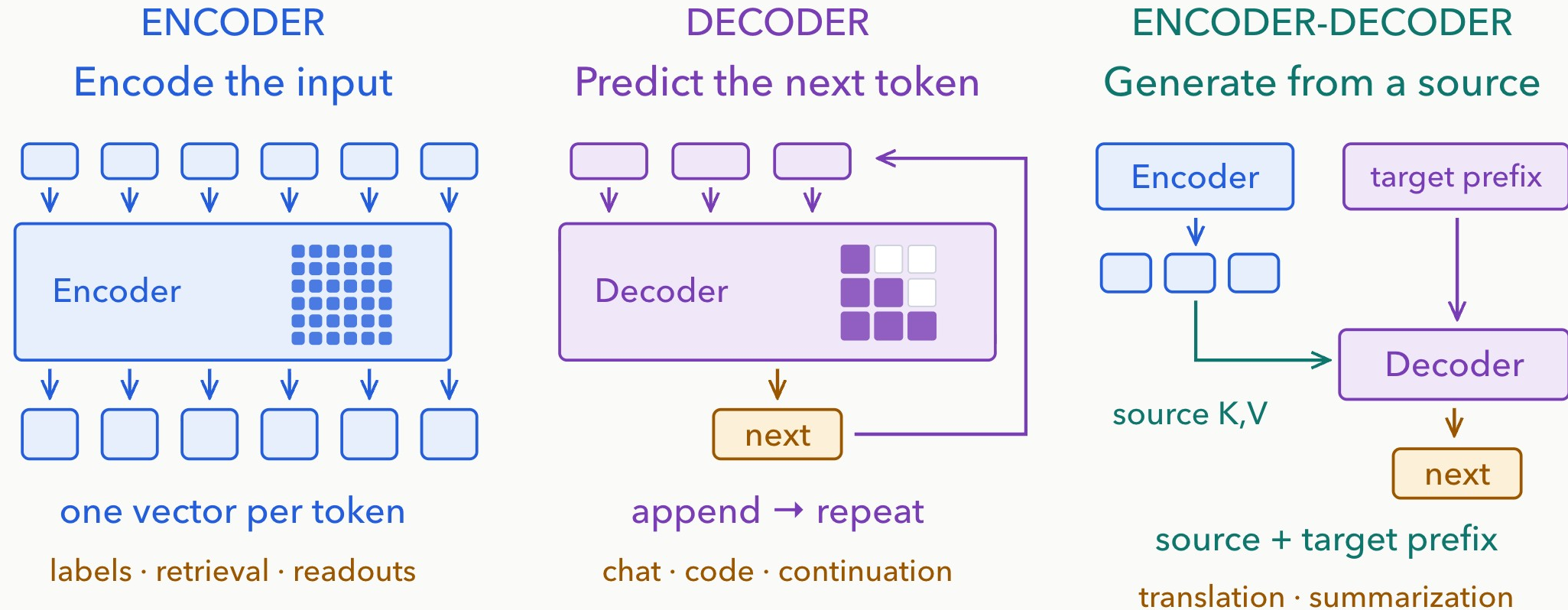
Optional labs, worked examples and reference diagrams

OPTIONAL REFERENCE DECK

Open the boxes when you need the details.

The main 54-slide lecture is complete without this material.

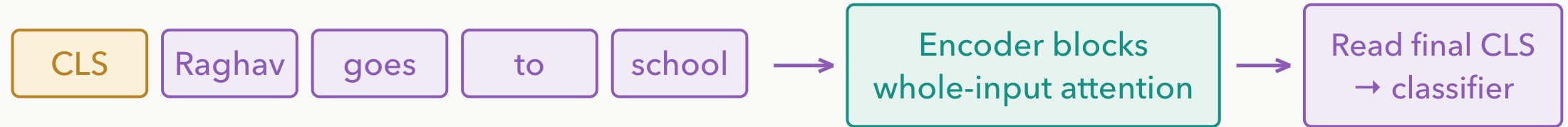
We already know how an encoder builds context



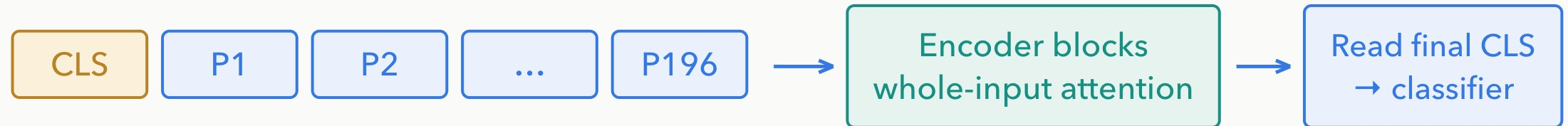
Today we reuse the encoder on the left: image patches become tokens, and the final image representation predicts a label.

A ViT is an encoder over image patches

TEXT



IMAGE



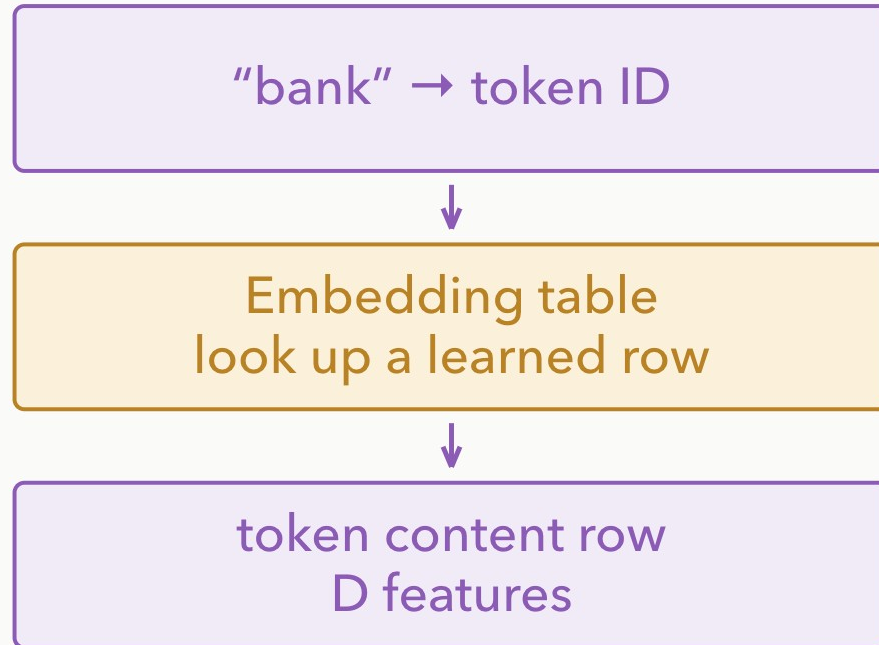
Same encoder idea. Different tokens.

Blue: vision states Purple: language states Amber: learned CLS / position

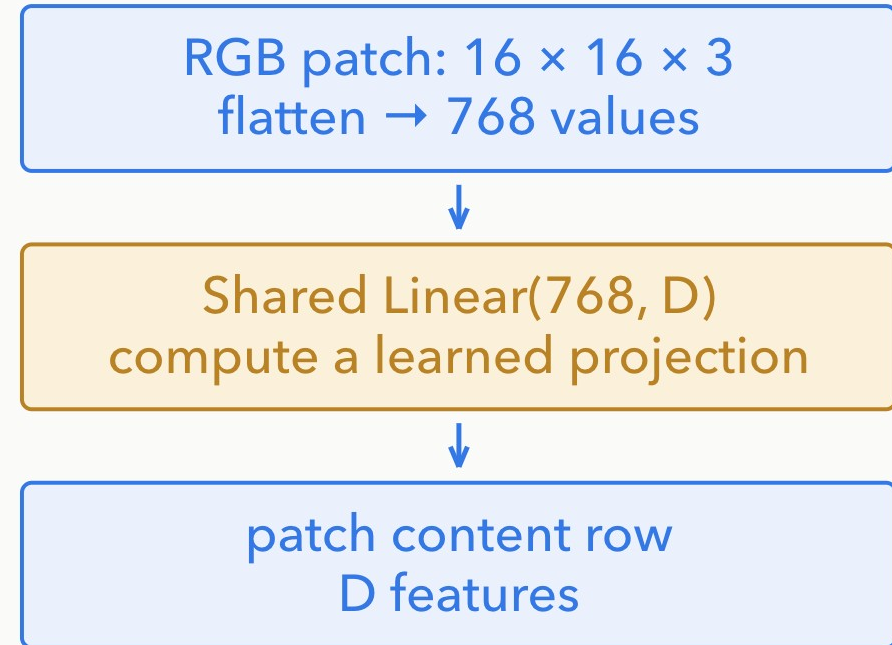
Attention builds context in either sequence. The input representation and the task head determine how we use the encoder.

Text looks up a row; an image patch computes one

TEXT



IMAGE



Add learned position → send the rows into the encoder

Both routes produce a row of D features. Text learns a vocabulary table; vision learns a shared pixel projection. Our image checkpoint chooses $D = 192$.

SECTION

01

The image classification task

A text encoder built context from supplied tokens.

What should a model predict from a photograph?

Labeled photos

→ The image task

→ Useful clues

Start with the dataset, choose an image label, then ask which parts of the photograph help us decide.

What does our animal dataset look like?



dog

Newfoundland



dog

Pug



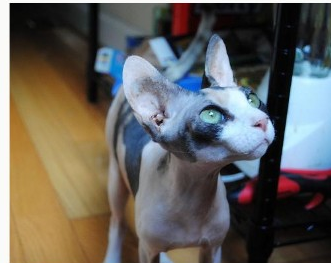
dog

Great Pyrenees



cat

Persian



cat

Sphynx



cat

Birman

Oxford-IIIT Pet: six examples. Each photo has a species label and a breed label.

What animal do you see?

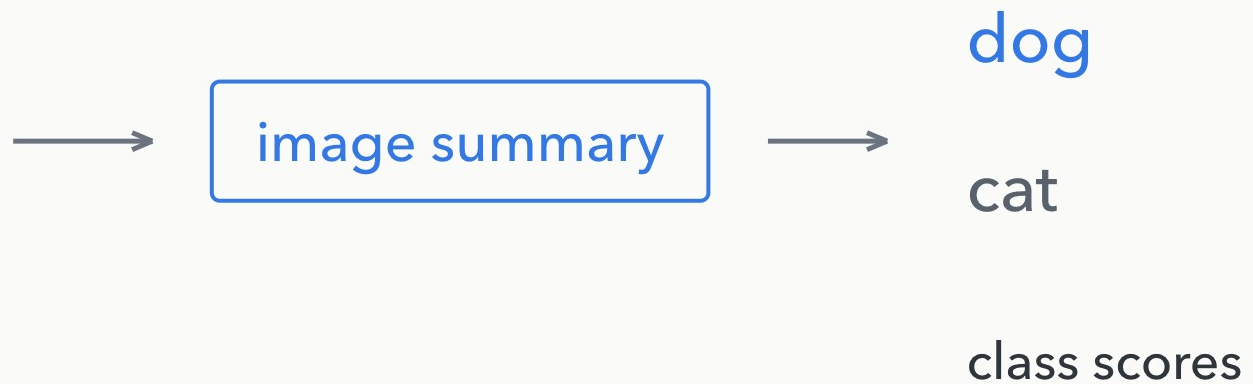


One photograph.
One image label.

What clues did you use?

Which parts of the photograph helped you decide?

Our task today: classify the whole image



For the rest of this lecture, we observe the whole photograph and predict one class label.

Attention: what can the face tell this patch?

source: face



→ face information

send clues

receiver: dark patch



fur, shadow, background?

↓
patch numbers

Recall Q, K, V from text

Q (receiver): what to look for

K (source): what can match

V (source): information to send

Next: the image versions, then a numerical example.

LOCAL ROLES

Q / receiver

K / source

V / message

Queries and keys determine weights; values supply the information to combine. The dark patch is our receiver, and the face patch is one possible source.

Should every source contribute equally?

source: face



face information

Illustrative shares

receiver: dark patch



source: branches



branch information

larger share

fur, shadow, background?

patch numbers

smaller share

LOCAL ROLES

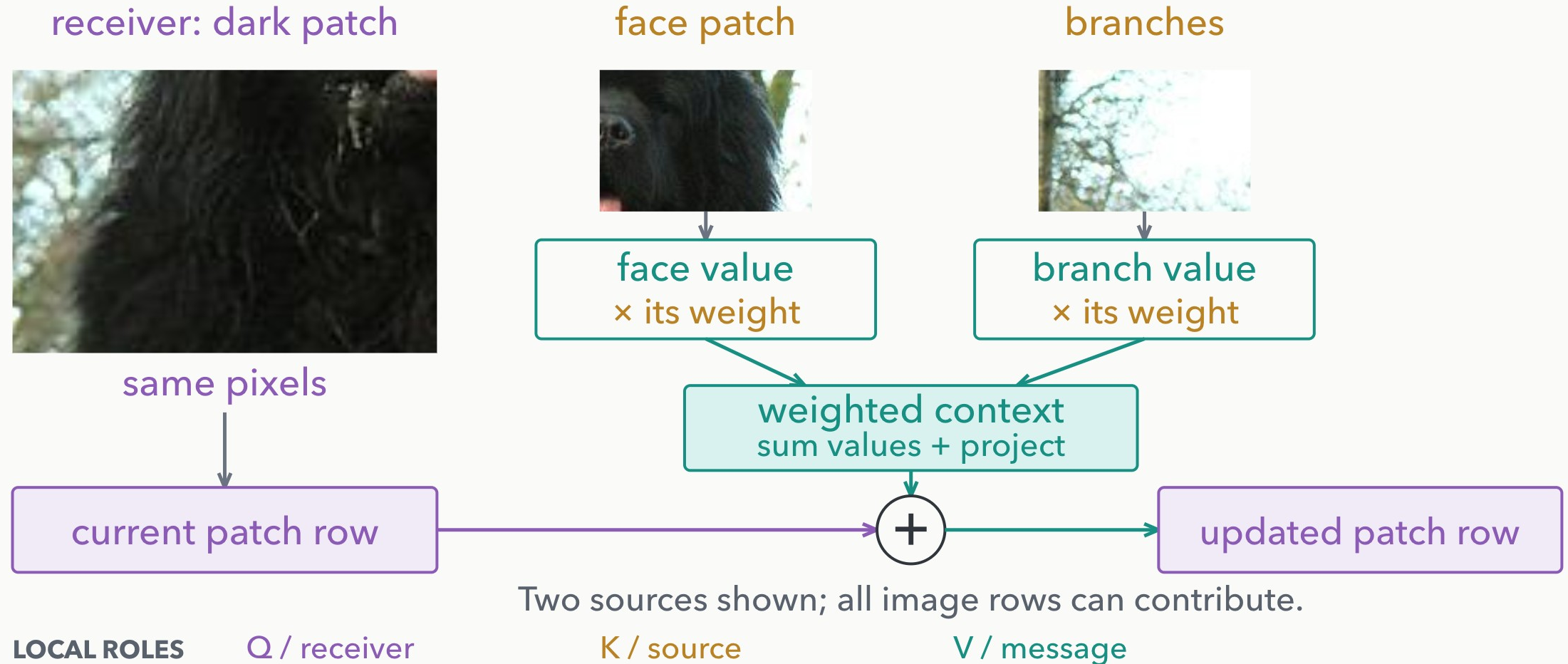
Q / receiver

K / source

V / message

Here, face clues could help more than branches. Attention weights control how much each source contributes to the message.

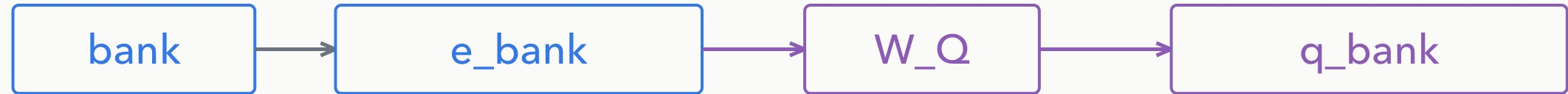
What changes when the patch gets context?



The face and branches contribute value vectors. Attention weights mix those vectors, and an output projection forms the message added to the dark patch's row. The pixels stay fixed.

What could a query be in the image?

Text · receiver



Which context could help bank?

Image · receiver



Could this dark texture belong to the animal?

LOCAL ROLES

Q / receiver

K / source

V / message

The row being updated makes a query. Its learned projection determines what kinds of source information it can match.

What could a key be in the image?

Text · source

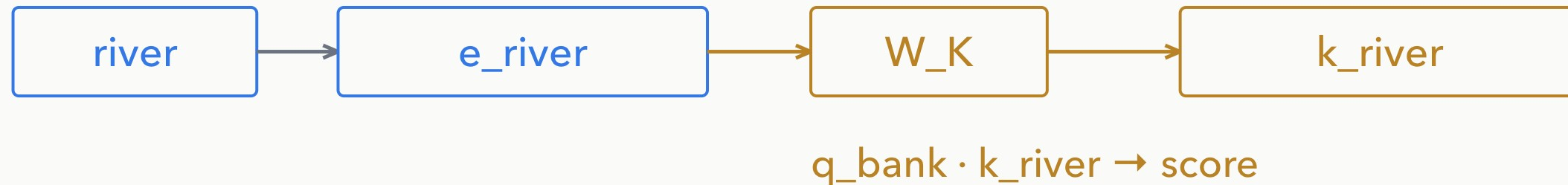
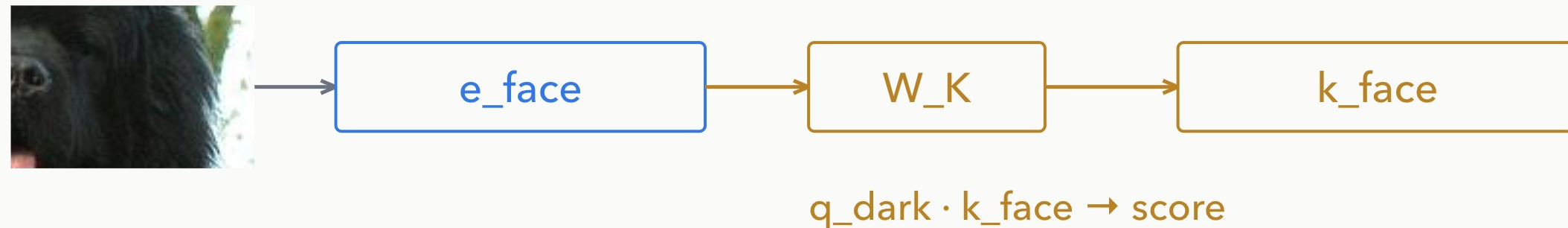


Image · source



LOCAL ROLES

Q / receiver

K / source

V / message

Each source row makes a key. Comparing the receiver's query with source keys gives the scores used to choose attention weights.

What information would a value send?

Text · source

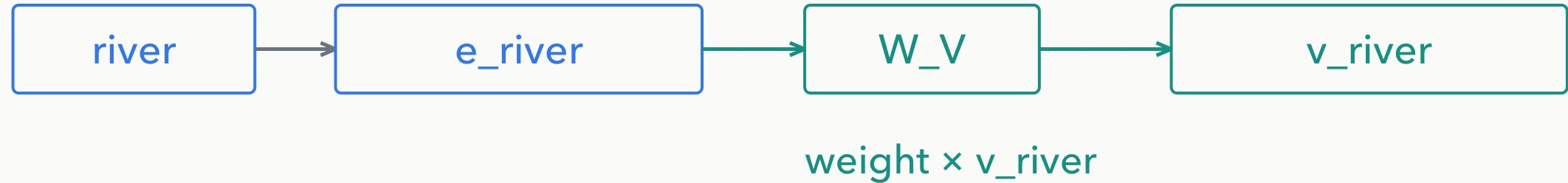
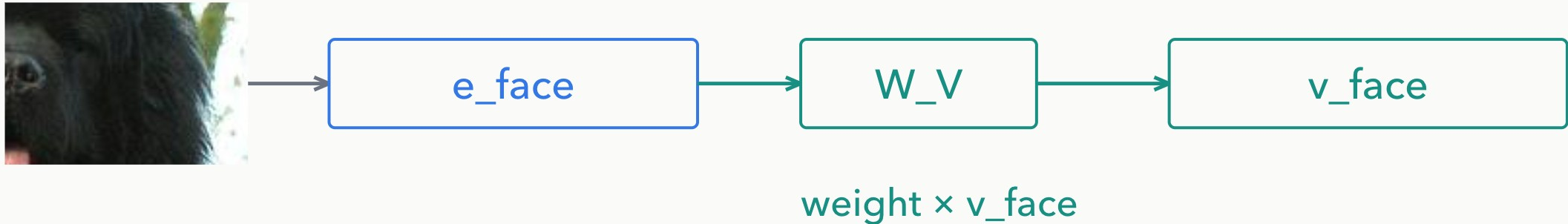


Image · source



LOCAL ROLES

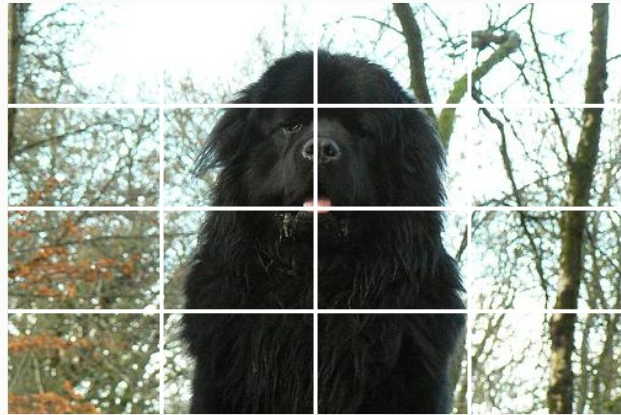
Q / receiver

K / source

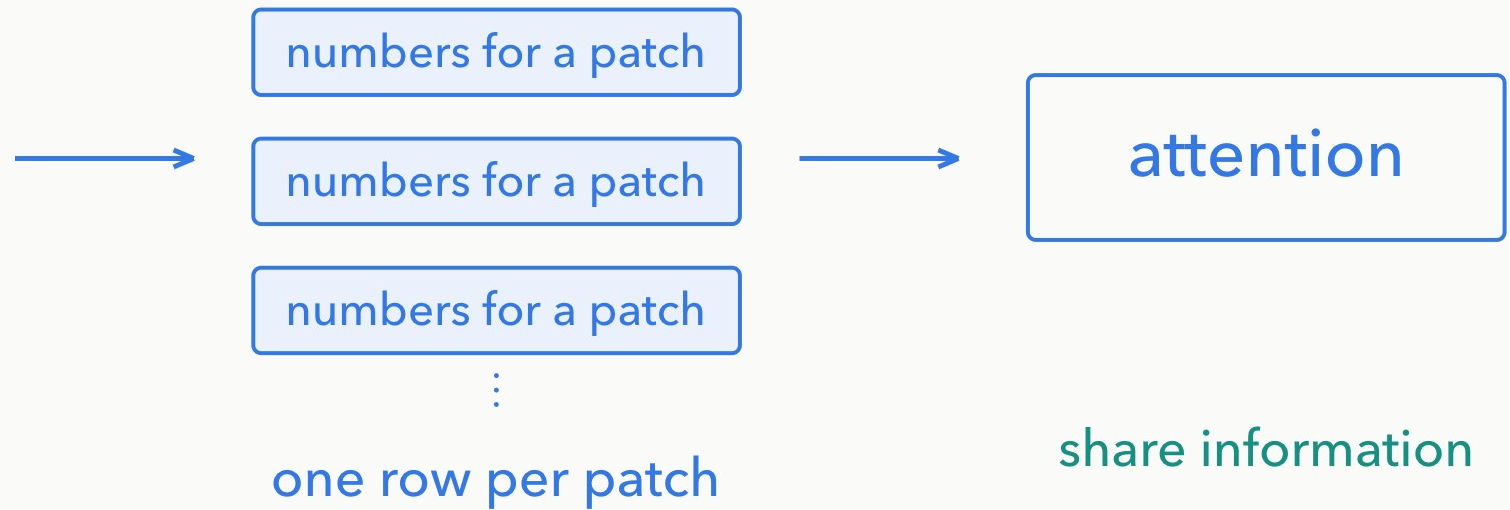
V / message

Q and K choose the weights. V supplies the information to combine using those weights; the combined message updates the receiving representation.

How can we give this photograph to attention?

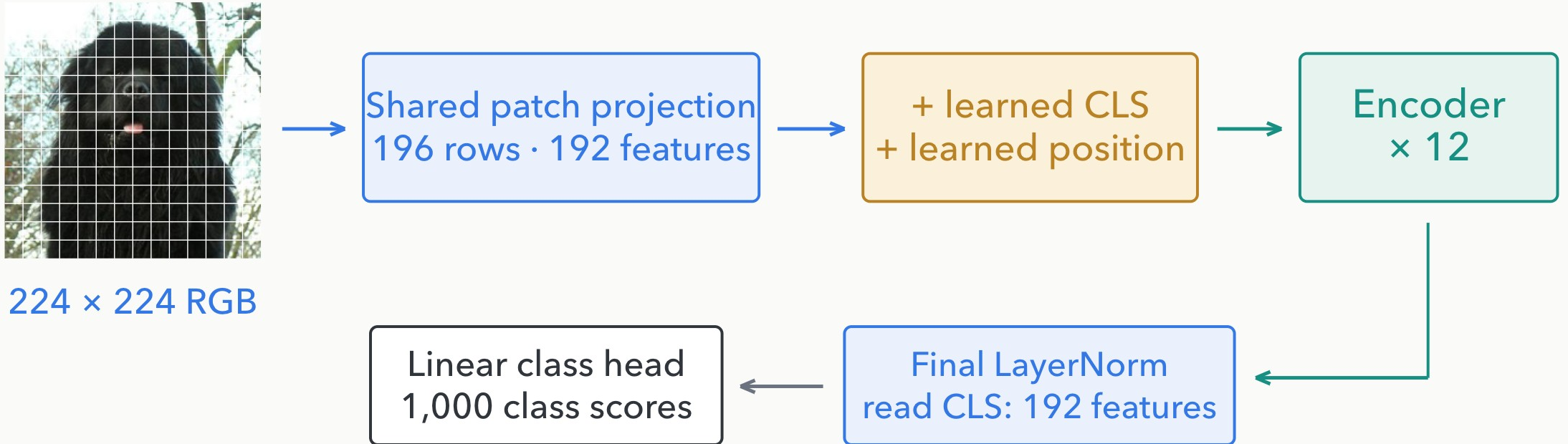


whole photograph



Attention works on rows of numbers. Next, we choose image patches and turn their pixels into those rows.

The image classifier, drawn as one encoder pipeline



ViT: Dosovitskiy et al., 2020 · dimensions shown for our ViT-Tiny checkpoint

Patches become feature rows. Learned CLS and position prepare the input. Encoder blocks build context; the final CLS feeds the classifier.

SECTION

02

From pixels to patch embeddings

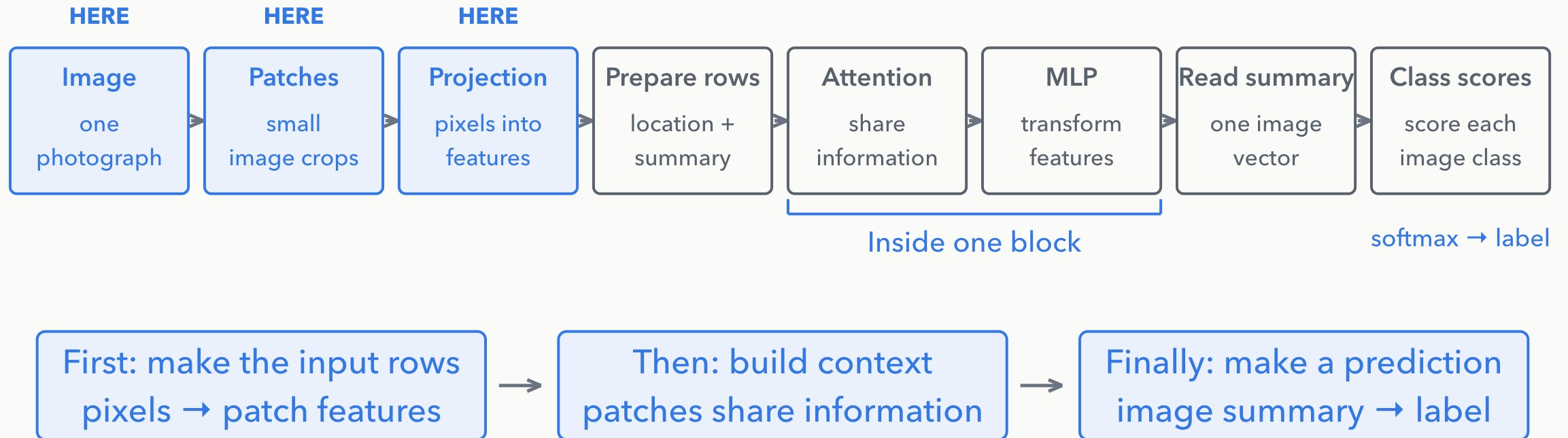
Attention updates a row of numbers for each token.

How do we make those rows from an image?

RGB pixels → Shared linear layer → Patch embeddings

Read one small patch, calculate its embedding, then scale the same operation to a real photograph.

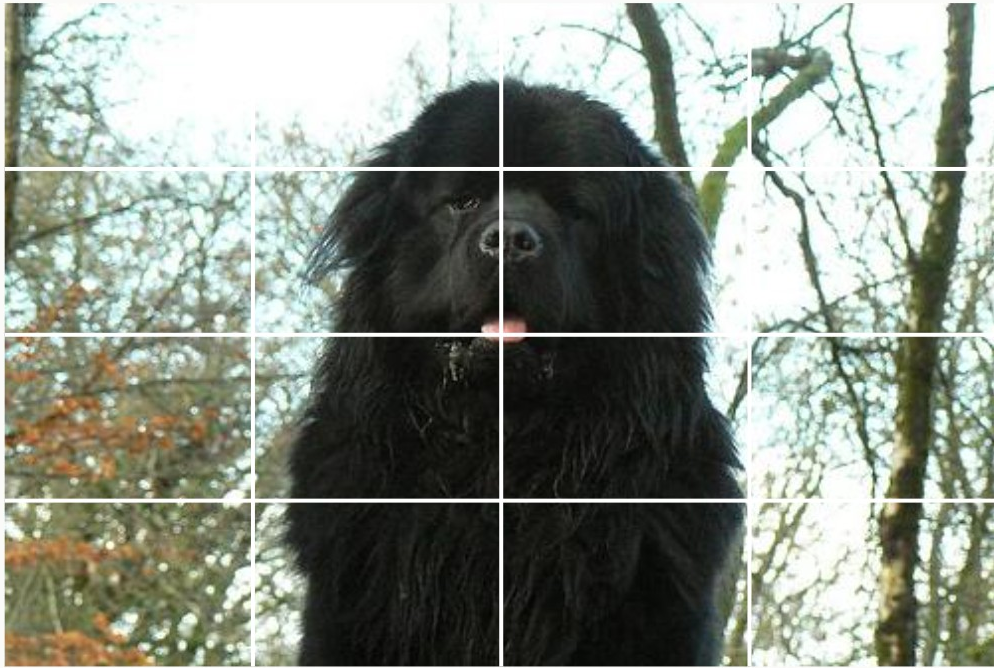
The whole route: photograph to prediction



The photograph stays fixed. The rows of features change as we follow the arrows.

First turn pixels into patch features. Prepare those rows, let them exchange information, then read one image summary to score the classes. We will open each box as we reach it.

Where do the patch boundaries go?



P6



P7



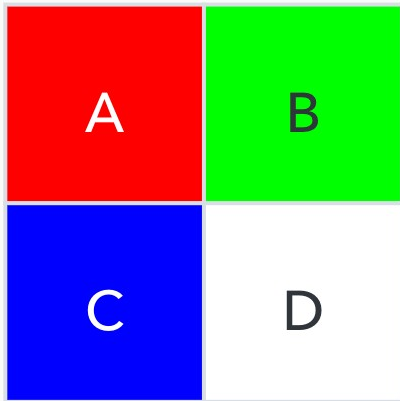
P10



P11

The grid cuts through the photograph before the model knows where the dog is.

Read the RGB values of each pixel



$2 \times 2 \times 3$

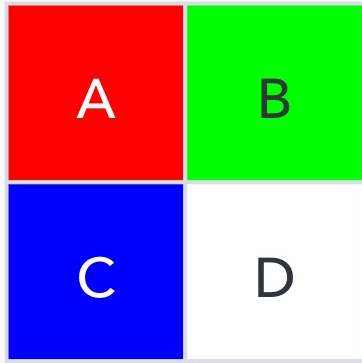
pixel	R	G	B
A	1	0	0
B	0	1	0
C	0	0	1
D	1	1	1

$4 \text{ pixels} \times 3 \text{ channels} = 12 \text{ values}$

For this calculation, divide RGB values by 255. Read A, B, C, D: top-left, top-right, bottom-left, bottom-right.

Flatten one channel at a time: R, then G, then B

4 pixels $\times$ 3 channels = 12 values



one RGB patch

R values				G values				B values			
A.R	B.R	C.R	D.R	A.G	B.G	C.G	D.G	A.B	B.B	C.B	D.B
1	0	0	1	0	1	0	1	0	0	1	1
1	2	3	4	5	6	7	8	9	10	11	12

A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D inside each channel

One row x_1 : inputs and weights use this same order.

We use all R values, then G, then B: 12 values for this patch. Pixel-by-pixel RGB also works if the weight columns follow that order. Our PyTorch code uses the channel-first order.

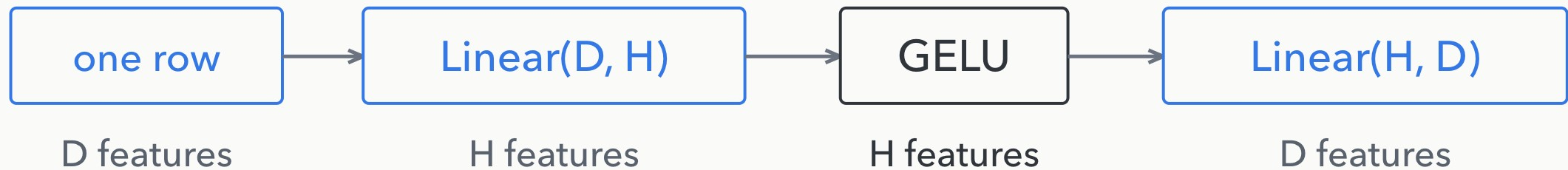
Do we apply an activation after the patch layer?

Patch embedding · at the image input



$c_1 = x_1 W + b$. Bias makes this affine; no ReLU or GELU follows.

Later, after attention · the block MLP

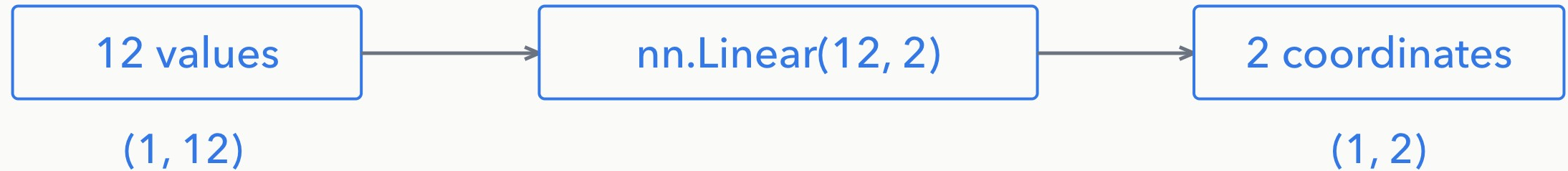


Our patch embedding uses one affine layer. The later block MLP puts GELU between two linear layers. D is the embedding width; H is the MLP's hidden width.

What does “projection” mean here?

Input: x_1

Output: c_1

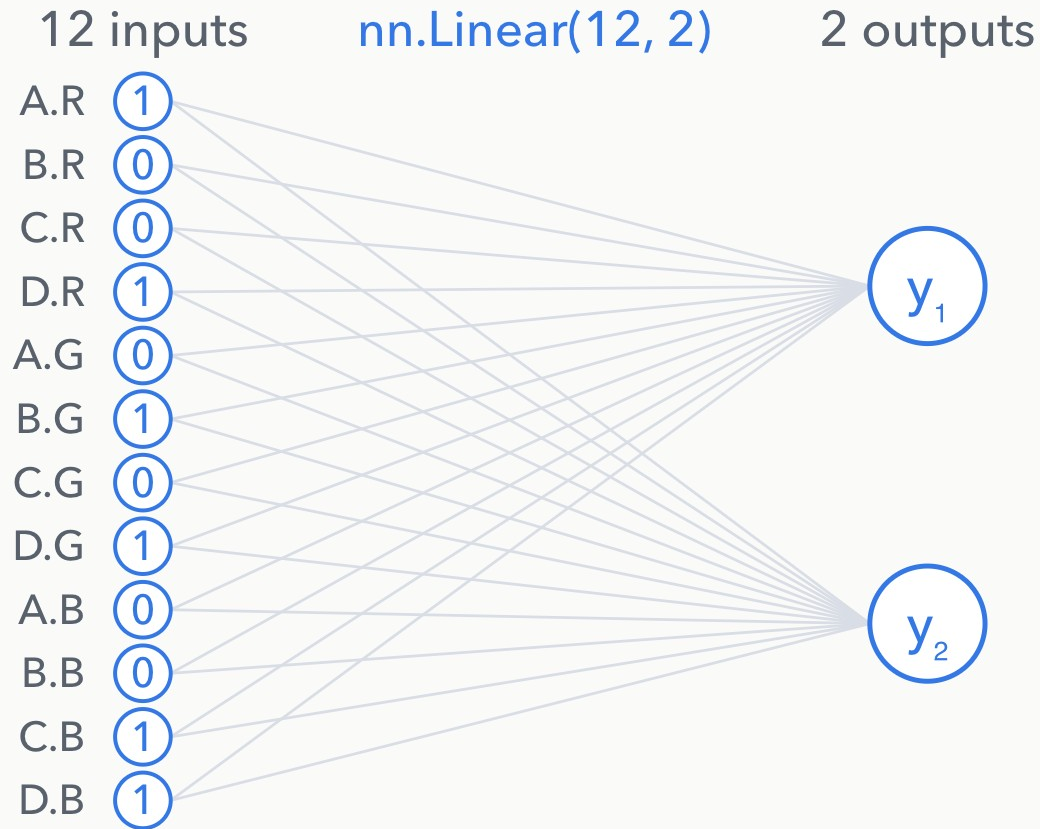


$$c_1 = x_1 W + b$$

$$(1, 2) = (1, 12) \times (12, 2) + (1, 2)$$

A projection forms weighted sums of the pixel values and adds a bias. This one linear layer turns 12 inputs into a 2-coordinate patch embedding.

12 input numbers, 2 output numbers



One patch in.
Two features out.

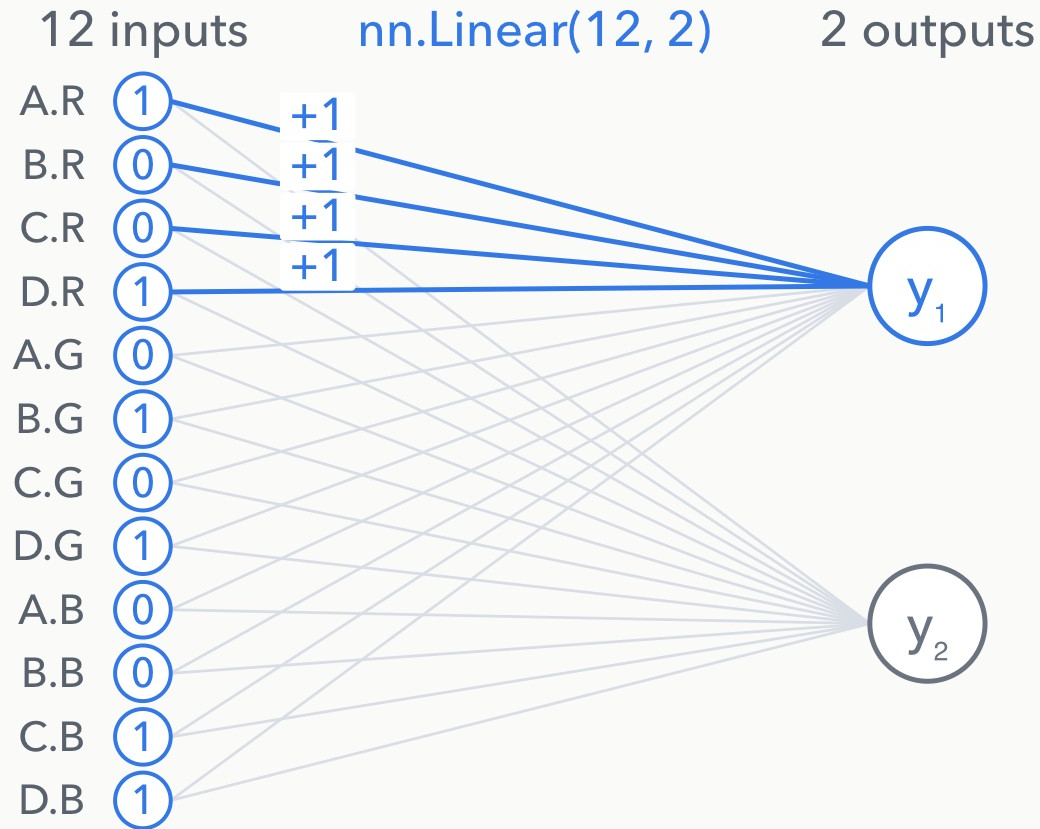
Each line has a weight.
Each output adds a bias.

$$\mathbf{c}_1 = [y_1, y_2]$$

24 weights + 2 biases

Each input node holds one RGB value. Both outputs read all 12 inputs. Together, y_1 and y_2 form the embedding for this one patch.

Follow the connections into output 1



Output 1

$$\begin{aligned} &A.R + B.R \\ &+ C.R + D.R \end{aligned}$$

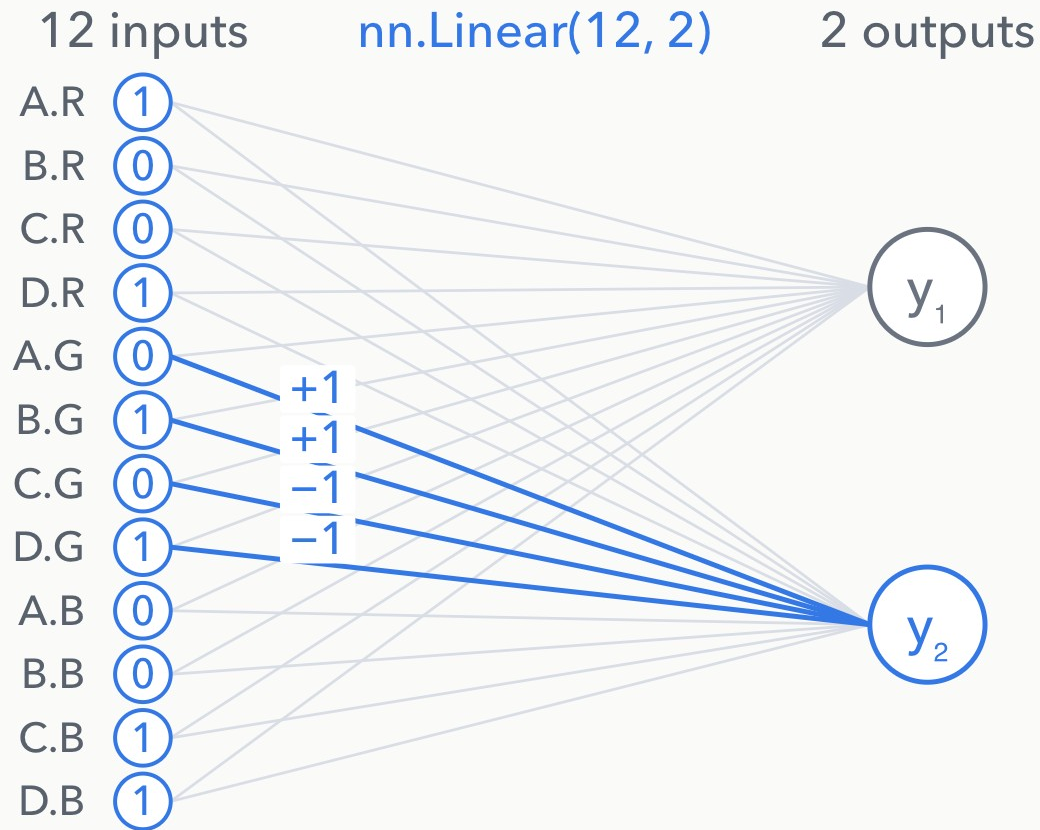
Other incoming weights: 0

$$\begin{aligned} &1 + 0 + 0 + 1 \\ &+ 0.5 \text{ (bias)} \end{aligned}$$

$$= 2.5$$

Multiply each input by its connection weight, add the contributions, then add the bias. The highlighted connections show the nonzero weights for this output.

Now follow the connections into output 2



Output 2

$$\begin{aligned} &A.G + B.G \\ &- C.G - D.G \end{aligned}$$

Other incoming weights: 0

$$\begin{aligned} &0 + 1 - 0 - 1 \\ &- 0.5 \text{ (bias)} \end{aligned}$$

$$= -0.5$$

Multiply each input by its connection weight, add the contributions, then add the bias. The highlighted connections show the nonzero weights for this output.

These two numbers are the patch embedding



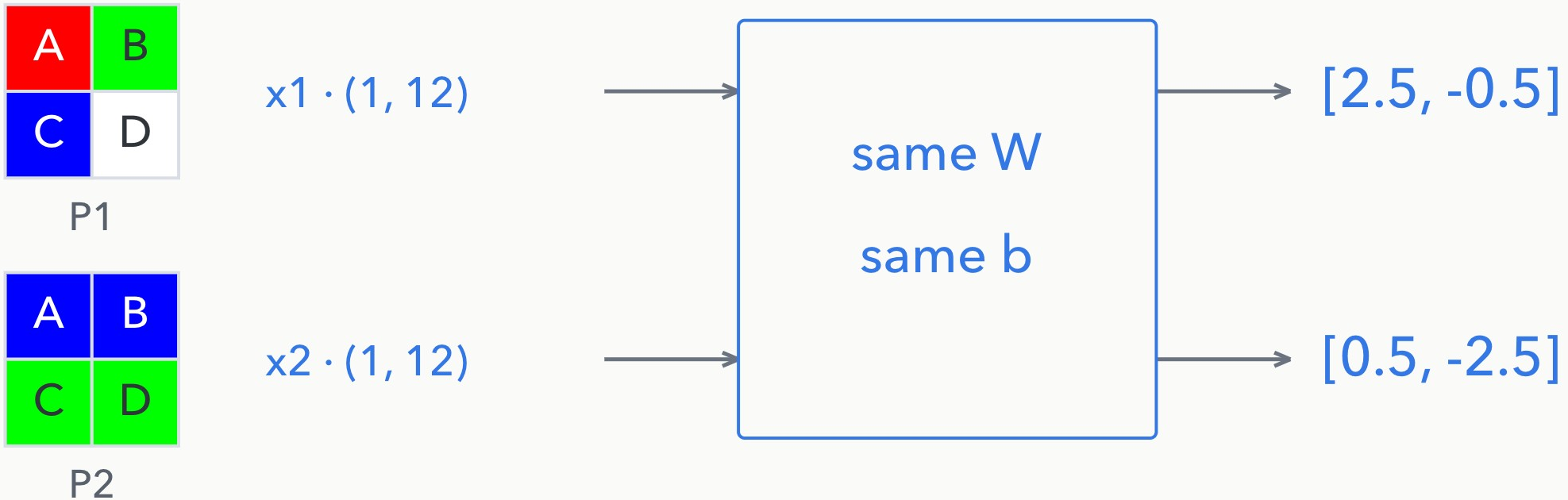
No activation: -0.5 stays -0.5 .

Later block MLP: Linear $\rightarrow$ GELU $\rightarrow$ Linear

c_1 represents the content of patch 1. Its two coordinates are features for the model; the image classifier comes later.

Apply the very same layer to another patch

Create once: `proj = nn.Linear(12, 2)`

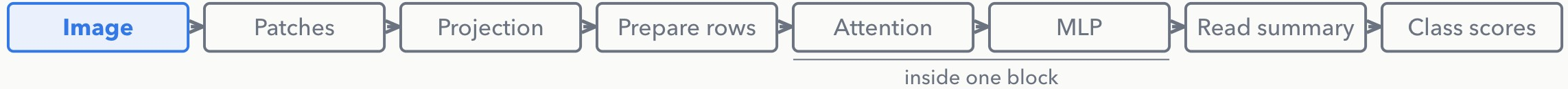


$C = \text{proj}(X): (2, 12) \rightarrow (2, 2)$

The pixel rows differ. Both use the same 26 parameters, so we compute both embeddings with one call: $C = \text{proj}(X)$.

Start with the same dog photograph

Photo walkthrough · Step 1 of 12



The same dog photograph

Input image

$224 \times 224 \times 3$

224 rows of pixels

224 columns of pixels

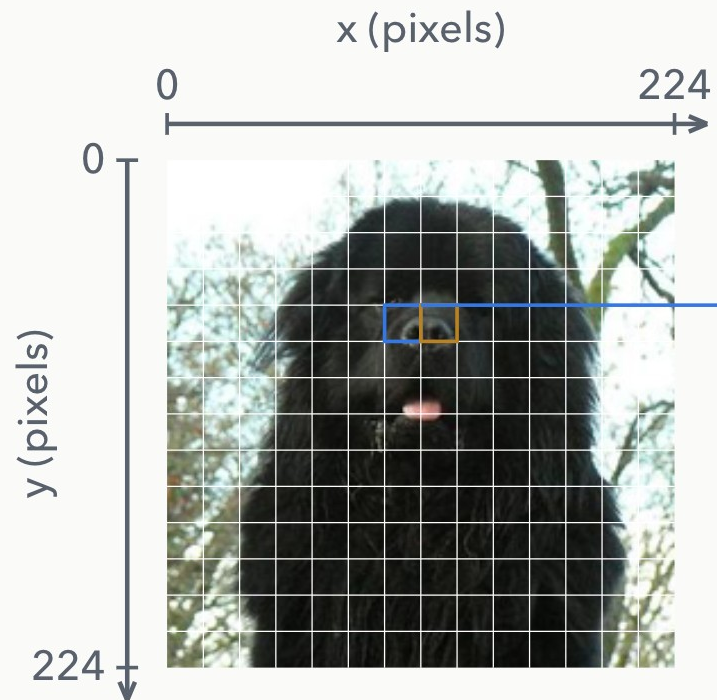
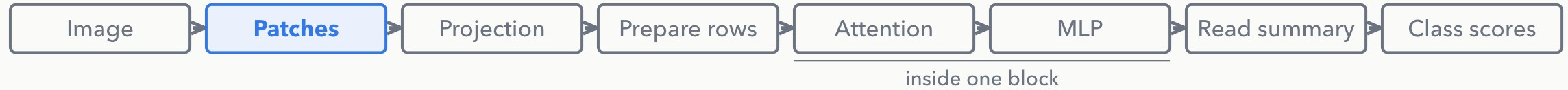
3 values per pixel: R, G, B

Next: cut this image into equal-sized patches.

We now carry this one image through the real model's input pipeline. It has already been resized and cropped to 224×224 RGB pixels.

Split the image into 16×16 patches

Photo walkthrough · Step 2 of 12



One patch

16 pixels wide

16 pixels high

3 RGB channels

P63 · enlarged

Next: number the pieces, one image row at a time.

The x-axis runs right; the y-axis runs down. Both span 224 pixels. Cut every 16 pixels along each axis. Each piece keeps its RGB values; we will count the pieces next.

Number the patches row by row

Photo walkthrough · Step 3 of 12



Use the image axes:

columns: $224 \div 16 = 14$

rows: $224 \div 16 = 14$

total = columns $\times$ rows

$14 \times 14 = 196$ patches

shape: $196 \times 16 \times 16 \times 3$

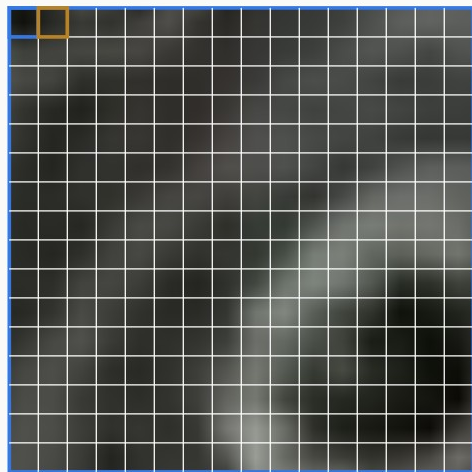
Number left to right, then continue on the next row. Use the image and patch sizes to calculate the row length and total before revealing the answers.

Read the RGB values inside patch 63

Photo walkthrough · Step 4 of 12



P63 · enlarged



$16 \times 16 = 256$ pixels

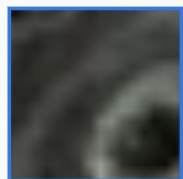
Pixel in P63	R	G	B
first	16	17	12
second	41	42	37

256 pixels $\times$ 3 RGB values
= 768 numbers in this patch

P63 contains 256 pixels. Its first pixel is RGB [16, 17, 12]; its next pixel is [41, 42, 37]. Three values per pixel give 768 numbers in this same patch.

Normalize those same RGB values

Photo walkthrough · Step 5 of 12



Same P63 pixels; apply the checkpoint's normalization.

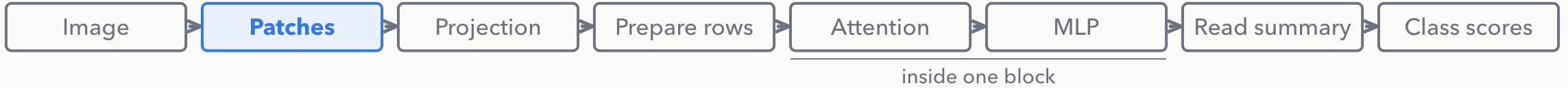
Pixel	8-bit RGB		Normalized RGB
1	[16, 17, 12]	—————>	[−0.875, −0.867, −0.906]
2	[41, 42, 37]	—————>	[−0.678, −0.671, −0.710]

Each channel: $(\text{value} / 255 - 0.5) / 0.5$

The checkpoint rescales every channel using the same formula. For the first red value, $(16 / 255 - 0.5) / 0.5 \approx -0.875$. P63 still contains 768 values.

Flatten patch 63 in the same order as the code

Photo walkthrough · Step 6 of 12



All R values → all G values → all B values



P63

R (256) $[-0.875, -0.678, -0.561, \dots]$

G (256) $[-0.867, -0.671, -0.553, \dots]$

B (256) $[-0.906, -0.710, -0.592, \dots]$

$x_{63}: 1 \times 768$

Within each channel: left to right, top to bottom.

Concatenate 256 red values, 256 green values and 256 blue values. This gives one 768-number patch row. F.unfold and the reshaped Conv2d weights use this exact ordering.

Pass that row through the shared linear layer

Photo walkthrough · Step 7 of 12



P63 is now x_{63} : a row of 768 normalized pixel values.



Each output = a weighted sum of the 768 inputs + its bias. No activation.

This layer reads 768 values and computes 192 output features. Its learned weights and biases are shared by every patch. One patch row enters; one embedding row leaves.

Read the 192 output features for patch 63

Photo walkthrough · Step 8 of 12



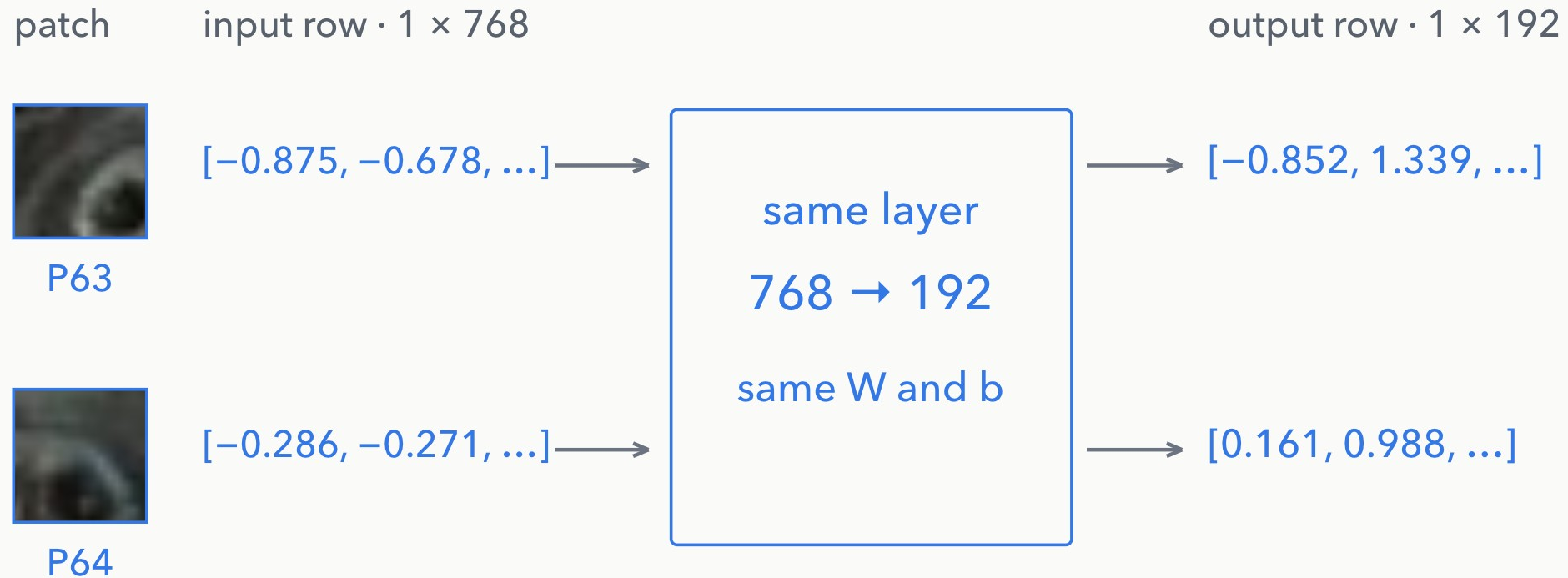
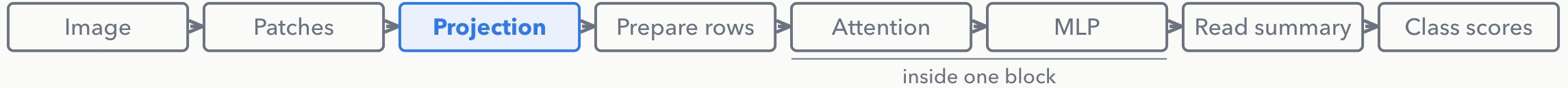
feature 1	feature 2	feature 3	...	feature 192
-0.852	1.339	0.504	...	-1.199

$c_{63} = [-0.852, 1.339, 0.504, \dots, -1.199]$ shape: 1×192

The layer produces c_{63} , the content embedding for P63. These are actual outputs from the pretrained model, rounded here. The first feature is -0.852 and the last is -1.199 .

Pass patch 64 through the very same layer

Photo walkthrough · Step 9 of 12

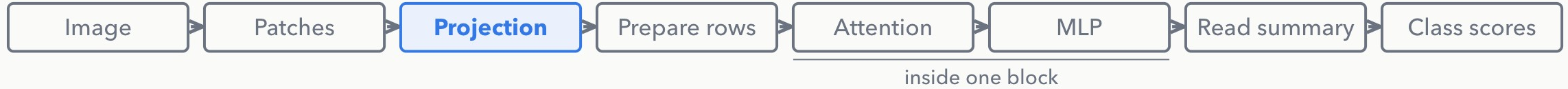


P63 and P64 produce different features using the same learned parameters.

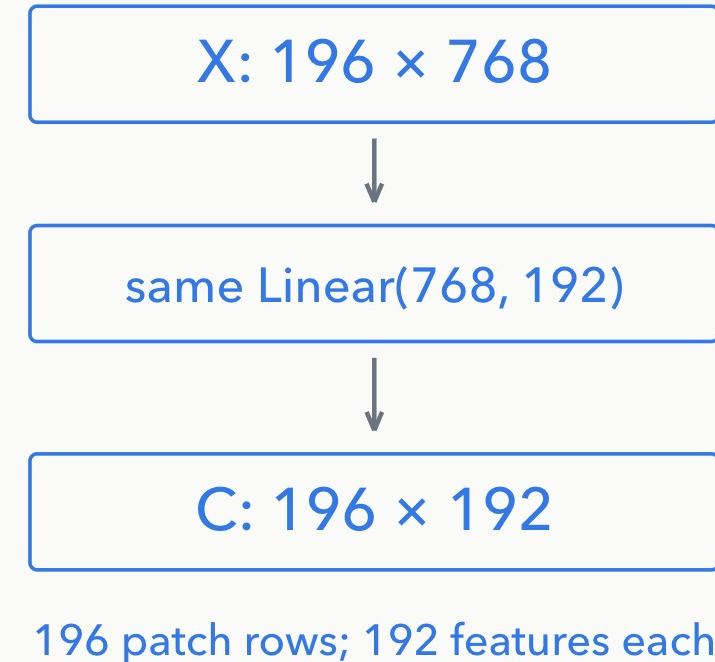
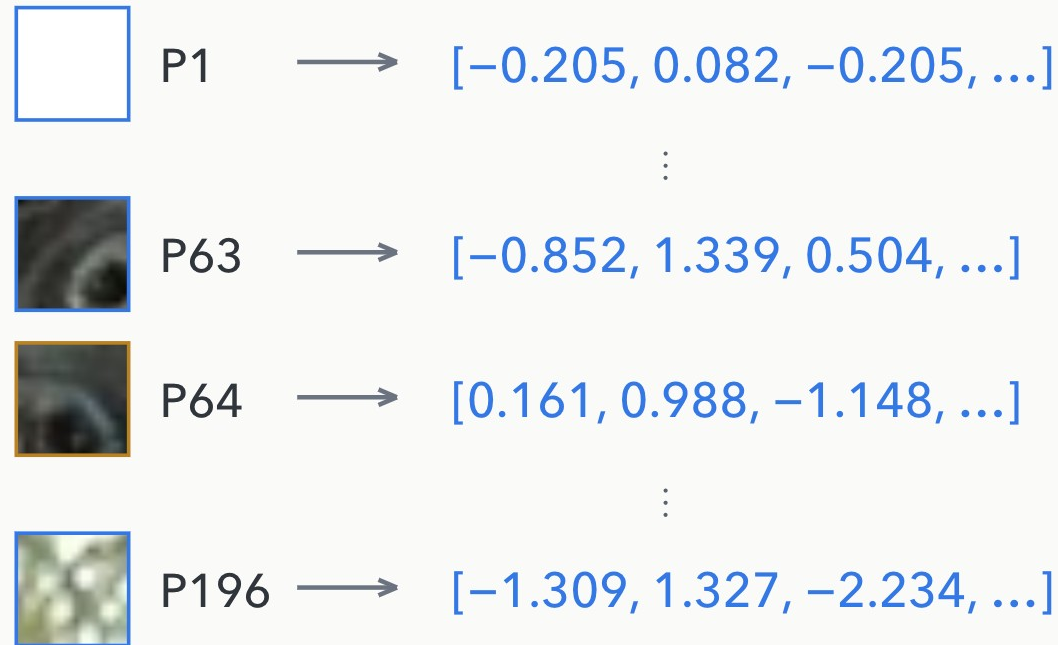
P64 is the patch immediately to the right of P63. Its different pixels give a different embedding. Both patches use the same weights and biases, and both produce 192 output features.

Stack the 196 output rows into C

Photo walkthrough · Step 10 of 12



One content embedding for every image patch



Repeat the same projection for all 196 patches and keep their order. The input matrix X is 196×768 . The output matrix C is 196×192 : one content embedding per image patch.

SECTION

03

Prepare the rows, then classify the image

Our photograph has become 196 rows of patch features.

Where is each patch? How do we get one image summary?

Add location

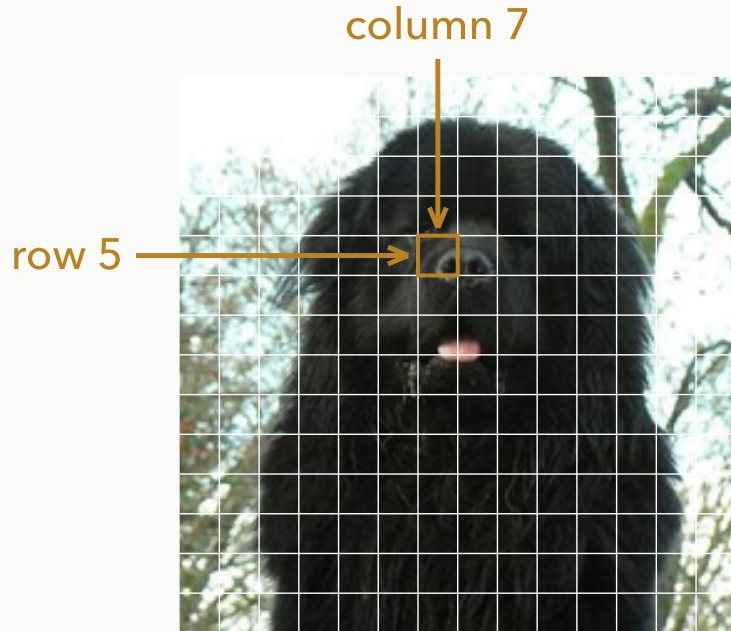
→ Introduce CLS

→ Resume the forward pass

Keep the photograph fixed. Give each patch its location, introduce the summary row, then follow the complete input through the classifier.

Give each patch its location in the photograph

Keep the photograph exactly as it is.



P63 · one of 196 patches

Content c_{63} : features from this crop
Already computed by Linear(768, 192)

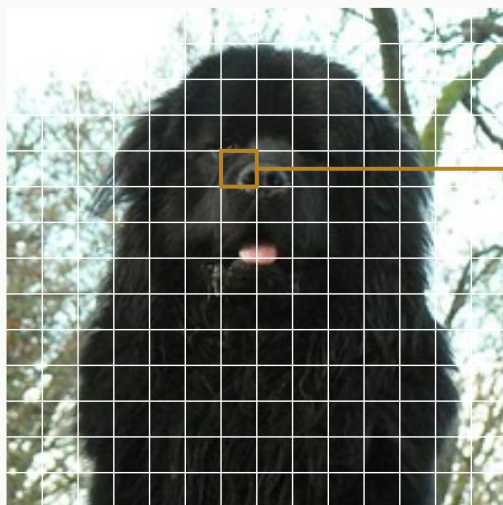
Position p_{63} : a vector for this grid slot
Row 5, column 7 → patch number 63

The shared pixel projection was never given the row or column. Add that information next.

Recognizing a face depends on how its parts are arranged. c_{63} describes this crop; p_{63} tells attention where it belongs. Add them so the model can use both appearance and layout.

Position is a learned lookup table

P: 196 grid slots $\times$ 192 coordinates



P63: row 5, column 7

P1	[... 192 coordinates ...]
P2	[... 192 coordinates ...]
$\vdots$	$\vdots$
P63	[-0.815, -0.083, ...]
$\vdots$	$\vdots$
P196	[... 192 coordinates ...]

Slot 63 selects p_{63} for every image.

$196 \times 192 = 37,632$ learned numbers

Compare: patch projection has 147,648 parameters.

Each grid slot has a trainable 192-number row, shared across images. Slot 63 always selects p_{63} ; its pixel-derived content changes from photo to photo. The patch-position table has 37,632 parameters.

Add position to these content rows

Photo walkthrough · Step 11 of 12



P63

We already computed c_{63} using the patch projection.

Add its position row; repeat for every image patch.

content from pixels

$$c_{63} \cdot 1 \times 192$$

$[-0.852, 1.339, \dots]$

row 63 of the learned table

$$+ \quad p_{63} \cdot 1 \times 192$$

$[-0.815, -0.083, \dots]$

input to the block

$$= \quad e_{63} \cdot 1 \times 192$$

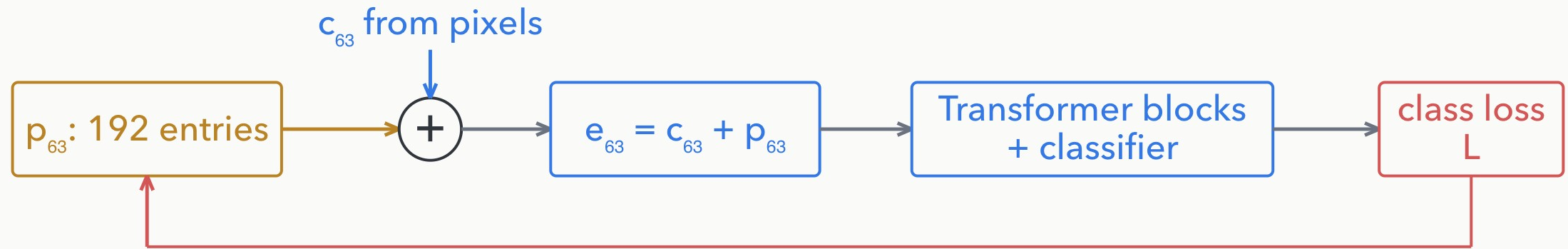
$[-1.666, 1.256, \dots]$

All patch rows: $C (196 \times 192) + P (196 \times 192) = E (196 \times 192)$

c_{63} comes from the patch pixels. p_{63} is learned for its grid location. Add matching coordinates to get e_{63} . All three rows have 192 coordinates; addition does not double the width.

The image loss teaches the position table

Start with small random values; train with the class loss.



Backpropagation: gradients reach every trainable position row.

Illustrative SGD update of one entry

$$p_{63',1}: 0.020 - 0.1 \times 0.30 = -0.010$$

old value – learning rate $\times$ gradient

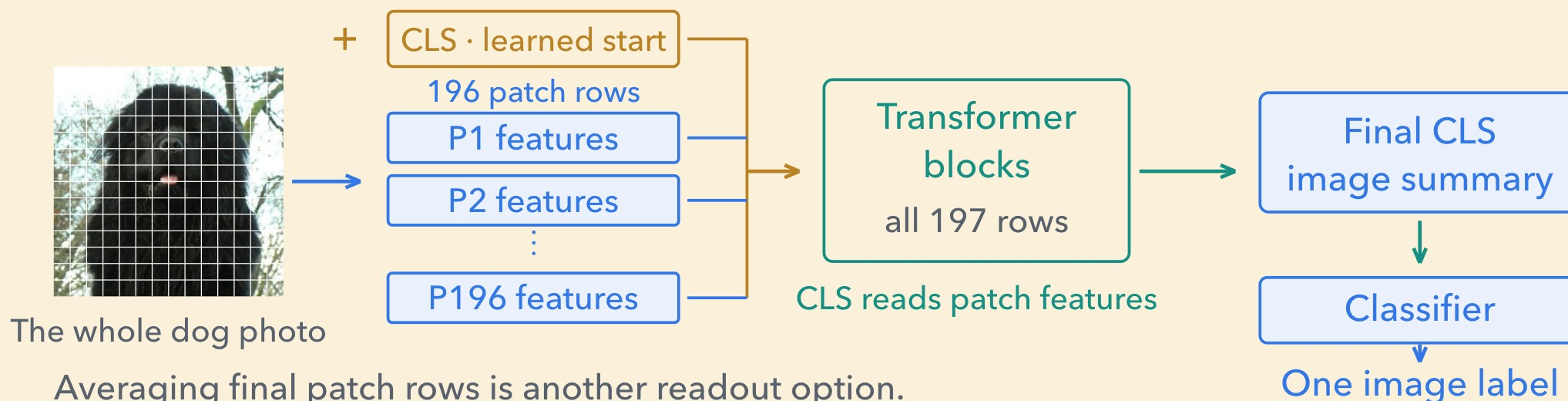
Alternative: fixed sinusoidal
sin/cos(position, coordinate)
No learned table entries.

The class loss supplies gradients for the position table and other model parameters. Training updates the table; inference reuses it unchanged. Fixed sinusoidal encodings instead compute position vectors from a formula.

A SHORT DETOUR · WHY ADD CLS?

One image label from 196 patch rows

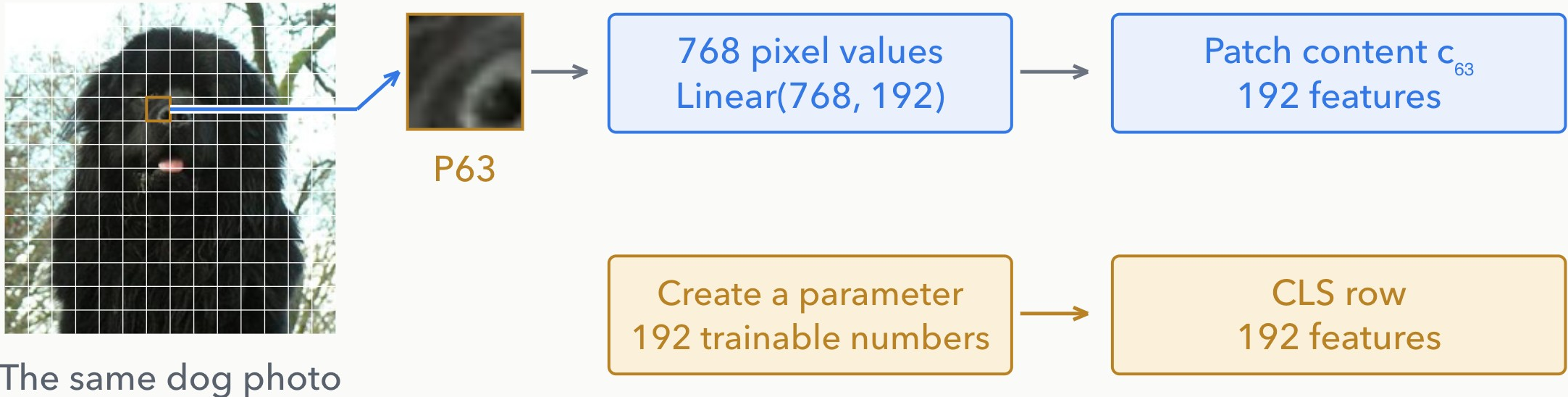
Which representation should the classifier read?



Patch embeddings give us many rows, while the target is one label for the photograph. We need a way to combine patch information into the representation that the classifier will read.

Add a summary row beside the dog's patch rows

The photograph still has exactly 196 patches.

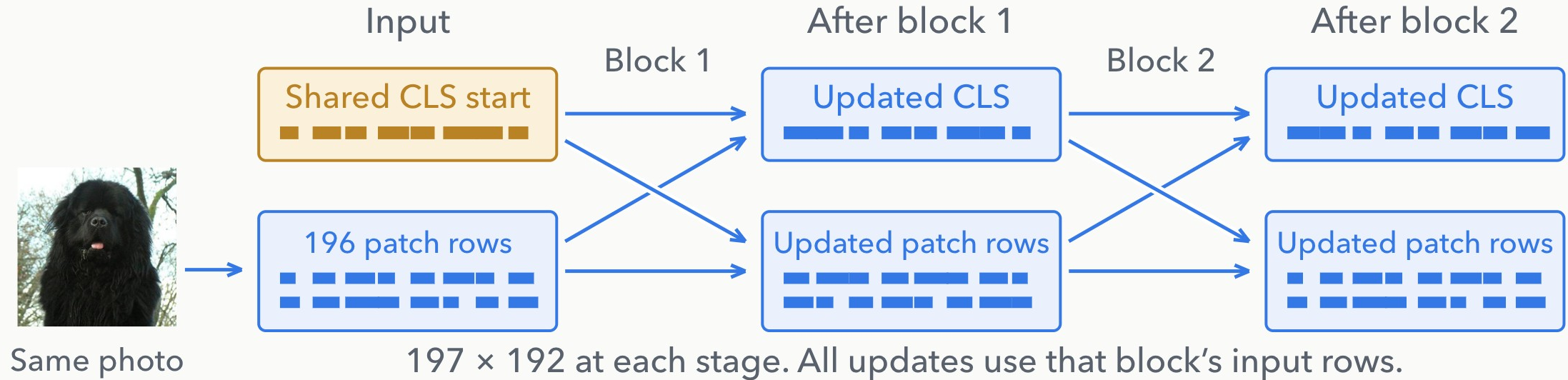


196 image rows + 1 summary row = 197 rows

A patch row comes from pixels. CLS comes from a separate set of model parameters. No extra crop is cut from the photograph. Its 192 coordinates match the width of every patch row.

What makes CLS an image summary?

CLS and patch rows can read one another's current features.



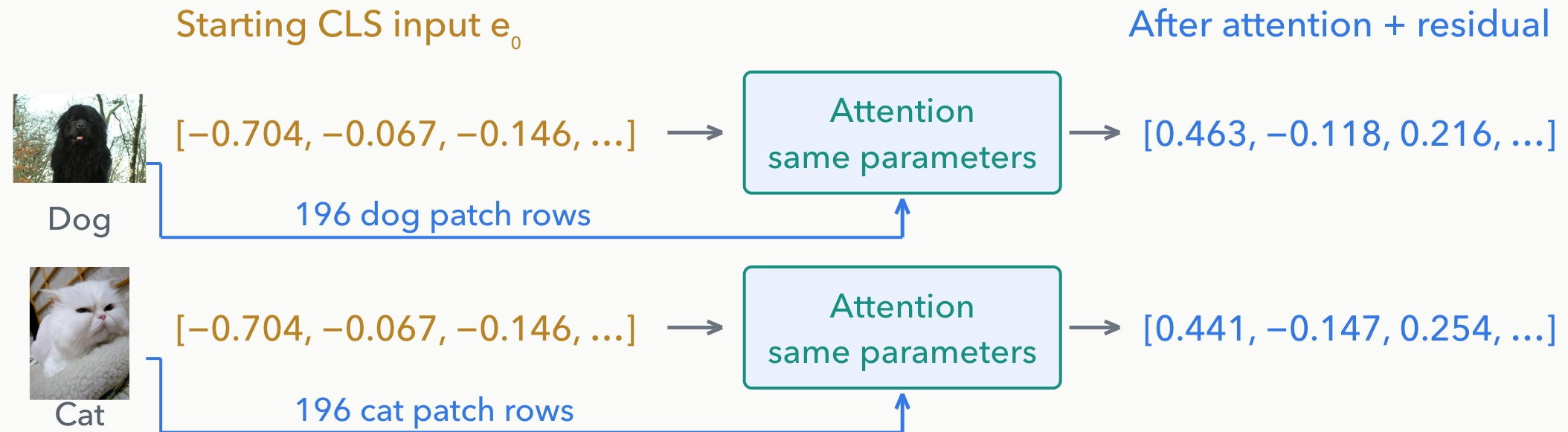
Continue through block 12. The classifier then reads final CLS.
The class loss trains this row to carry useful image information.

CLS can read all patch rows; each patch can read CLS and every patch. All updates use the rows entering that block. Block 2 reads the updated CLS and patches produced by block 1.

The same CLS start reads two different photographs

Shared learned CLS $s = [-0.356, -0.038, -0.072, \dots]$

Add the same position vector before either image enters attention.



First 3 of 192 coordinates shown. Both CLS rows remain 192 numbers wide.

Both images use the same starting CLS and the same model parameters. Their patch rows differ, so attention produces different updates. These are measured CLS values after the first attention update and residual addition.

Add the summary row: $196 + 1 = 197$

Model route · highlighted operation



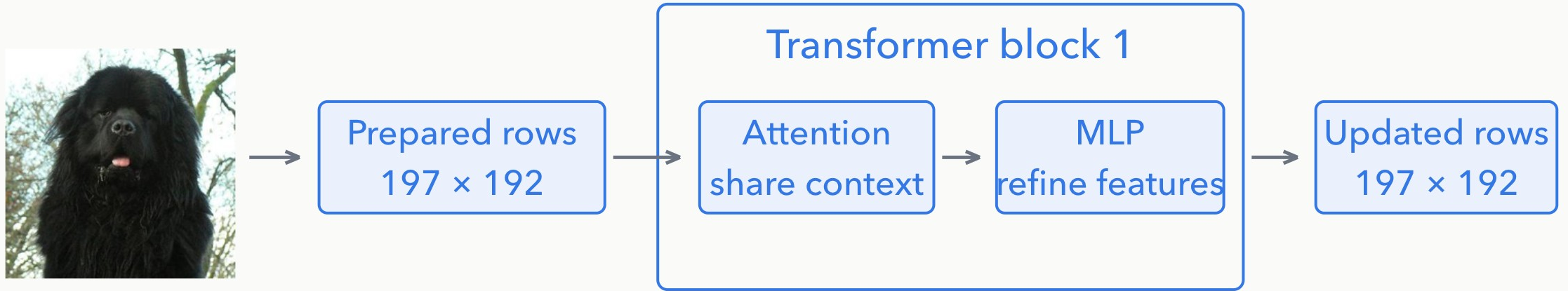
The full sequence entering block 1

CLS	learned token	+	p_0	=	e_0	1×192
P1	c_1	+	p_1	=	e_1	1×192
P63	c_{63}	+	p_{63}	=	e_{63}	1×192
P196	c_{196}	+	p_{196}	=	e_{196}	1×192

Stack: $[e_0; e_1; \dots; e_{196}] \rightarrow E$ has shape 197×192

Each patch keeps its content-plus-position row. CLS gets its own learned position too. Stack the summary and patch rows into E: 197 rows, each with 192 features. Batch size one is omitted here.

Start with one Transformer block



Our dog photograph

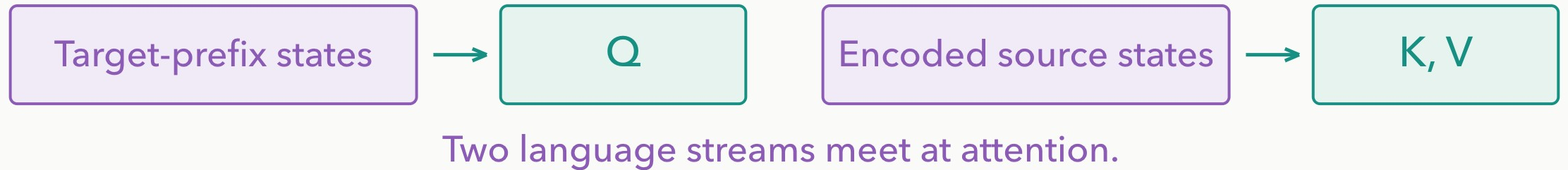
Each row keeps 192 features. Its numbers are updated.

First, let's follow this one block.

Our 196 patch rows and one CLS row enter block 1. Attention shares information across rows; the MLP transforms each row. The output still has 197 rows and 192 features per row.

Self-attention: Q, K and V share the same input

TRANSLATION • cross-attention



ViT • self-attention

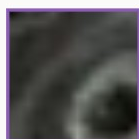


ViT has one stream. Translation had two streams.

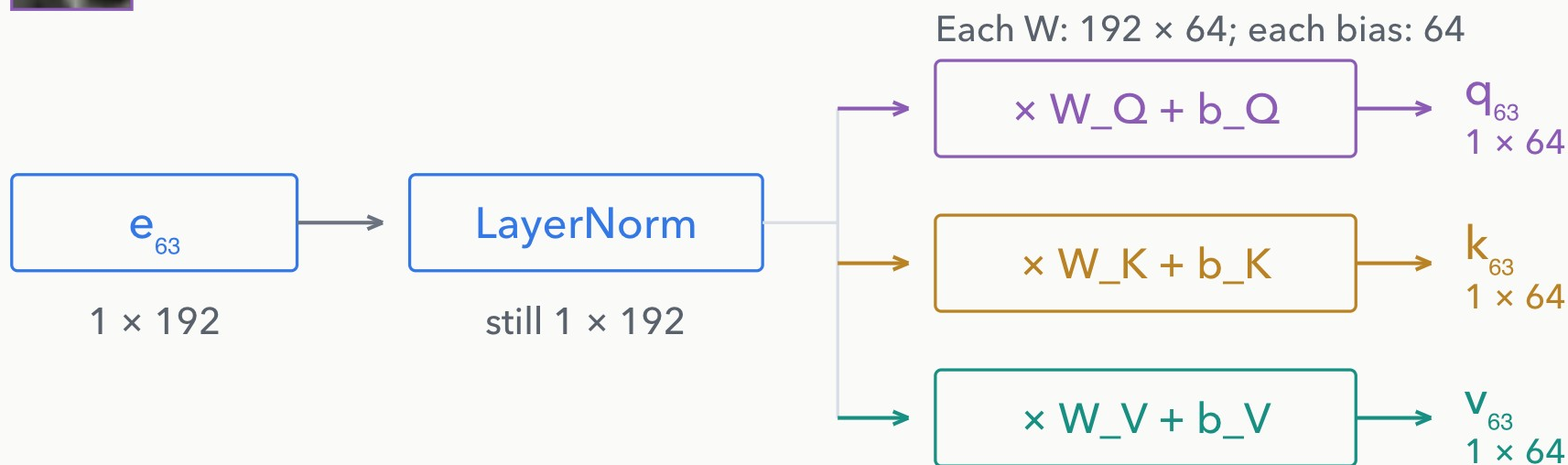
Self-attention uses one input sequence. Cross-attention takes queries from one stream and keys and values from another.

Make queries, keys and values from these rows

Photo walkthrough · Step 12 of 12



P63: pixels $\rightarrow$ patch projection $\rightarrow c_{63} \rightarrow$ add $p_{63} \rightarrow e_{63}$
All 197 rows are ready. Follow patch 63 into the first block.



LOCAL ROLES

Q / receiver

K / source

V / message

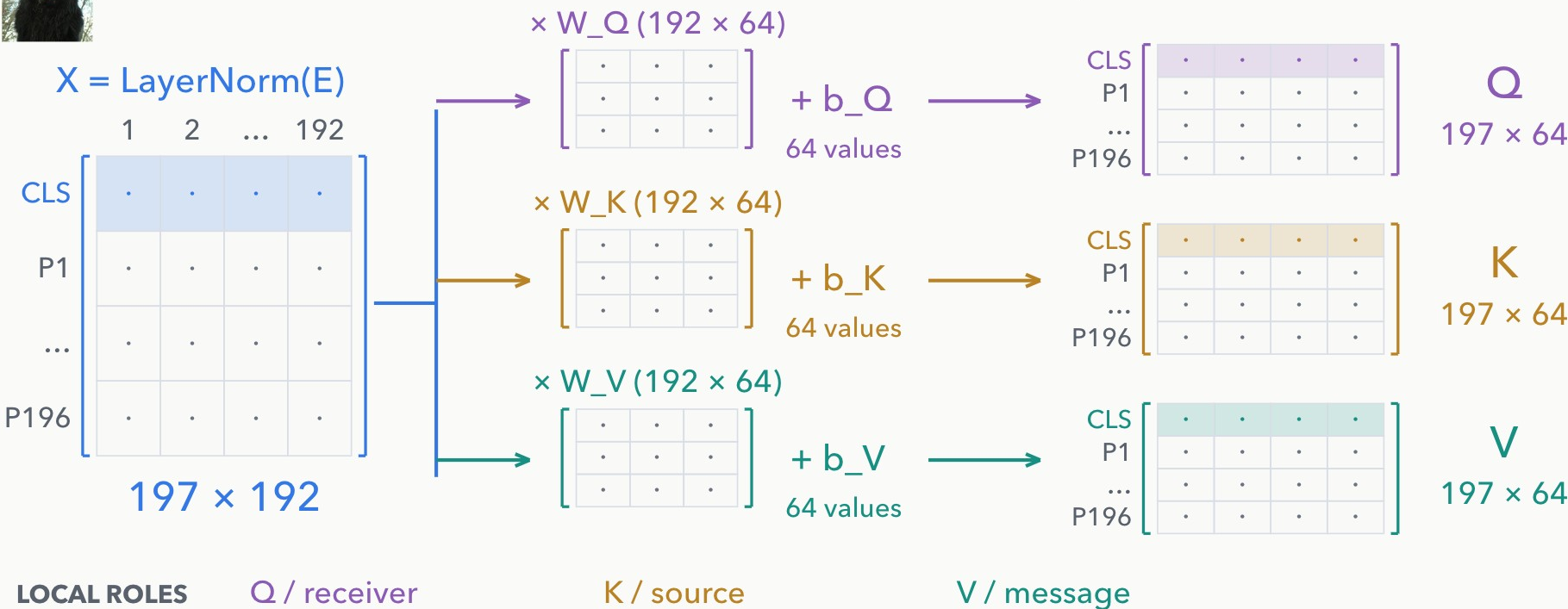
LayerNorm rescales a row's features and keeps its width. We name it here and study its details later. Three learned maps then make Q, K and V; the diagram shows one of three heads.

The same input matrix feeds three learned projections

Model route · highlighted operation



Follow one head in block 1 · 192 features / 3 heads = 64 per head



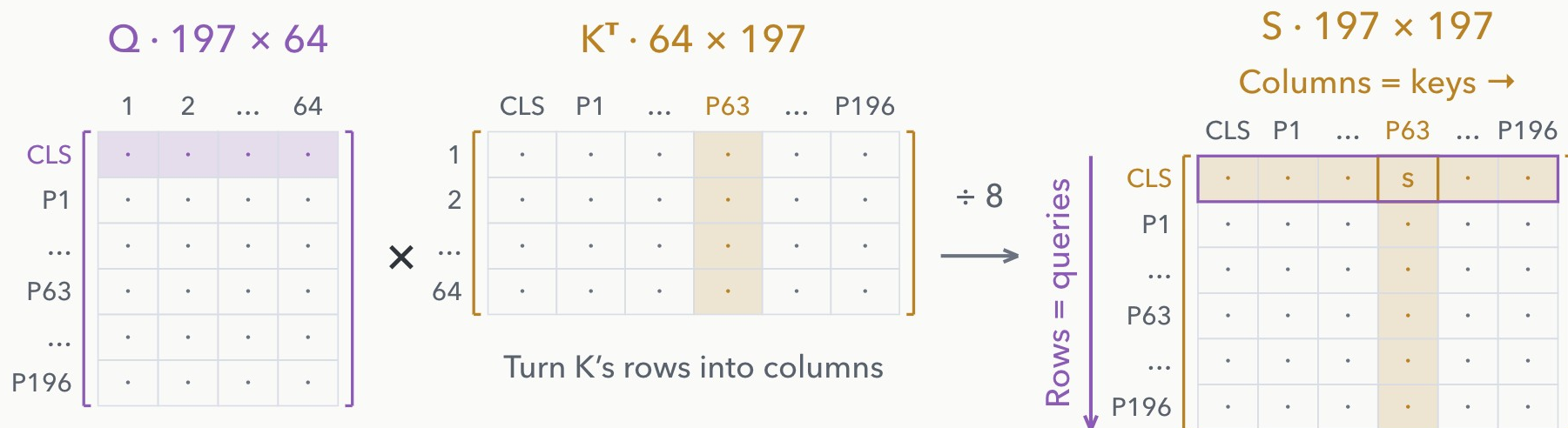
One normalized matrix X feeds three different learned projections. Q, K and V each contain 197 rows of 64 features, in the same token order. This is one head; dots stand for omitted values.

One query-key comparison fills one matrix cell

Model route · highlighted operation



Rows: receiving queries. Columns: source keys. CLS is first on both axes.



Highlighted cell: $s(\text{CLS}, \text{P63}) = q(\text{CLS}) \cdot k(\text{P63}) / 8$

CLS row: what CLS reads. CLS column: how each query scores CLS.

LOCAL ROLES

Q / receiver

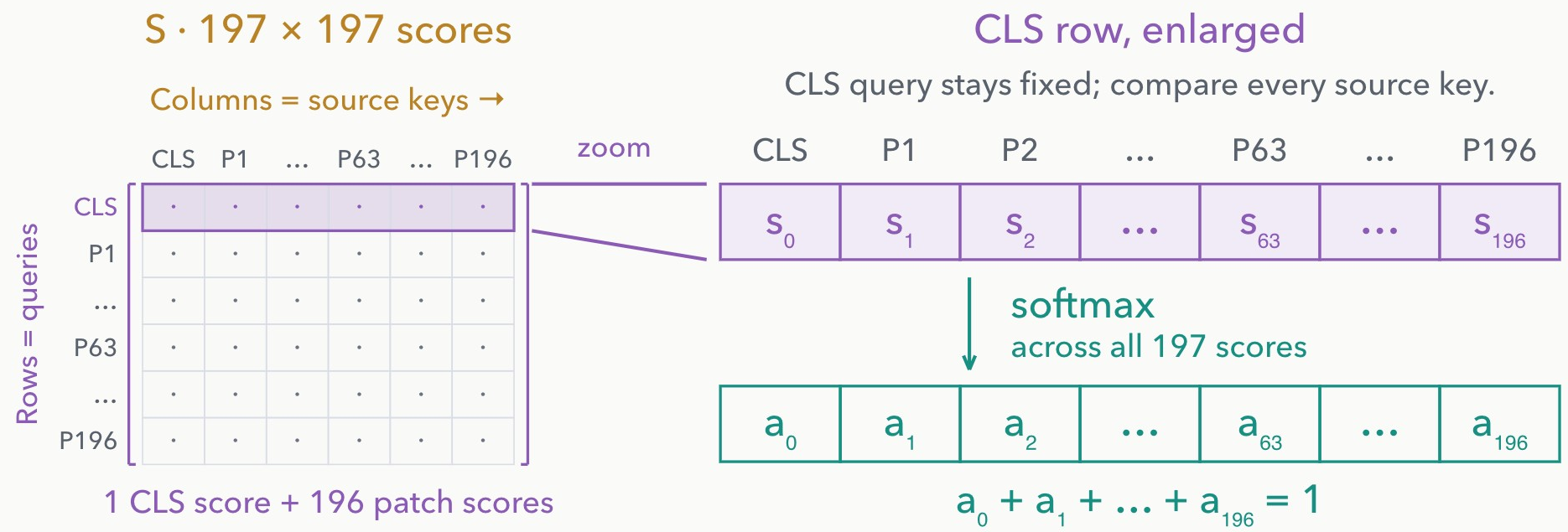
K / source

V / message

Fix the receiving query to CLS and move across the columns. These scores determine CLS's source weights after softmax. Moving down the CLS column changes the query while keeping the CLS key fixed.

Follow the CLS row from scores to weights

Model route · highlighted operation



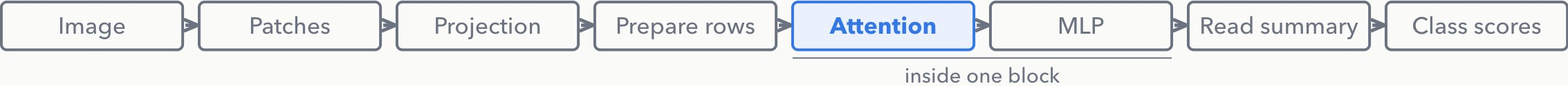
Patch queries update patches → the next block's CLS reads those updated patches.

LOCAL ROLES Q / receiver K / source V / message

Earlier blocks update patches so later CLS queries can read their new features. In the final block, a CLS-only readout could compute just the CLS output, using all 197 incoming keys and values.

Turn each query's 197 scores into 197 source weights

Model route · highlighted operation



S: matching scores

	CLS	P1	...	P63	...	P196
CLS	.	.	.	s	.	.
P1	.	.	.	.	.	.
...	.	.	.	.	.	.
P63	.	.	.	.	.	.
...	.	.	.	.	.	.
P196	.	.	.	.	.	.

softmax
across each row

A: attention weights

	CLS	P1	...	P63	...	P196
CLS	.	.	.	a	.	.
P1	.	.	.	.	.	.
...	.	.	.	.	.	.
P63	.	.	.	.	.	.
...	.	.	.	.	.	.
P196	.	.	.	.	.	.

Highlighted CLS row: sum = 1

a(CLS, P63): how much weight CLS gives to P63's value row.

LOCAL ROLES Q / receiver K / source V / message

Apply softmax across each score row. The CLS row becomes weights over CLS and all 196 patches, summing to one. Each other query gets its own weight row. These weights tell us how to mix the value rows.

Every image row can read every image row

Model route · highlighted operation



Text: predict the next token

	t ₁	t ₂	t ₃	t ₄	t ₅	t ₆
t ₁	•	×	×	×	×	×
t ₂	•	•	×	×	×	×
t ₃	•	•	•	×	×	×
t ₄	•	•	•	•	×	×
t ₅	•	•	•	•	•	×
t ₆	•	•	•	•	•	•

Read the current and earlier positions

ViT: classify the whole image

	CLS	P1	...	P63	...	P196
CLS	•	•	•	•	•	•
P1	•	•	•	•	•	•
...	•	•	•	•	•	•
P63	•	•	•	•	•	•
...	•	•	•	•	•	•
P196	•	•	•	•	•	•

Read every row, including itself

Filled cell = permitted connection. × = blocked. This shows access, not weight size.

No causal mask here: CLS can read P196, and P196 can read CLS.

LOCAL ROLES

Q / receiver

K / source

V / message

Next-token prediction hides later text positions. Here the complete photograph is available before classification. Every query can use every key and value, including itself. The ViT attention matrix has no causal triangle removed.

A message for CLS works like a message for bank

Model route · highlighted operation



Choose a receiver. Mix source values to make a message for that receiver.

Text · receiver: bank



Image · receiver: CLS



Same weighted-sum operation. Different receiver and sources.

LOCAL ROLES Q / receiver K / source V / message

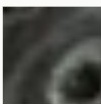
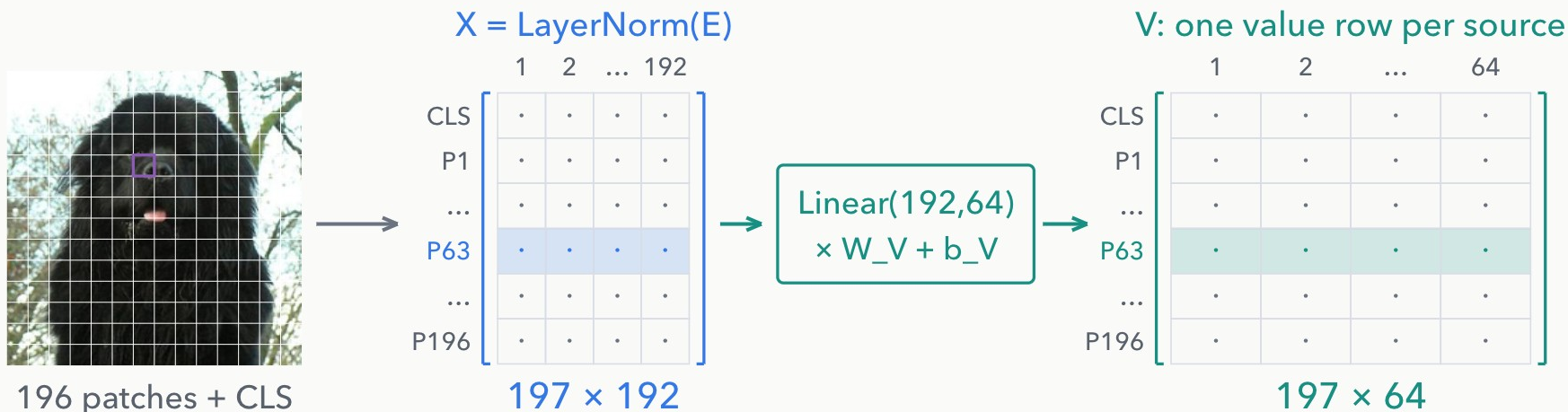
In text, bank receives a weighted mixture of allowed token values. Here, CLS receives a weighted mixture of image-token values, including its own. The query chooses the receiver; the values supply the message.

The dog's feature rows become value rows

Model route · highlighted operation



Same dog · block 1 · head 1



$P63 \rightarrow v_{63} = [-0.174, -1.201, \dots, -1.308]$

64 learned features · rounded values from this dog

LOCAL ROLES

Q / receiver

K / source

V / message

Use the same normalized input X that produced Q and K. The value projection makes 64 features for each of its 197 rows. P63's crop identifies one source; its value row contains learned features.

One weight scales all 64 features in its value row

Model route · highlighted operation



1 · Pick receiver CLS

$A \cdot 197 \times 197$ weights

Columns = source keys →

	CLS	P1	...	P63	...	P196
Rows = queries	CLS	.	.	.	a_{63}	.
	P1	.	.	.	.	.
	...	.	.	.	.	.
	P63	.	.	.	.	.
	...	.	.	.	.	.
	P196	.	.	.	.	.

$A[\text{CLS}, \text{P63}] \cdot \text{one weight}$

0.000504

2 · Pick source P63

3 · Pick P63's value row

$V \cdot 197 \times 64$ values

Columns = value features →

	1	2	...	64	
Rows = sources	CLS	.	.	.	.
	P1	.	.	.	.
	...	.	.	.	.
	P63	$V_{63/1}$	$V_{63/2}$	.	.
	...	.	.	.	.
	P196	.	.	.	.

$V[\text{P63}, :] \cdot \text{all 64 features}$

-0.174	-1.201	...	-1.308
-0.000088	-0.000606	...	-0.000659

LOCAL ROLES Q / receiver K / source V / message

4 · Scale each feature

Contribution to CLS · 1×64

=

Use Next to select CLS's row in A, P63's column, P63's row in V, then its feature coordinates. One scalar scales all 64 values. This contribution is addressed to CLS.

Each source contributes a weighted value row

Model route · highlighted operation



Keep the receiver fixed: every weight below comes from the CLS row.

Source	CLS weight	Value row · 64 features	Contribution to CLS
CLS CLS	0.860556	0.418 0.071 ...	0.359857 0.060762 ...
P1	0.002298	-0.211 -0.027 ...	-0.000485 -0.000063 ...
P63	0.000504	-0.174 -1.201 ...	-0.000088 -0.000606 ...
P196	0.000233	-1.607 1.181 ...	-0.000374 0.000275 ...

Four source examples. Every contribution still has 64 features.

LOCAL ROLES Q / receiver K / source V / message

Repeat the P63 calculation for other sources. Each uses its own CLS weight and value row. These are four measured examples from the same head; every contribution is a 64-feature vector addressed to CLS.

Add the contributions to make one CLS message

Model route · highlighted operation



Add the same feature across sources. Keep all 64 features.

Weighted contributions for CLS · each 1×64				
CLS		0.359857	0.060762	...
P1	+	-0.000485	-0.000063	...
P63	+	-0.000088	-0.000606	...
P196	+	-0.000374	0.000275	...
Other 193	+	-0.005982	-0.003703	...
				→
				Message for CLS 0.353 0.057 ... 1×64

197 source contributions → one message for the receiving CLS query.

LOCAL ROLES Q / receiver K / source V / message

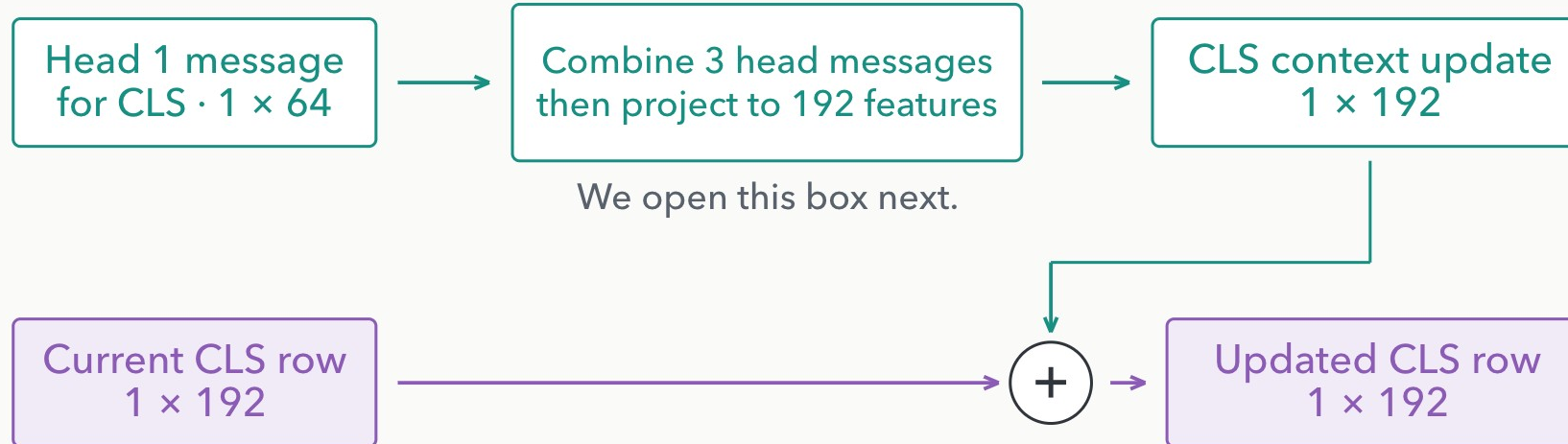
Add corresponding coordinates from all 197 weighted value rows, including CLS itself. The result has 64 features: one message for CLS in this head. The other 193 sources are grouped to keep the addition readable.

Where does the CLS message go?

Model route · highlighted operation



Same pattern as text: current token row + context update → updated token row.



The CLS representation changes. It is still the same receiving token.

LOCAL ROLES

Q / receiver

K / source

V / message

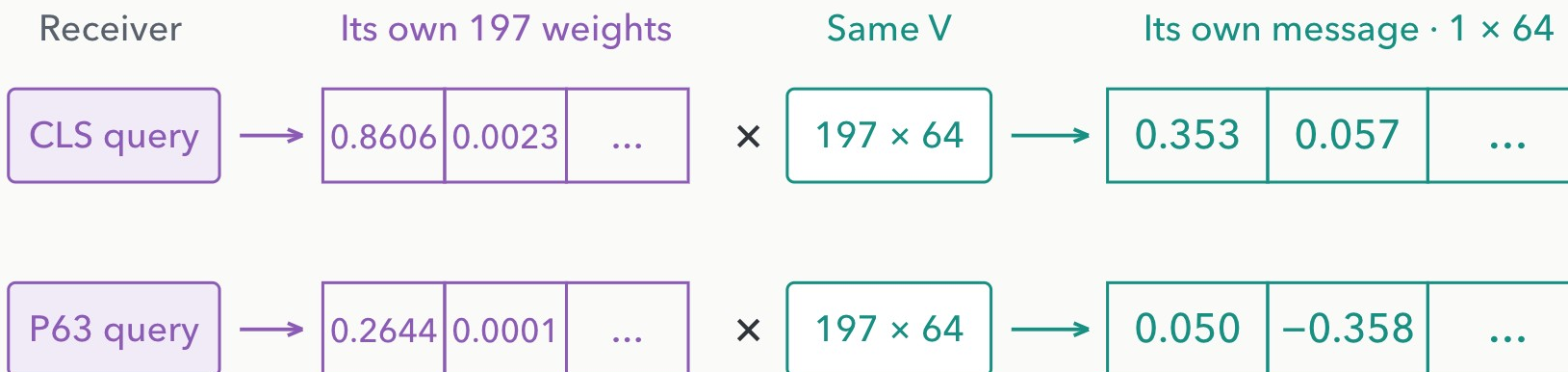
The CLS messages become a 192-feature context update after combining heads and projecting. Add this update to the CLS row entering the block. The result is a new representation of the same CLS token, as in text.

Each query gets its own message

Model route · highlighted operation



Keep the source values. Change the receiving query and its weights.



CLS's messages update CLS. P63's messages update P63. Every row has its own.

LOCAL ROLES

Q / receiver

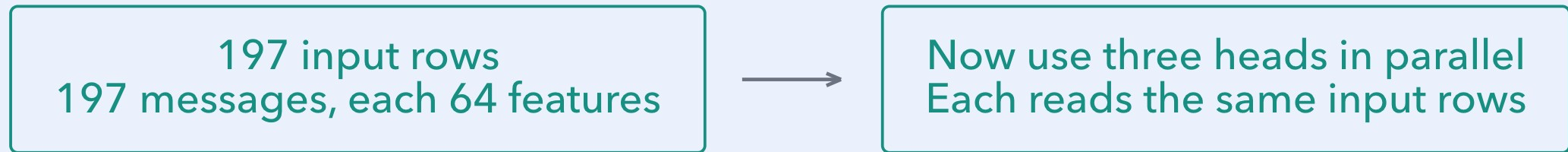
K / source

V / message

P63's query mixes the same source values using P63's weights, producing P63's own message. CLS and every other patch do the same. Each receiver's messages ultimately update its own input row through the residual path.

Head 1 is complete.

Q and K → source weights → mix V → a message for every row.



First combine their messages. Then continue to the MLP.

We have finished one head's attention calculation. This model has three heads. Each starts from the same input rows and computes its own messages. Next, follow those parallel paths and combine their outputs before the MLP.

What might different heads look for in this photograph?

Illustrative possibilities · learned heads are not assigned these jobs

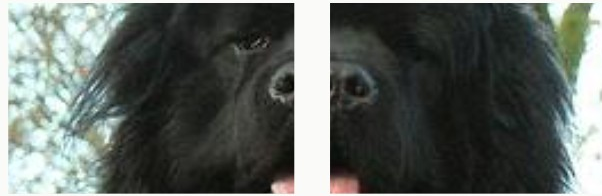
Nearby texture



a different weighted
message

Does this fur continue nearby?

Animal parts



a different weighted
message

Which face clues belong together?

Wider context



a different weighted
message

How does this region fit the scene?

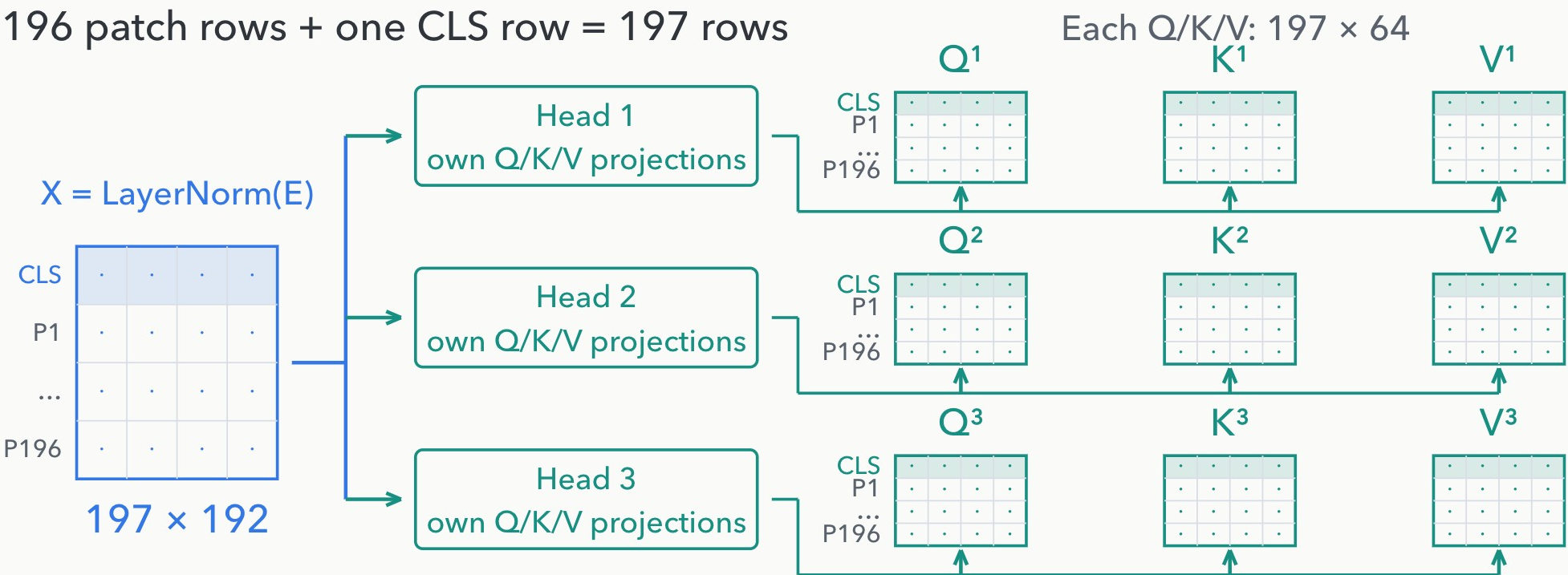
Different heads can learn different matching patterns and different messages. Local texture, relationships between parts and wider context are possible intuitions. These are hypotheses, not measured head labels.

The same rows feed three sets of Q, K and V

Model route · highlighted operation



196 patch rows + one CLS row = 197 rows

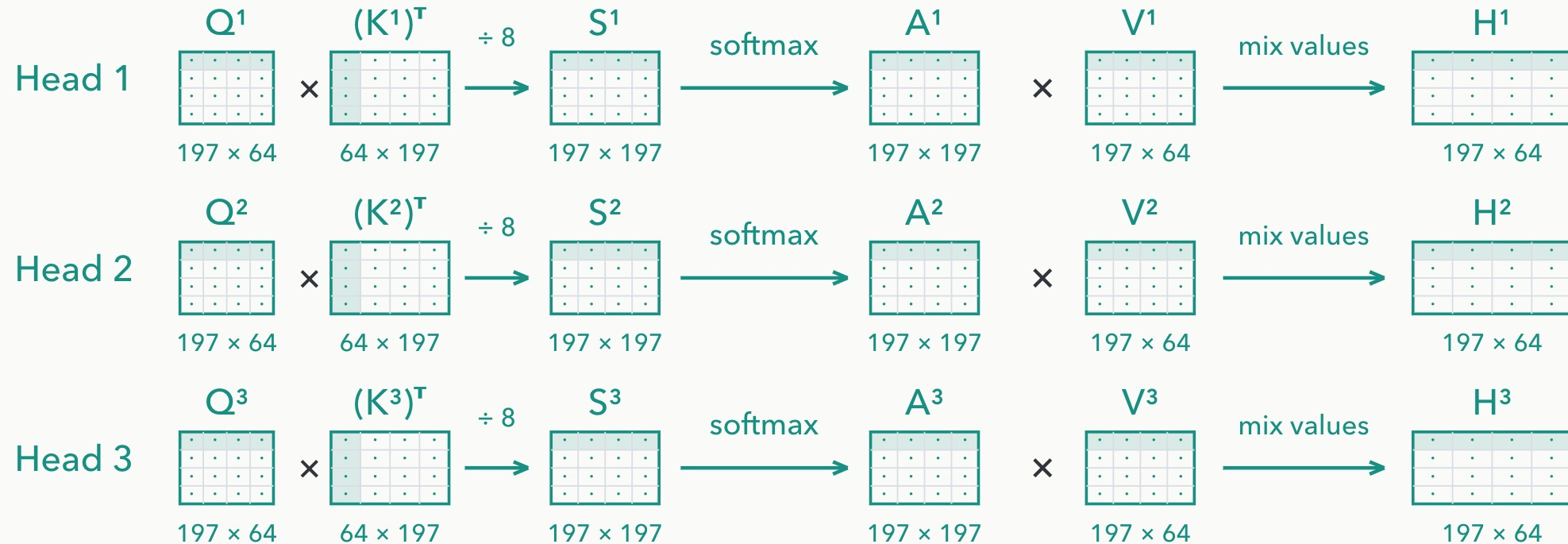
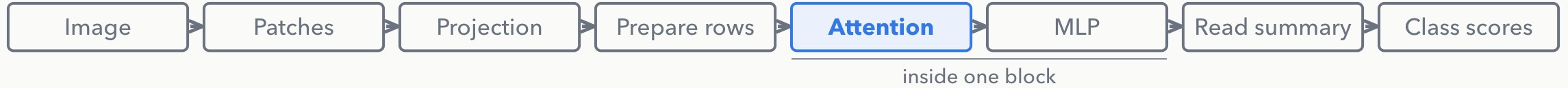


One shared CLS input; each head projects it differently.

All three heads read the same normalized matrix X. Each has its own Q, K and V projection weights and biases. Every output matrix has 197 rows and 64 features. The highlighted first row is always CLS.

Each head repeats the complete attention calculation

Model route · highlighted operation



Each head has its own matching scores, attention weights and messages.

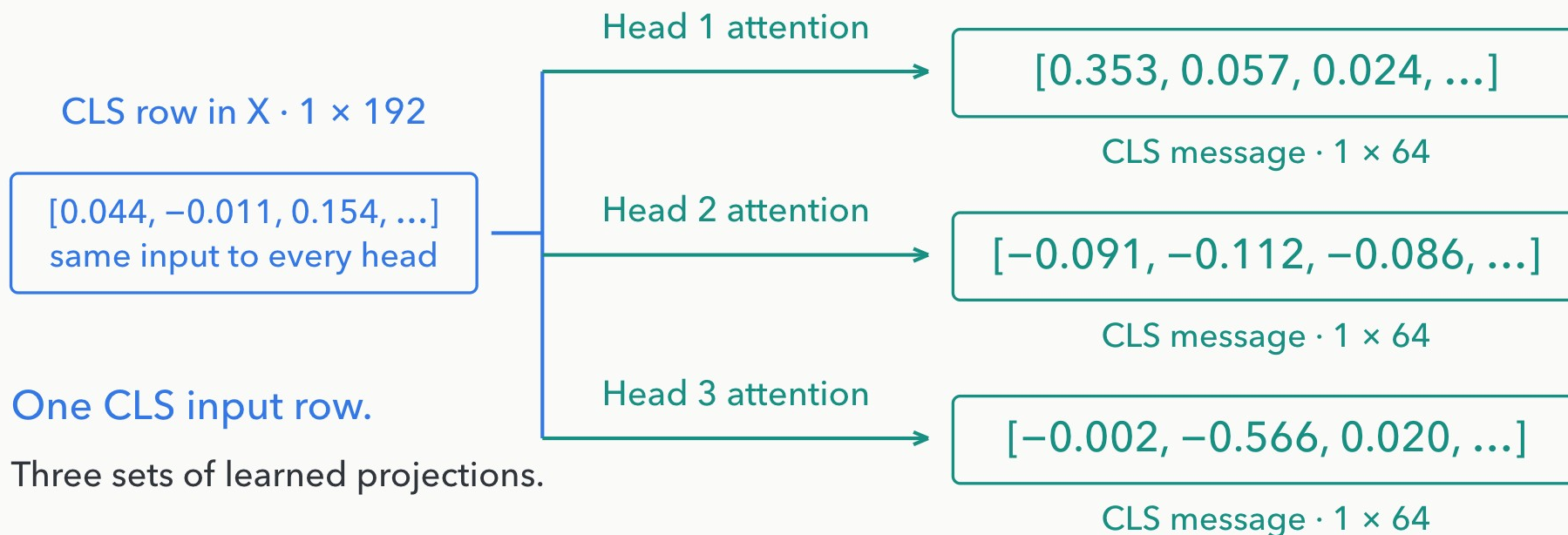
Within each head, compare Q with K , scale the scores, apply softmax across each row, then multiply by that head's V . Each path returns 197 messages of 64 features. The heads operate in parallel.

One CLS input produces three different messages

Model route · highlighted operation



Follow the same dog and the same receiving CLS row.



CLS is the same input row for all heads. Their learned projections produce different queries, keys and values, so their attention computations produce different messages. These measured messages each contain 64 features. There is still one CLS token.

Concatenate the three CLS messages

Model route · highlighted operation



Head 1 CLS message

[0.353, 0.057, ...]

1×64

Head 2 CLS message

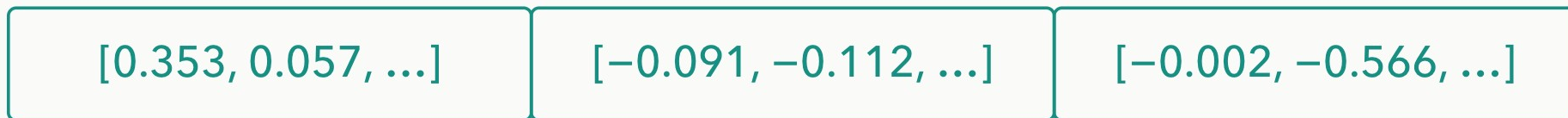
[-0.091, -0.112, ...]

1×64

Head 3 CLS message

[-0.002, -0.566, ...]

1×64



features 1-64

features 65-128

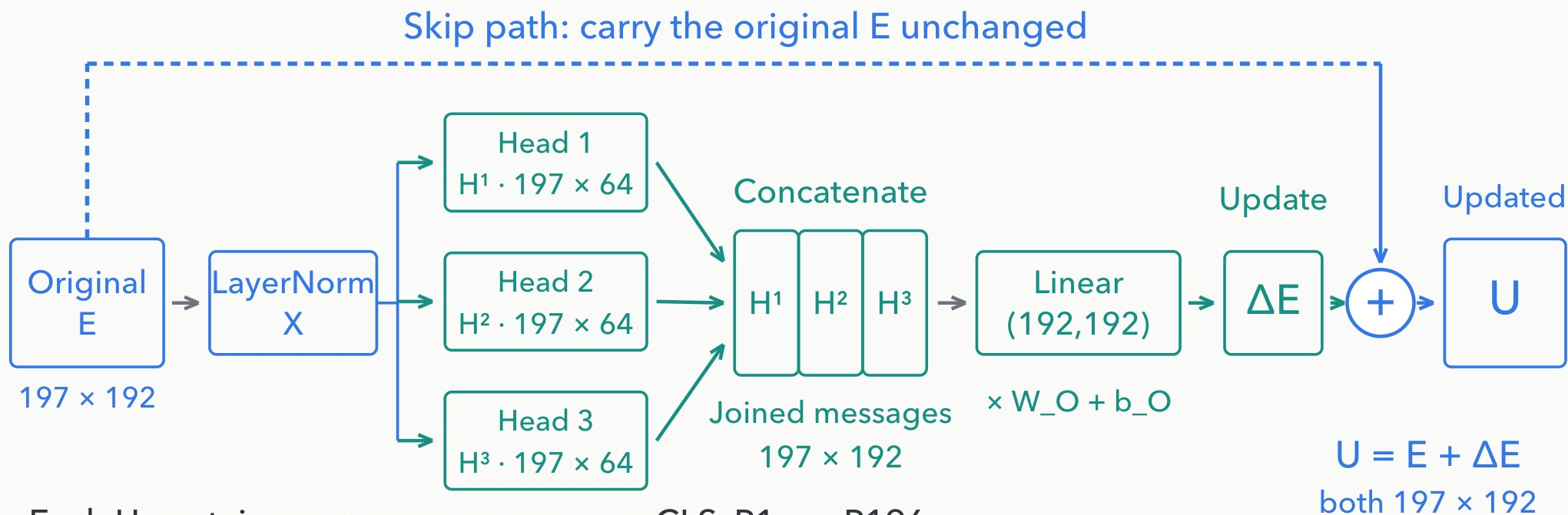
features 129-192

One joined row · $1 \times 192 (64 + 64 + 64)$

Keep all three messages by placing their coordinates side by side. The joined CLS row has 192 features: 64 from head 1, then 64 from head 2, then 64 from head 3. Concatenation has no learned parameters.

Keep the embedding; add the context from attention

Model route · highlighted operation



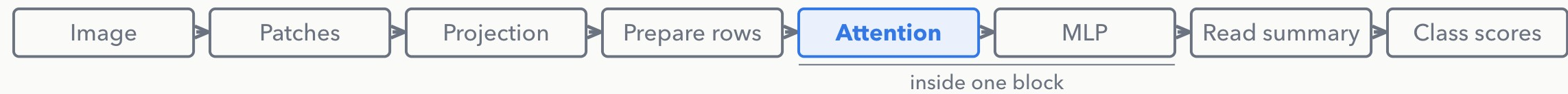
Each H contains one message per row: CLS, P1, ..., P196.

The same residual pattern as text: original embedding + context update.

Follow two paths from E. The heads produce messages; concatenate and project them to make ΔE . The skip path carries E directly to addition. $U = E + \Delta E$ is the contextualized representation passed to the MLP.

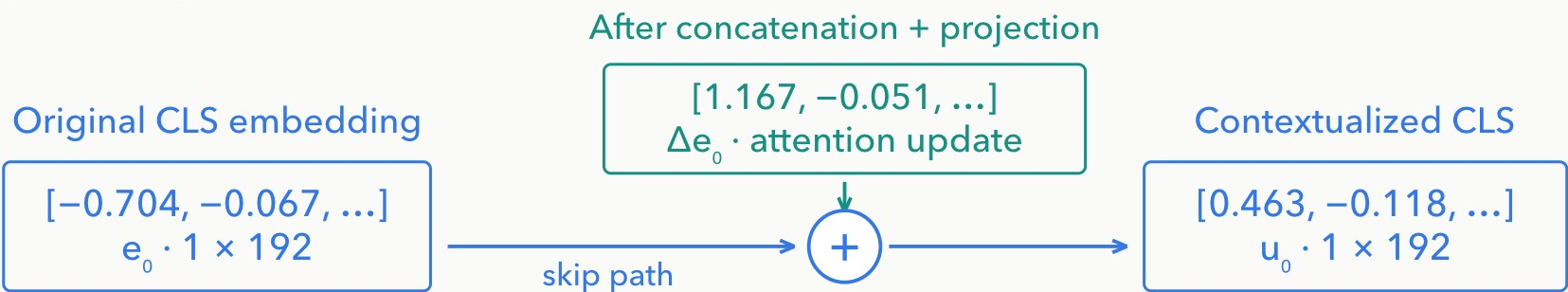
The dog's CLS keeps its input and gains context

Model route · highlighted operation



Same dog · block 1 · zoom in on the CLS row

Every vector below contains 192 numbers.



First coordinate: $-0.704 + 1.167 \approx 0.463$

Next: MLP → later blocks → final CLS → class scores.

LOCAL ROLES

Q / receiver

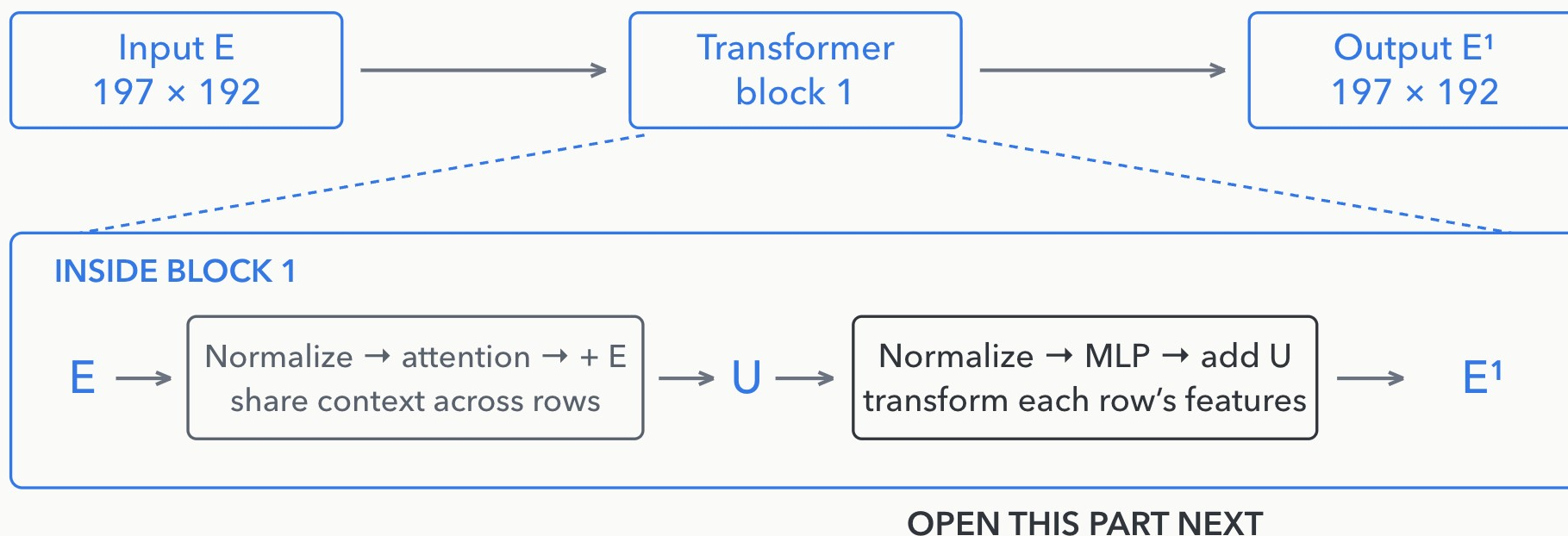
K / source

V / message

Add the projected attention update coordinate by coordinate to the original CLS embedding. This produces a contextualized CLS row of the same width. It continues through the MLP and later blocks before the classifier reads the final summary.

Open block 1: attention, then the MLP

Model route · highlighted operation



All three matrices: 197×192 . One CLS row: 1×192 .

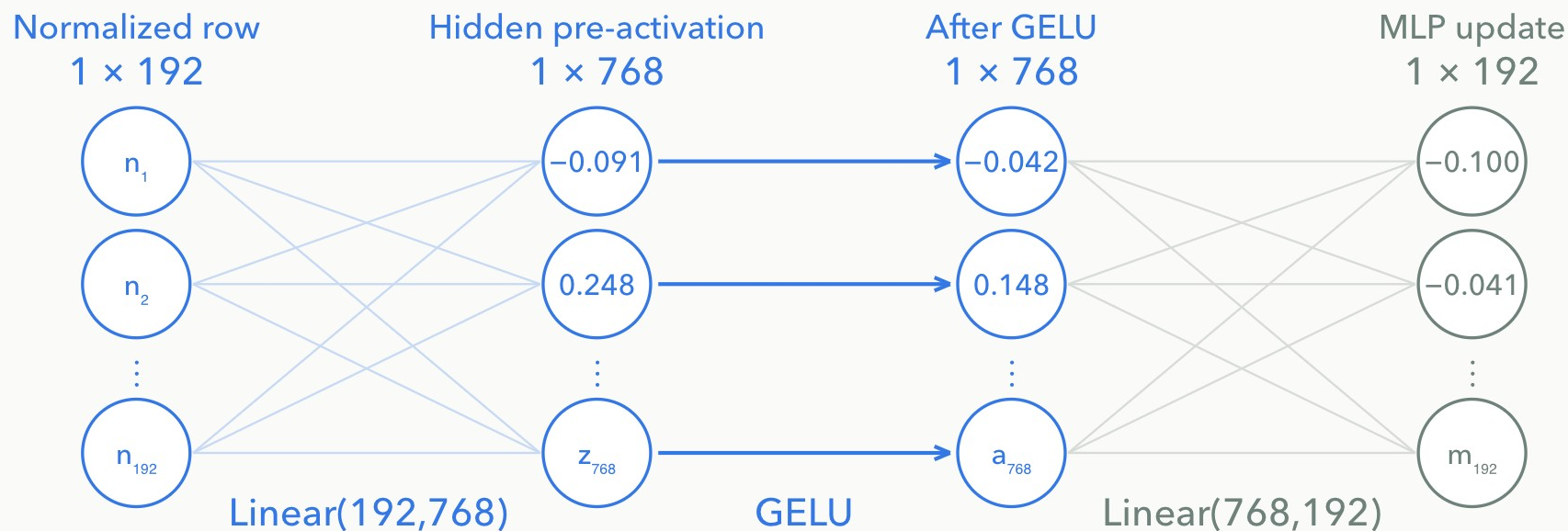
One block makes two updates. Attention lets rows exchange information; the MLP transforms each resulting row. Each branch adds its update to its input. The block receives and returns 197 rows, each 192 features wide.

Open the MLP: 192 inputs, 768 hidden units, 192 outputs

Model route · highlighted operation



Follow the dog's CLS row after LayerNorm.



192 → 768 → 768 → 192 features · biases in both linear layers

Each linear layer connects every input feature to every output unit. GELU transforms each hidden activation. Only a few neurons are drawn; dots omit the rest. The displayed numbers are measured values for this dog's CLS row.

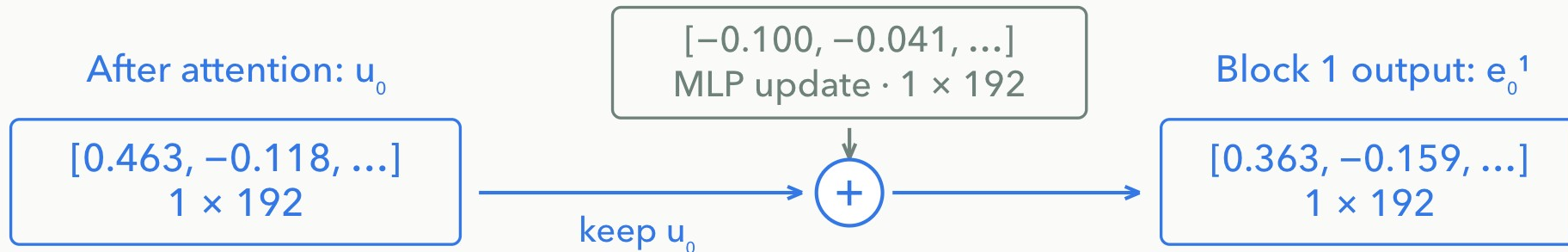
Add the MLP update to finish block 1

Model route · highlighted operation



Same dog · finish the second residual addition

The MLP produces an update for the current CLS row.



First coordinate: $0.463 - 0.100 \approx 0.363$

Do this for CLS and every patch: $E^1 = U + \text{MLP}(\text{LayerNorm}(U))$.

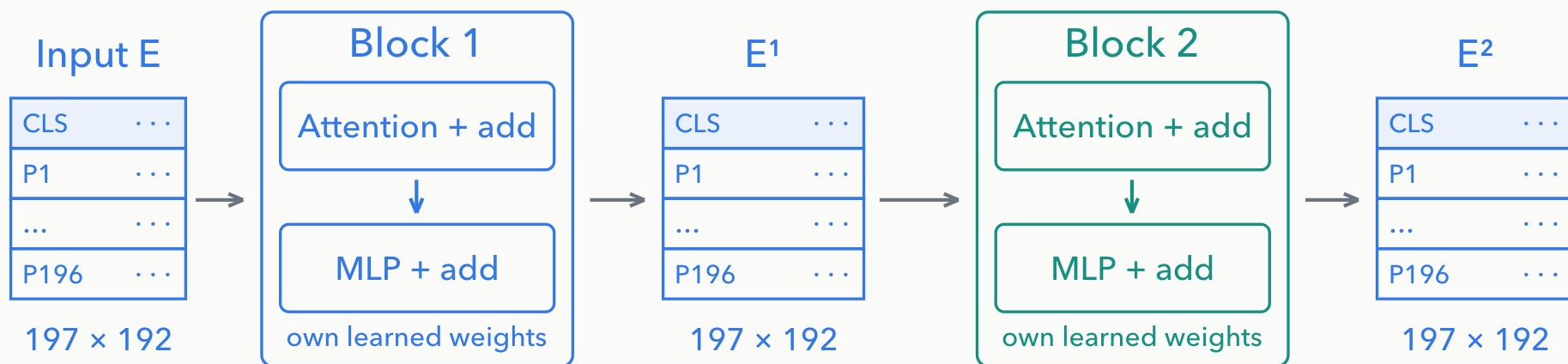
Keep the row produced by attention and add the MLP's update. The result is this row's output from block 1. Apply the same operation to all 197 rows. Their feature values change; their identities and width stay the same.

Pass the complete output of block 1 into block 2

Model route · highlighted operation



Same dog · carry every updated row forward



Block 1's output is exactly block 2's input.

All 197 rows continue. Both blocks return 192 features per row.

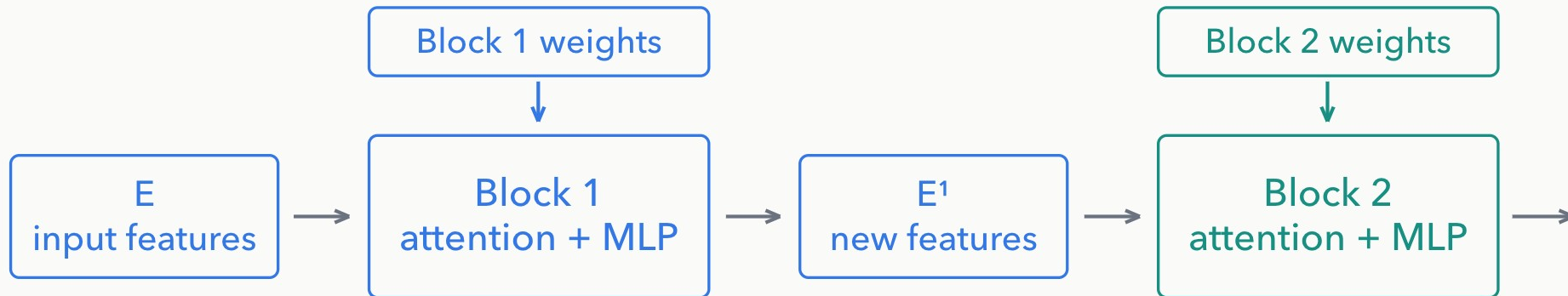
The entire output matrix continues to the next block: CLS and all 196 patch rows. Block 2 has its own attention and MLP weights. It applies the same sequence of operations to the feature values produced by block 1.

What changes as the rows move through the blocks?

Model route · highlighted operation



Feature values flow forward; each block uses its own parameter set.



Recomputed in each block:

$Q, K, V \rightarrow$ attention weights $\rightarrow$ messages $\rightarrow$ MLP activations $\rightarrow$ updated rows

Learned weights stay fixed in this forward pass. Training updates them later.

Each block recomputes queries, keys, values, attention weights and MLP activations from its input. Its learned parameters stay fixed during this forward pass. The next block uses a different parameter set. Matrix dimensions and row identities stay the same.

Continue through the stack, then classify the image

Model route · highlighted operation



One forward pass through 12 distinct Transformer blocks



Every arrow carries all 197×192 feature values.

After block 12 → final normalization → read CLS → class scores

One image. Twelve blocks. One prediction at the end.

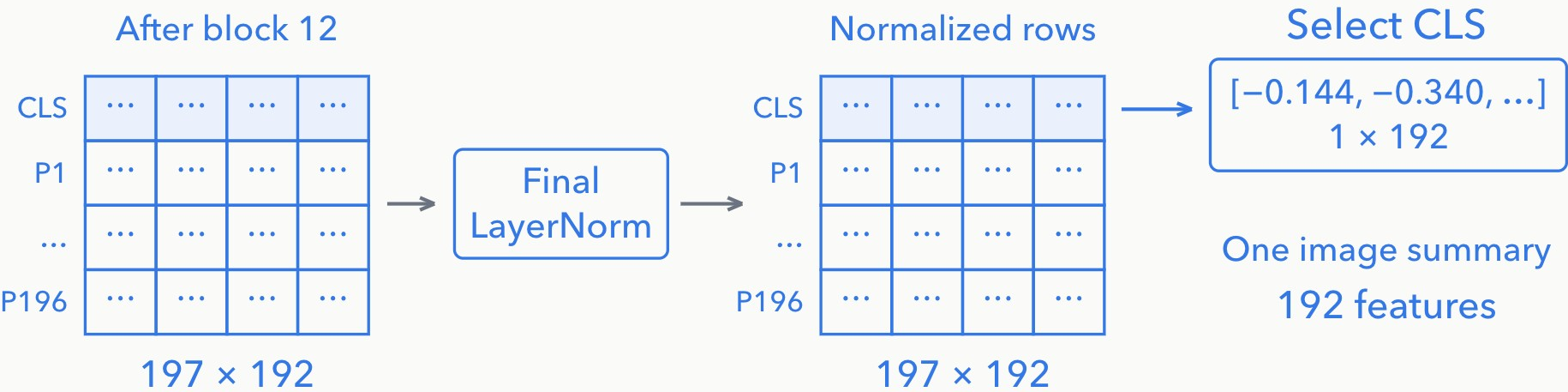
Pass the updated features through blocks 1 to 12 in order. The architecture repeats, with separate learned parameters in each block. After block 12 and final normalization, read CLS and calculate one set of class scores for this image.

Select CLS from the final feature matrix

Model route · highlighted operation



The same dog has now passed through all 12 blocks.



Read the highlighted row; the patch rows have already supplied context.

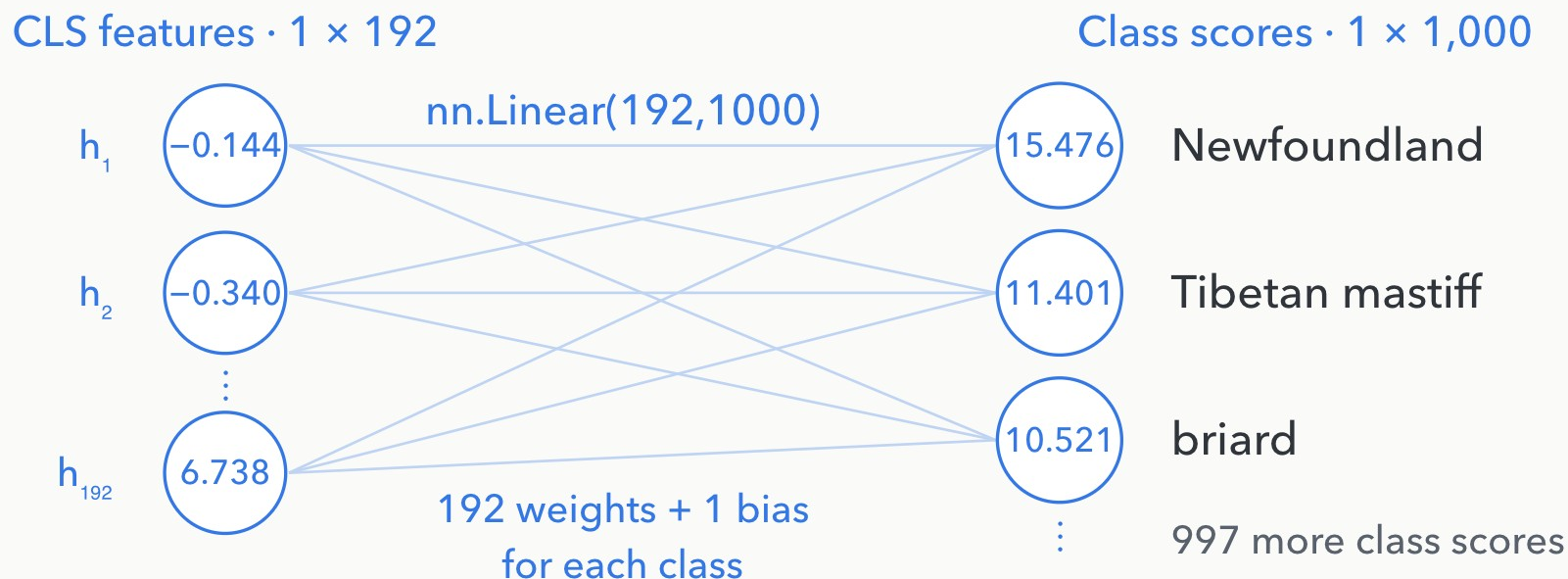
After block 12, all 197 rows are still present. Final LayerNorm keeps the same shape. Select the highlighted CLS row: one image summary with 192 features. Its values now reflect information gathered from this dog’s patches.

Open the classifier: 192 features become 1,000 scores

Model route · highlighted operation



This checkpoint predicts 1,000 ImageNet classes.



One affine layer produces scores. Softmax follows.

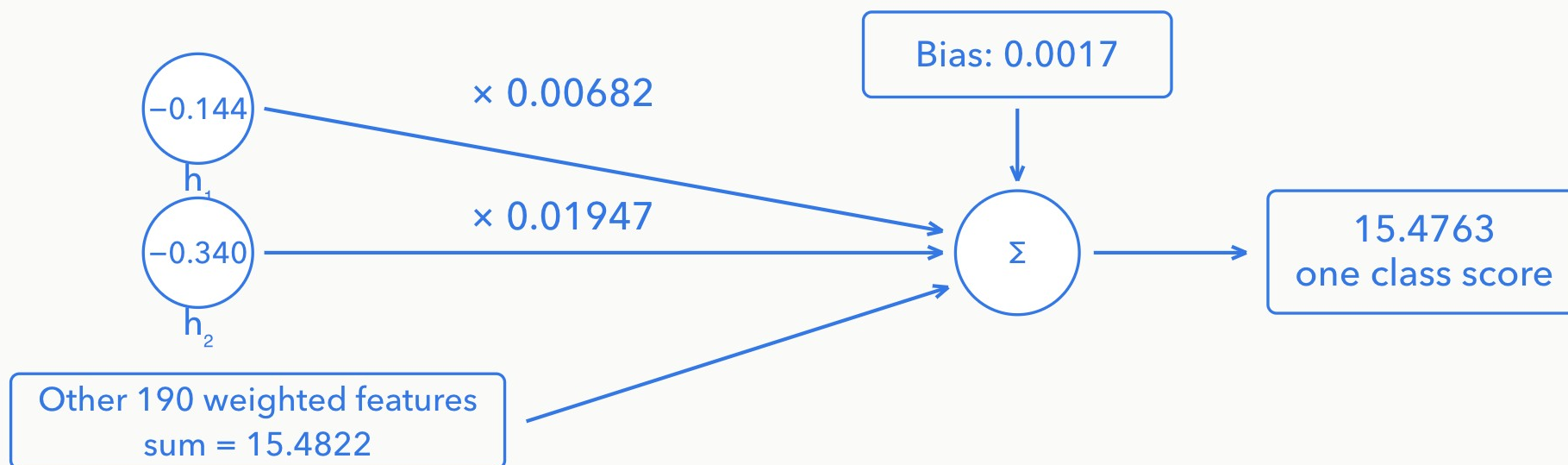
Every class has one output neuron connected to all 192 CLS features. Each neuron uses its own learned weights and bias to produce a score. Only three of the 1,000 class neurons are drawn. Their scores are measured.

One class score is a weighted sum plus a bias

Model route · highlighted operation



Zoom into one output neuron: Newfoundland



$$-0.0010 - 0.0066 + 15.4822 + 0.0017 \approx 15.4763$$

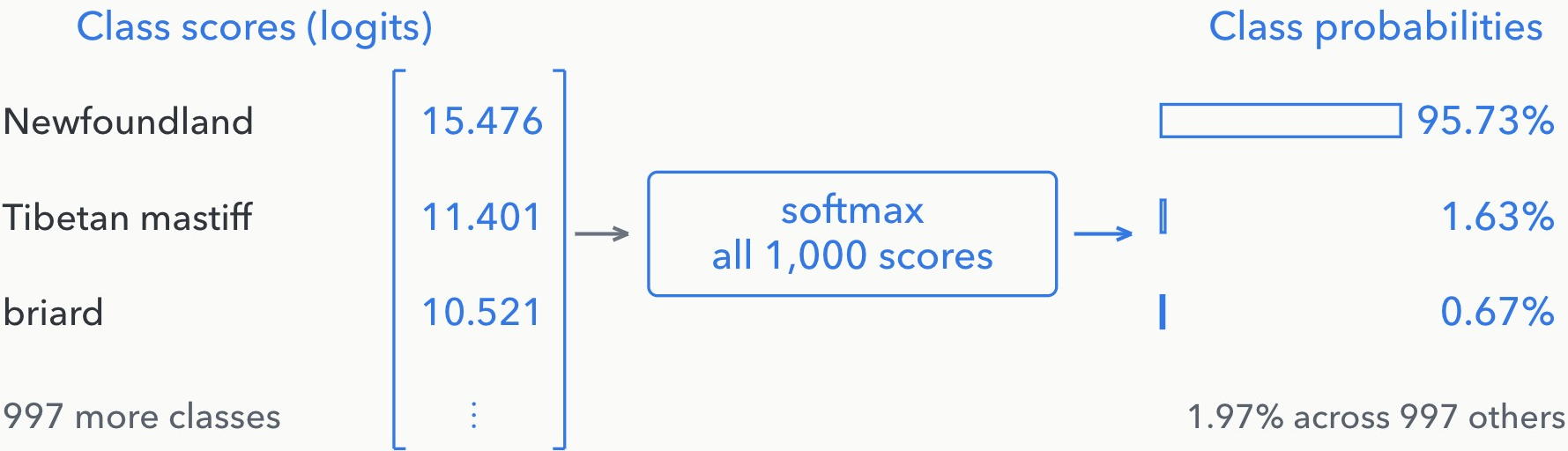
Multiply each CLS feature by this class's learned weight. Sum all 192 products and add its bias. For this dog, the Newfoundland neuron produces 15.4763. Every other class performs the same calculation with its own weights and bias.

Turn all 1,000 scores into class probabilities

Model route · highlighted operation



The alternatives are class labels: normalize over all 1,000.

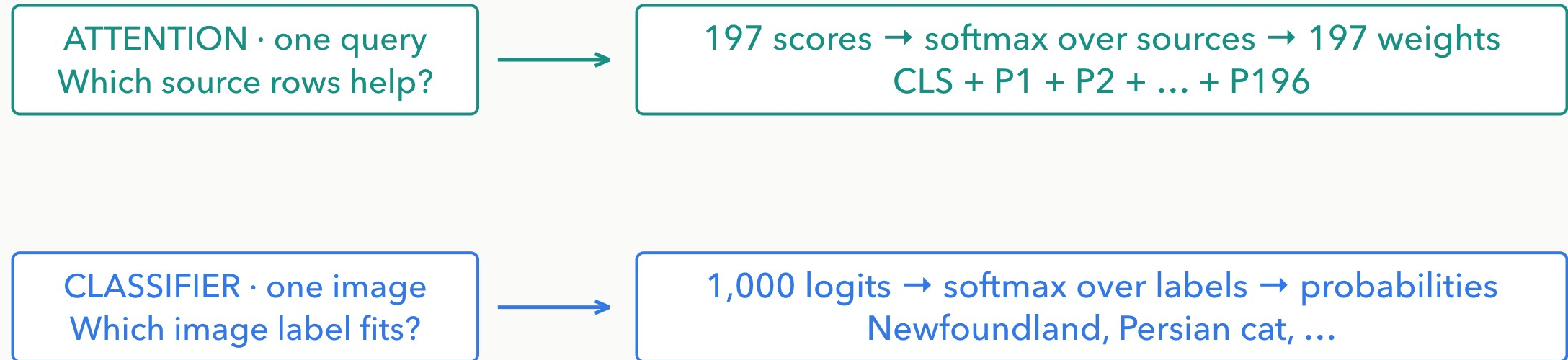


$$p(\text{Newfoundland}) = \exp(15.476) / \text{sum of exp(all 1,000 scores)}$$
$$\approx 0.9573 \cdot \text{All 1,000 probabilities sum to 1.}$$

Exponentiate each score and divide by the sum over all 1,000 classes. Newfoundland receives 95.73% probability. The displayed classes are only three alternatives; the remaining 997 also enter the denominator. Choose the class with the largest probability.

Two softmaxes, two different questions

The same normalization, applied to two different questions



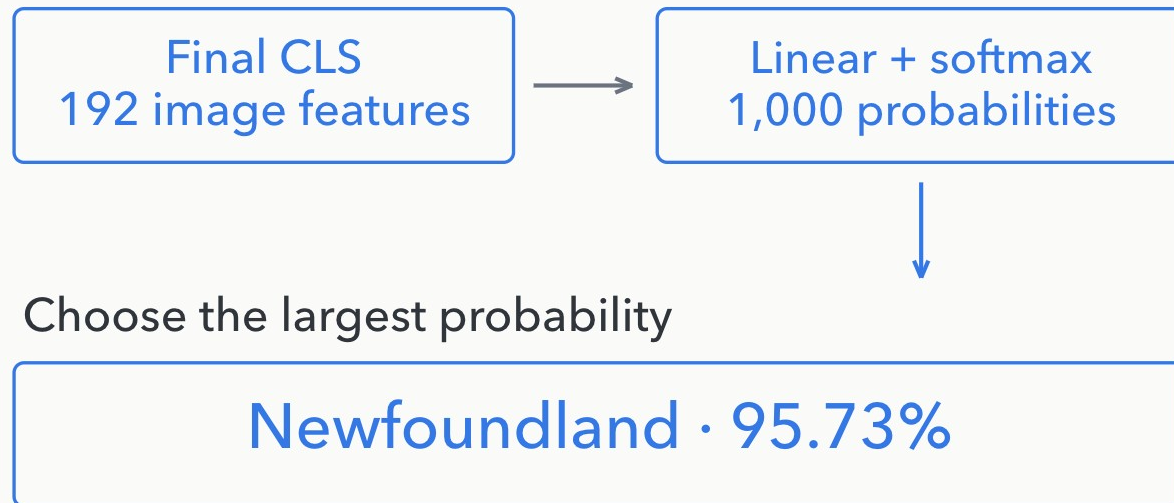
Attention normalizes across 197 source rows for each query and head. The classifier normalizes across 1,000 labels for each image. Both sum to one along their chosen axis; only the second distribution predicts the image label.

The same dog now has its final prediction

Model route · highlighted operation



The same photograph



Pixels → patches → contextual features → one image prediction.

The classifier reads the final CLS, scores 1,000 labels, and selects Newfoundland as the most probable class. Its probability for this photograph is 95.73%. We have completed one forward pass from pixels to an image prediction.

SECTION

04

Whole model walkthrough

We have opened each part of the classifier.

How do forward and backward fit together?

One example: follow the full model to a label loss, then reverse the path to compute gradients and update the parameters.

One shape trace from pixels to class scores

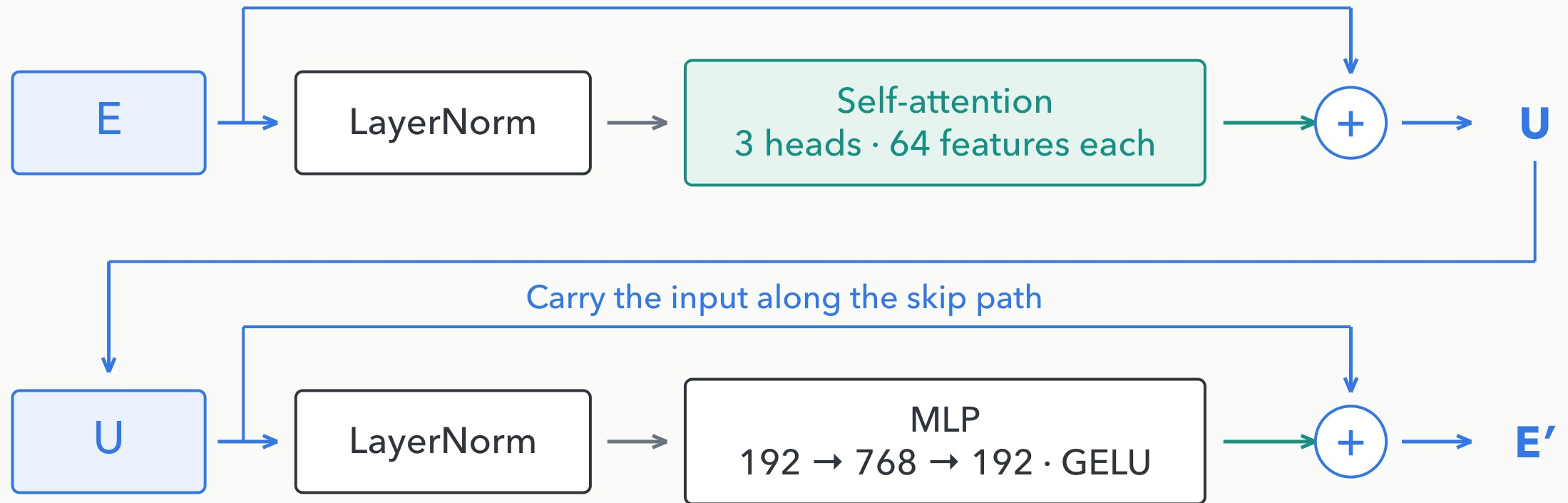
ONE IMAGE · batch axis omitted



Blocks preserve the shape. They change what each row represents.

Inside each block: mix, transform, keep the residual

ONE PRE-LAYERNORM BLOCK · input and output: 197×192

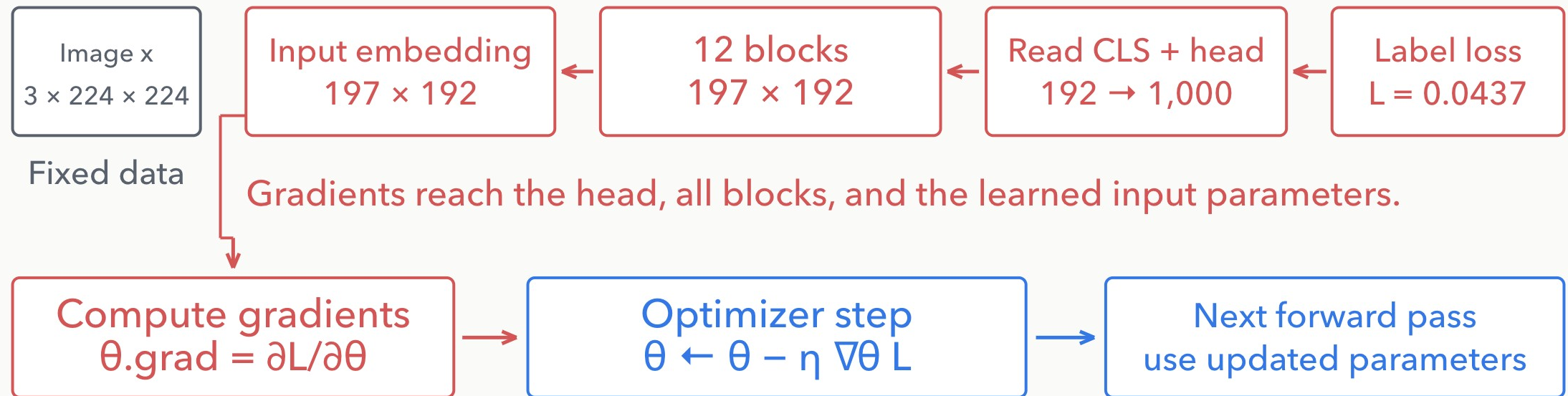


Attention mixes rows. The MLP transforms each row. Both add a residual update.

Repeat this structure 12 times, with different learned weights in each block. All 197 rows are updated in parallel.

Backward: compute gradients, then update the model

Same model, reverse direction: loss $\rightarrow$ parameter gradients



`backward()` computes gradients; `step()` changes the parameters.

Reverse the forward dependencies to compute gradients for the trainable parameters. The optimizer uses these gradients to update them. The next forward pass uses the updated model. The image and its label remain fixed.

SECTION

05

CNNs, ViTs and inductive bias

Attention lets patches exchange information.

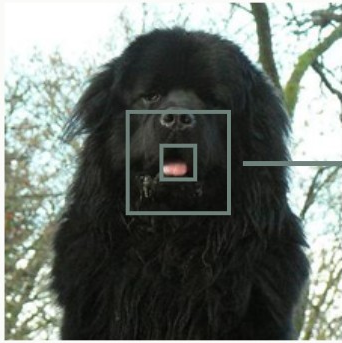
How do they differ, and which should we try?

Compare context, mixing, wider views and readouts, then connect those choices to inductive bias and practical data and compute constraints.

Two ways to build an image representation

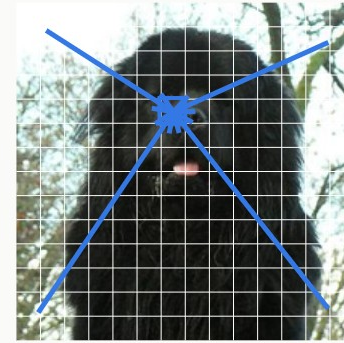
1 · Gather context: which inputs can interact in one layer?

CNN · local filters



One local neighbourhood
feeds one output location.
Start with nearby clues.

ViT · global attention



One query can use
every source patch.
Near and far are available.

2 · Mix information: what determines each contribution?

Learned filter

→ Reuse the same weights
at every image location.

Query + keys

→ Compute weights
for this query + image.

Both learn parameters. ViT recomputes attention weights from the current features.

First compare the available connections. Then press Next to compare the mixing weights. CNN filters stay fixed during a forward pass. ViT projection parameters also stay fixed, while attention weights are computed from the current query and source features.

A wider view, then one image label

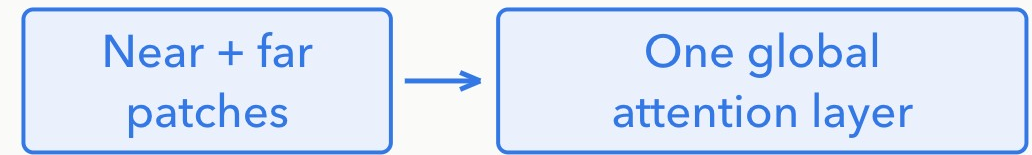
3 · Build a wider view: how do distant clues meet?

CNN · local filters



Successive layers connect larger regions.

ViT · global attention



Later blocks refine those relationships.

4 · Read out a label: turn many locations into one vector.

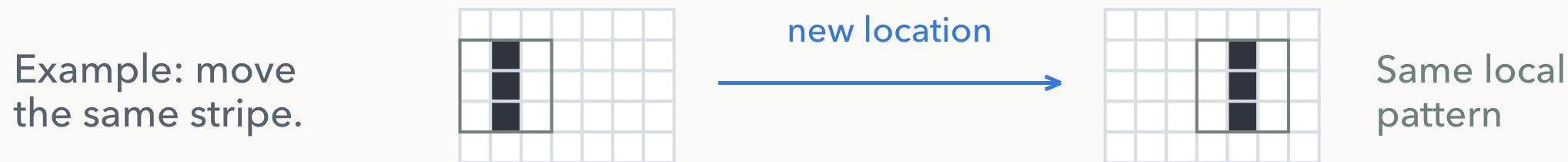


Both produce an image-level summary. Pooling is also a valid ViT readout.

Trace context through the layers, then press Next to inspect the readout. Both models can use the whole image. A classifier needs one image-level vector: CNNs often use spatial pooling; this ViT reads final CLS.

Inductive bias: a useful starting assumption

Inductive bias = assumptions built into how the model learns.



What is built in?	CNN	ViT · global attention
Spatial structure	Local filters, reused across the image	Patches + position; flexible global mixing
The shifted stripe	The same filter can detect it elsewhere	Position can change the attention pattern
Learned from data	Which local filters help	Which patch relations help

A CNN builds in local reuse: one learned detector can look for a pattern at many locations. ViT also has structure—patches, shared projections and position signals—but gives attention more freedom to learn spatial relationships.

Which is a sensible starting point?

Practical starting points · validate on your task and hardware.

Your situation	CNN	ViT · global attention
 Few labels; train from scratch	Useful baseline: built-in local reuse	Pay close attention to the training recipe
 Few labels; pretrained weights	Fine-tune a suitable pretrained CNN	Fine-tune a suitable pretrained ViT
 Phone or tight latency budget	Try a compact CNN; measure on the device	Try an efficient ViT; measure on the device
 Distant clues across the image	Use enough depth for broad context	Global attention links distant patches directly

Compare validation accuracy, speed and memory under the same budget.

Treat these as starting points, not a ranking. Pretraining can matter more than the architecture name on a small dataset. Compare suitable models using the same validation set and measure speed and memory on the target hardware.

IMPLEMENTATION LAB · OPTIONAL

Build the same ViT in PyTorch



Use this lab in class, or work through it after the conceptual lecture.

All implementation details are retained. Every operation is paired with shapes, diagrams or a numerical equivalence check.

Keep the same photograph and add the batch axis

```
# rgb: (224, 224, 3), RGB per pixel
x = rgb.permute(2, 0, 1) # 3, 224, 224
x = x.unsqueeze(0)       # 1, 3, 224, 224
# Patch order: all R, then G, then B
```



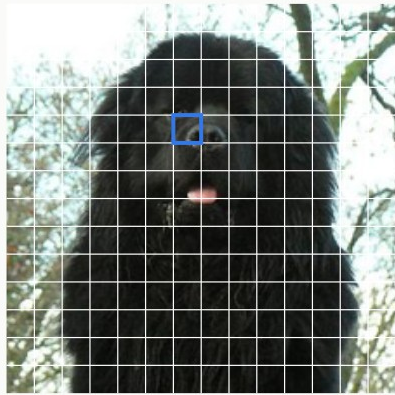
Height × width × RGB

PyTorch: $B \times C \times H \times W$
 $B \times 3 \times 224 \times 224$

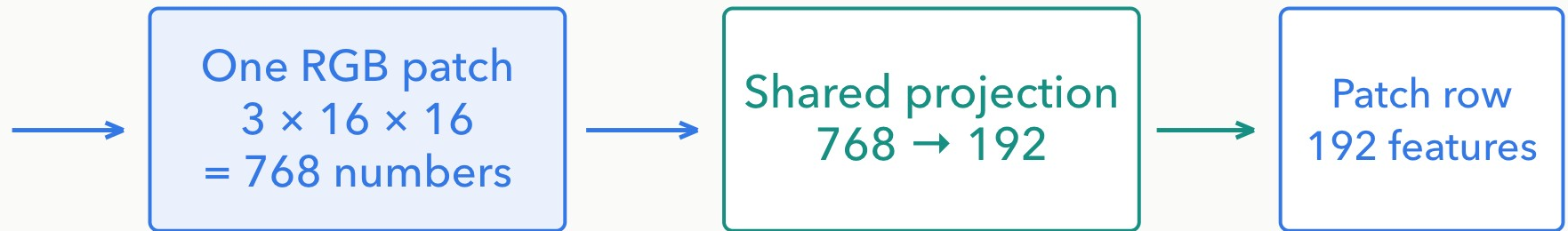
The prepared photograph is 224×224 with three RGB channels. PyTorch puts channels before height and width, then adds a batch axis. B counts images; it does not count patches.

Our target: turn every patch into 192 features

Implement the same patch projection we used for the dog.



$224 \times 224 \times 3$
196 patches



Apply the same learned weights and biases at every location.

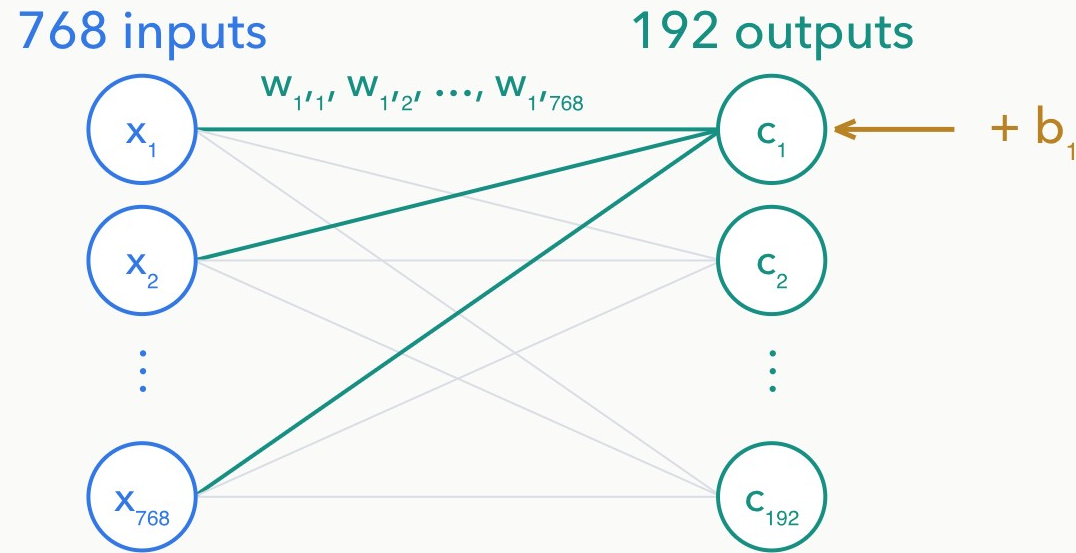
One image → 196 rows, each with 192 features.

Two implementations: flatten + Linear or Conv2d

Each RGB patch supplies 768 numbers. One shared projection turns it into 192 features. Reusing that projection across the 14×14 grid produces 196 rows. We will implement this exact operation in two ways.

See the patch projection as a layer of neurons

One affine layer: `nn.Linear(768, 192, bias=True)`



Open just the first output neuron

$$c_1 = w_{1'1}x_1 + w_{1'2}x_2 + \dots + w_{1'768}x_{768} + b_1$$

Every output has its own
768 weights + 1 bias.

192 × 768 weights + 192 biases = 147,648 parameters

Patch projection: no hidden layer or activation. Later MLP: Linear → GELU → Linear.

All 768 inputs connect to every output neuron. Each neuron computes a weighted sum and adds its own bias. The patch projection is one dense layer; the later Transformer MLP has two layers and a nonlinear activation.

Implementation 1: extract patches, then use Linear

```
linear = nn.Linear(768, 192, bias=True)

def project_with_linear(images, linear):
    patches = F.unfold(images, kernel_size=16, stride=16)
    # B x 768 x 196; each patch: all R, then G, then B
    # Within each channel: left to right, top to bottom
    patches = patches.transpose(1, 2)
    # B x 196 x 768: one flattened patch per row
    return linear(patches) # B x 196 x 192
```

One patch row: 256 R values | 256 G values | 256 B values.

Unfold extracts each non-overlapping patch as a column. Transpose makes patches into rows. Linear changes only the last axis, from 768 inputs to 192 features, reusing its weights for every patch and image.

images · B × 3 × 224 × 224



unfold: flattened columns
B × 768 × 196



transpose: patch rows
B × 196 × 768

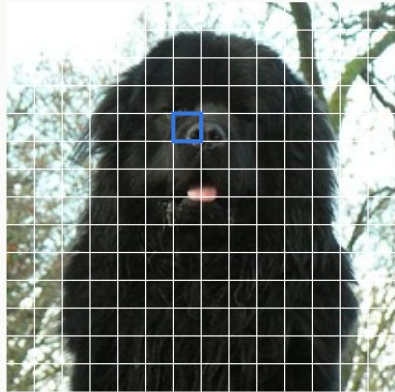


Linear: project each row
B × 196 × 192

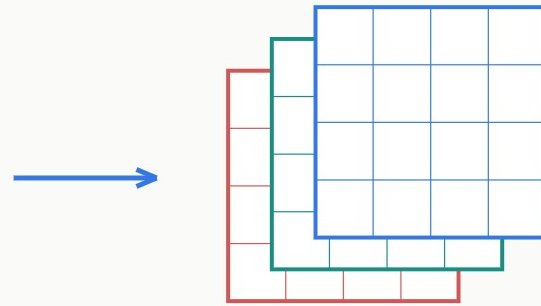
Implementation 2: the same projection with Conv2d

```
conv = nn.Conv2d(3, 192, kernel_size=16, stride=16,  
padding=0, bias=True)
```

One filter = one output neuron



14 × 14 locations



3 × 16 × 16 weights
all 768 pixels contribute

+ b_j

192 filters → 192 features
at each patch location

Stride 16: move by one patch.

Output: $B \times 192 \times 14 \times 14$

Same filters at every location. Kernel size = stride = 16; patches do not overlap.

Reshape each neuron's 768 weights into a $3 \times 16 \times 16$ filter. It reads the same pixels and adds the same bias. Conv2d applies all 192 filters at every patch location, giving a 192-channel feature grid.

Reshape the weights; keep the same parameter count

Identical numbers, stored with different shapes

Parameter	Linear	Conv2d
Weight shape	192×768	$192 \times 3 \times 16 \times 16$
Weights	147,456	147,456
Biases	192	192
Total	147,648	147,648

Match the input order: each filter flattens R, then G, then B.

```
with torch.no_grad():  
    linear.weight.copy_(conv.weight.flatten(1))  
    linear.bias.copy_(conv.bias)
```

Both implementations have 147,456 weights and 192 biases. Copying the exact weights and biases makes their computations equivalent. More patches or images create more output values, while the learned parameter count stays 147,648.

Same pixel × same weight, in both implementations

Dog patch P63 · first output feature · indices below start at 0

Flatten order: $R_0 \dots R_{255} \mid G_0 \dots G_{255} \mid B_0 \dots B_{255}$

Green pixel at (row 2, column 3) → flat index $256 + 2 \times 16 + 3 = 291$

Linear: one neuron

same 768 pixels

R_0	R_1	...	G_{35}	...	B_{255}
-------	-------	-----	----------	-----	-----------

$\times$ $\times$... $\times$... $\times$
 $W[0,291]$ selected

w_0	w_1	...	w_{291}	...	w_{767}
-------	-------	-----	-----------	-----	-----------

Conv2d: one filter, flattened

same 768 pixels

R_0	R_1	...	G_{35}	...	B_{255}
-------	-------	-----	----------	-----	-----------

$\times$ $\times$... $\times$... $\times$
 $W[0,1,2,3]$ selected

w_0	w_1	...	w_{291}	...	w_{767}
-------	-------	-----	-----------	-----	-----------

Selected product in both: $-0.623529 \times 0.019536 = -0.012181$

Add all 768 products + the same bias (-0.414880) → -0.851541

Both paths pair the same 768 pixels with the same 768 weights, sum their products, and add the same bias.

Verify it on both photographs: the features match

Same pretrained weights · same prepared dog and cat images

		P63 · first 3 of 192 features	Linear	Conv2d
	Dog	768 → 192	$[-0.852, 1.339, 0.504, \dots]$	$[-0.852, 1.339, 0.504, \dots]$
	Cat	768 → 192	$[-0.189, 0.104, -0.208, \dots]$	$[-0.189, 0.104, -0.208, \dots]$

Checked all $2 \times 196 \times 192 = 75,264$ features.

Largest float32 difference: 5.2e-06. Same result within rounding.

Both produce $B \times 196 \times 192$ rows for the same classifier.

Using identical pretrained weights, both paths produce the same dog and cat features within floating-point rounding. Conv2d packages patch extraction and projection into one image operation. Flatten and transpose its grid before adding CLS and positions.

Why package patch projection as Conv2d?

One image operation applies all 192 filters at all 196 locations.

Linear route



Conv2d route



Less reshaping code; optimized convolution backends. Same learned model.

Conv2d accepts the image layout directly and handles the repeated patch computations. It avoids an explicit unfolded patch tensor in our code and uses optimized convolution backends. Actual speed and memory depend on the hardware.

First turn the feature grid into patch rows

```
# x: (B, 3, 224, 224)
B = x.shape[0]
grid = self.patch(x)
rows = grid.flatten(2)
rows = rows.transpose(1, 2)
```

integer: number of images
(B, 192, 14, 14)
(B, 192, 196)
(B, 196, 192)



Flatten combines the 14×14 spatial grid into 196 locations. Transpose puts the 192 features last. Each row now describes one patch.

Then prepend CLS and add position

```
# Continue with rows: (B, 196, 192)
cls = self.cls.expand(B, -1, -1)          # (B, 1, 192)
rows = torch.cat([cls, rows], dim=1)      # (B, 197, 192)
return rows + self.pos                    # (B, 197, 192)

# self.cls: (1, 1, 192)    self.pos: (1, 197, 192)
```



Expand reuses the learned CLS start across the batch. Concatenation adds one row. Position addition changes the numbers while keeping 197 rows and 192 features.

Make queries, keys and values for three heads

```
B, N, D = x.shape                                # B, 197, 192
qkv = self.qkv(x)                                # B, 197, 576
qkv = qkv.reshape(B, N, 3, 3, 64)                # B, N, QKV, heads, features
q, k, v = qkv.permute(2, 0, 3, 1, 4).unbind(0)
# q, k, v: each (B, 3, 197, 64)
```



LOCAL ROLES

Q / receiver

K / source

V / message

Project each row once to produce all queries, keys and values. Split the 576 outputs into three roles, each with three heads of width 64.

Compute one message for every query in every head

```
scores = (q @ k.transpose(-2, -1)) / 8 # B, 3, 197, 197
weights = scores.softmax(dim=-1) # B, 3, 197, 197
messages = weights @ v # B, 3, 197, 64
```



For each image and head, every query scores all 197 sources. Softmax normalizes each query row. Multiplying by V combines the sources into one 64-feature message per query.

Join the head messages and project back to 192

```
joined = messages.transpose(1, 2)      # B, 197, 3, 64
joined = joined.reshape(B, N, D)       # B, 197, 192
return self.proj(joined)               # B, 197, 192
```



LOCAL ROLES

Q / receiver

K / source

V / message

Move the head axis beside its features, then join 3×64 into 192. A learned output projection mixes these features before the residual addition.

Build the layers inside one Transformer block

```
self.norm1 = nn.LayerNorm(192, eps=1e-6)
self.attn = Attention()
self.norm2 = nn.LayerNorm(192, eps=1e-6)
self.mlp = nn.Sequential(
    nn.Linear(192, 768), nn.GELU(),
    nn.Linear(768, 192))
```

Attention
mix information across rows



MLP
192 → 768 → 192

Create two normalization layers, one attention module and one MLP. The MLP expands and then restores the feature width for each row.

Use the two residual paths in order

```
# x: (B, 197, 192)
x = x + self.attn(self.norm1(x))
x = x + self.mlp(self.norm2(x))
return x
```



Normalize, compute the attention update, and add the input. Then normalize the updated rows, compute the MLP update, and add them again.

Create twelve blocks with separate learned parameters

```
self.blocks = nn.ModuleList([Block() for _ in range(12)])  
self.norm = nn.LayerNorm(192, eps=1e-6)  
self.head = nn.Linear(192, num_classes)
```



ModuleList creates twelve distinct blocks. The final normalization and class head read the result of the entire stack.

Run the stack, then read the final CLS

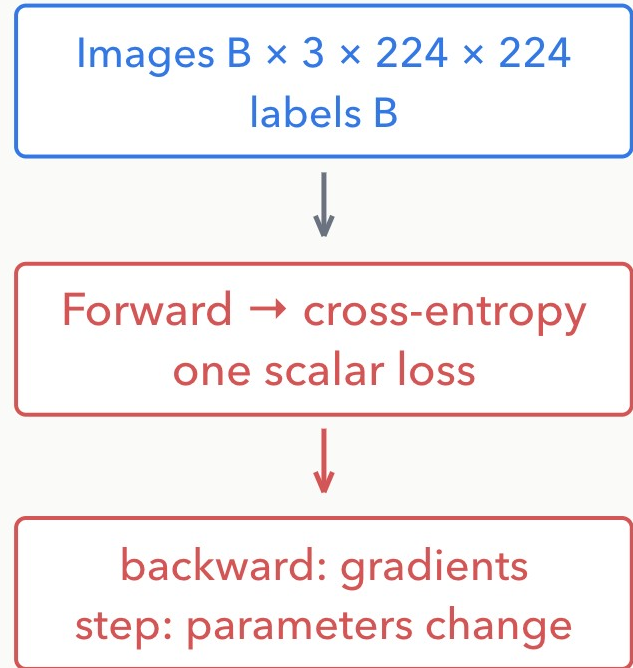
```
rows = self.embed(x)                # B, 197, 192
for block in self.blocks:
    rows = block(rows)              # B, 197, 192
summary = self.norm(rows)[: , 0]    # B, 192: final CLS
return self.head(summary)           # B, 1000 logits
```



Feed each block's output into the next. Normalize the final rows and select CLS for each image. The class head returns 1,000 scores.

Connect the loss diagram to one training step

```
model.train()  
optimizer.zero_grad(set_to_none=True)  
logits = model(images)           # B, 1000  
loss = F.cross_entropy(logits, labels)  # labels: B  
loss.backward()  
optimizer.step()
```



Cross-entropy consumes logits and class-index labels. Backward computes gradients; the optimizer updates the parameters it owns. Clear previous gradients before the next batch. This code shows the procedure without claiming a trained result.

Optional extensions

OPTIONAL EXTENSIONS

Transfer · evaluation · interpretation

These experiments extend the core ImageNet classification story.

SECTION

07

Adapt and evaluate the classifier

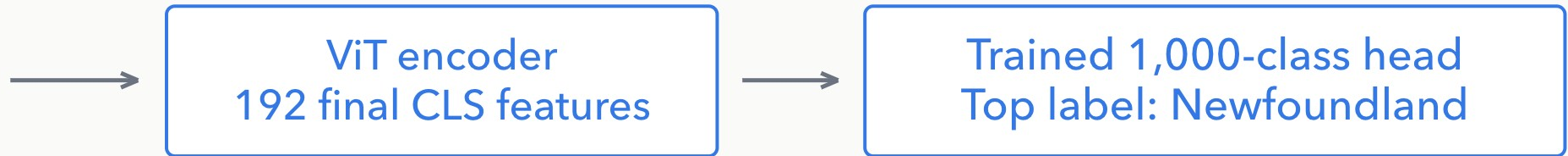
Our checkpoint predicts 1,000 ImageNet labels.

What if our labels or image domain change?

Choose trainable parameters, follow one batch, then evaluate on held-out images. We will describe the procedure without running a new experiment.

What was this model trained to predict?

The checkpoint already learned from labelled photographs.



Our saved dog prediction used this existing task and head.

The checkpoint was pretrained on ImageNet-21k and fine-tuned on ImageNet-1k. Its current head predicts 1,000 categories. Newfoundland is one of those labels.

Same photographs, a different label vocabulary

Now our application asks a simpler question: dog or cat?



Old label vocabulary
Newfoundland



Our new target
dog



Old label vocabulary
Persian cat



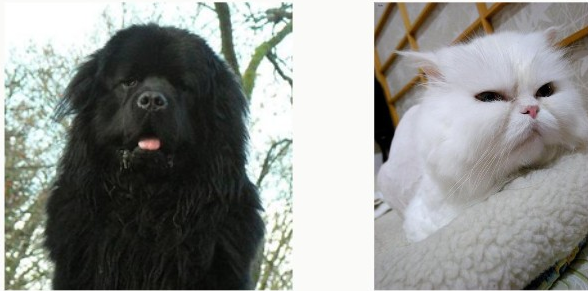
Our new target
cat

Our pet application groups breeds into two categories. We want two scores for every image, trained and evaluated for this dog/cat task. Start by reusing the encoder's visual features.

What if our users supply sketches?

A second kind of change: same labels, different-looking inputs.

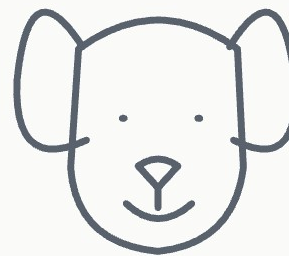
Pet photographs



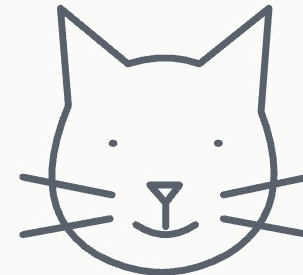
New domain



Sketches · schematic examples



dog



cat

Fur texture and colour disappear; outlines become more important.

Keep two outputs. Test on sketches; use labelled sketches to adapt if needed.

The labels can stay dog and cat while the image domain changes. A photo classifier may struggle with sketches. Evaluate on the target domain, then compare head training with encoder fine-tuning.

Replace the ImageNet head with our two-class head



pretrained ViT encoder
final CLS: 192 features

new nn.Linear(192, 2)
scores: [dog, cat]

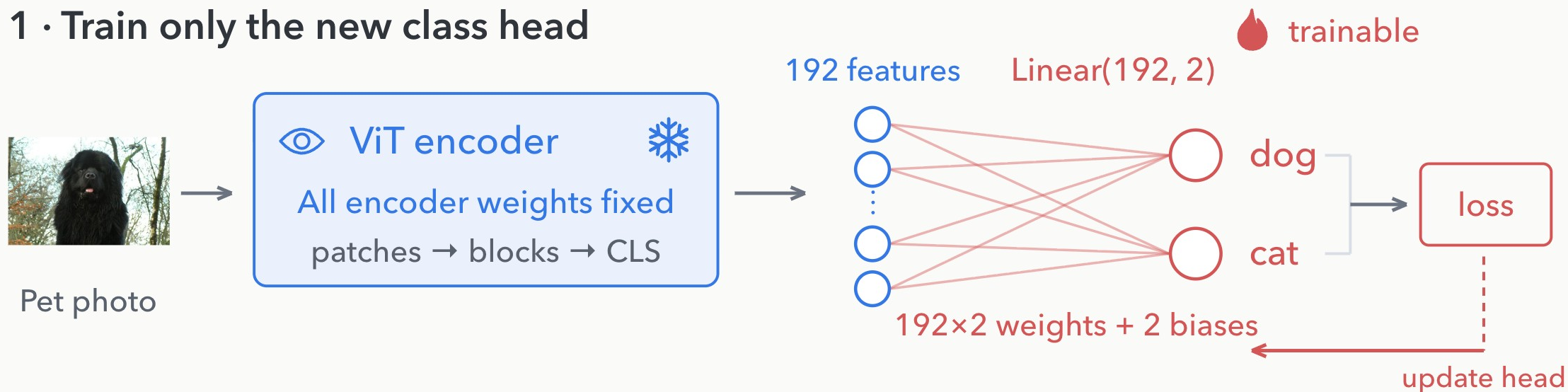
384 weights + 2 biases = 386 parameters

Keep the learned visual features to begin with.

For our dog/cat task, replace the ImageNet head with a new two-output linear layer. The encoder supplies a 192-number image representation. The new 386 head parameters must learn from labelled pet images.

Freeze the encoder; train the new head

1 · Train only the new class head



Frozen encoder: every image still gets its own 192 features.

Only the 386 head parameters receive optimizer updates.

Snowflake: weights stay fixed. Flame: weights can learn. Train the two-class head on labelled pet images. The frozen encoder still computes different features for different images.

Next option: fine-tune the last block as well

If head-only training is insufficient, adapt some features too.

2 · Optional:
fine-tune the
last block too

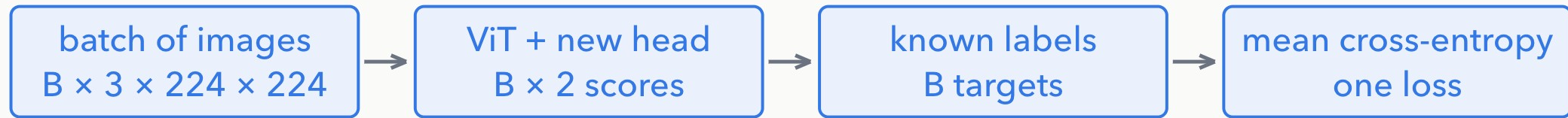


The loss can now change the last block and the class head.

Compare on validation data before choosing how much to unfreeze.

Unfreeze the last block and train it together with the head. Earlier encoder weights stay fixed. This lets some visual features adapt to the target data; validation determines whether it helps.

One batch follows the same forward and backward paths

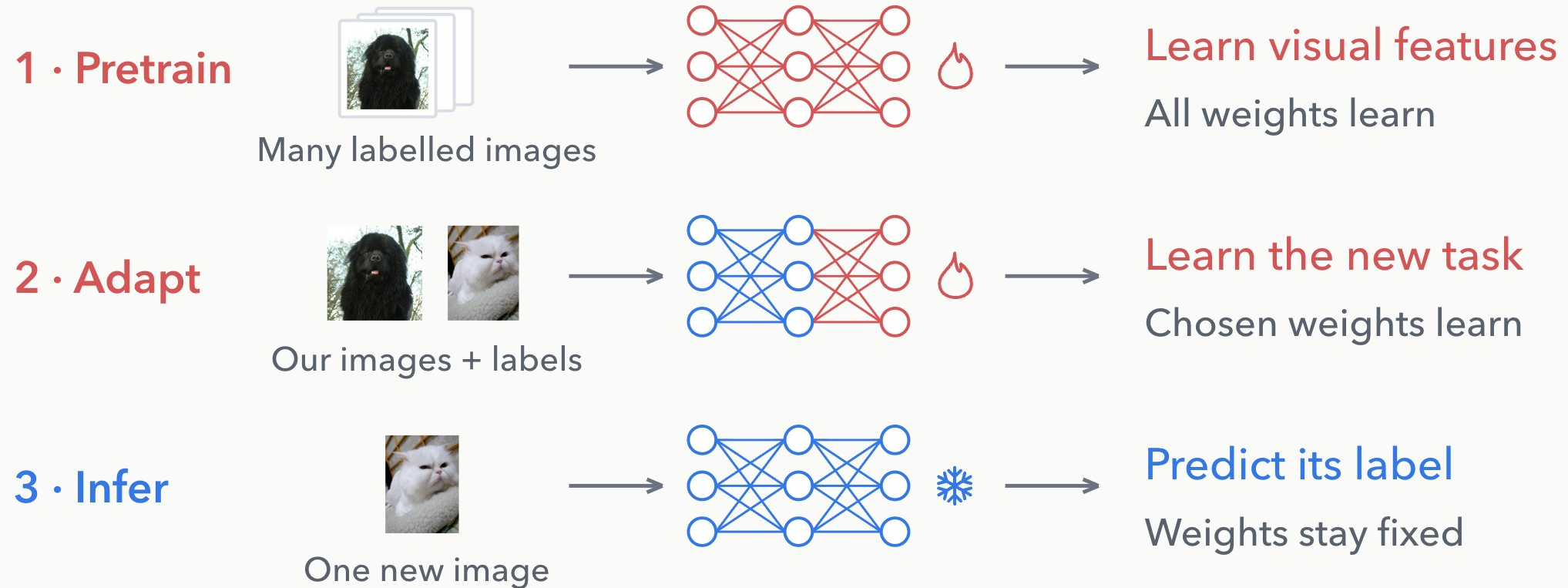


zero gradients → forward → loss → backward → optimizer step

Repeat on training batches. Evaluate separately with weights fixed.

Pair every image with its dog/cat label. Compute the average loss for the batch, backpropagate once, and update the trainable parameters. Repeat this procedure over training batches.

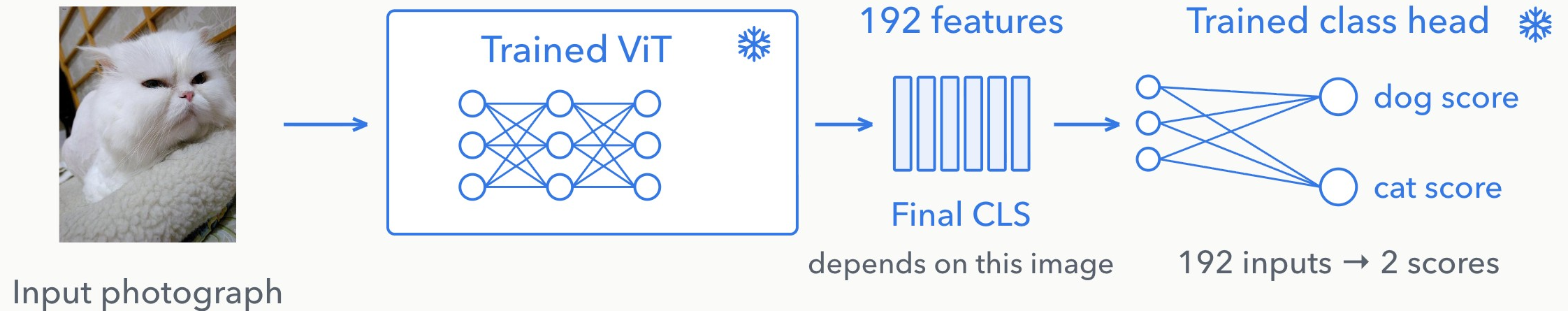
Three stages: learn, adapt, then predict



Orange connections and flames mark learning. Blue connections and snowflakes mark fixed parameters. During inference, a new image changes the features while the stored weights stay fixed.

What happens when we classify a new photograph?

One forward pass through the trained model



Softmax → two probabilities → choose the larger one.

The image produces its own CLS features and class scores. Both snowflakes mark fixed weights. Inference computes a prediction without a label, loss or optimizer update.

How would we check whether the classifier learned?

Before fitting: separate training, validation and test images



Inspect: dog → dog dog → cat cat → dog cat → cat

$\text{Accuracy} = \text{correct predictions} / \text{number of test images}$

Keep mistakes beside correct examples. A confident score can still be wrong.

Use held-out images to measure accuracy and the two kinds of confusion. Inspect successes and mistakes, including confident errors. A probability on one photograph is not the classifier's test accuracy.

SECTION

08

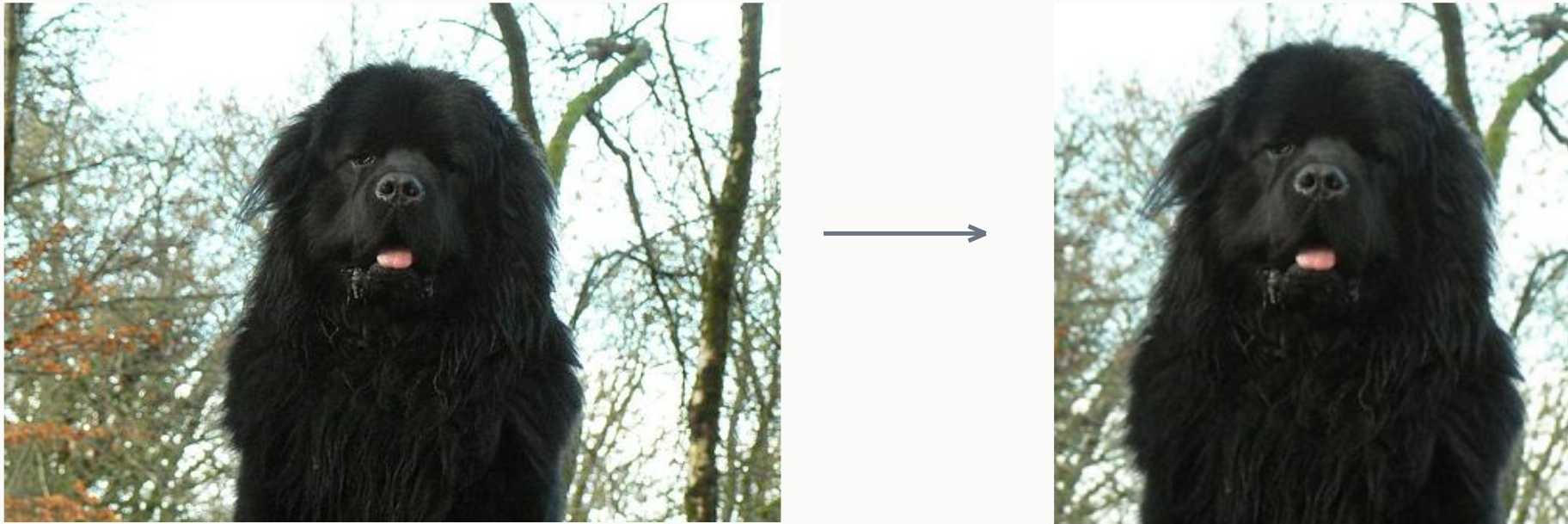
Return to the real photographs

Training and inference have different jobs.

What do the saved photo predictions establish?

Return to the saved dog and cat predictions. These outputs come from the ImageNet checkpoint, not from a newly trained pet classifier.

Which pixels are we giving the real model?



original photograph → supplied resize / crop → normalize RGB

The model receives the square crop on the right, after RGB normalization.

What did the model call our dog?



Top 3 of 1,000 labels

Newfoundland 95.73%



Tibetan mastiff 1.63%



briard 0.67%



These probabilities come from running this photograph through the pretrained model.

Does one correct photograph tell us the accuracy?

probability of the known breed: 0.957

one successful prediction

How often does that happen on new photos?

To estimate accuracy, count correct predictions over an appropriate test set.

What happens when we give it the cat?



Same checkpoint. Another photograph.

Persian cat	96.71%
-------------	--------

Angora	1.26%
--------	-------

plastic bag	0.43%
-------------	-------

Keep the same model and preprocessing. Change only the photograph.

SECTION

09

Look inside the trained model

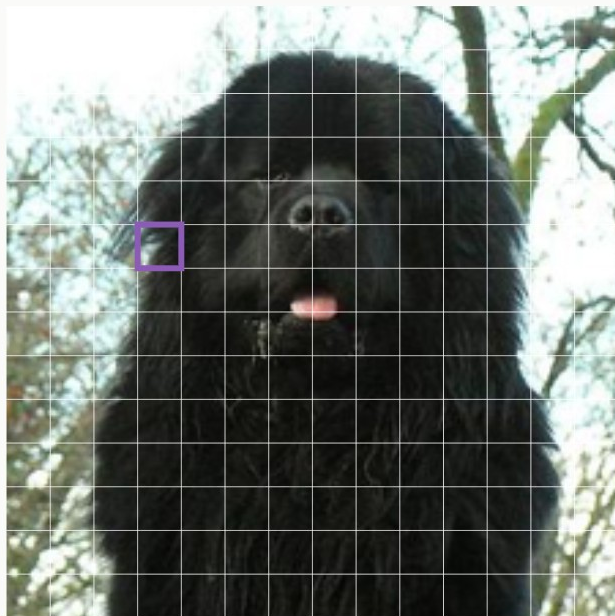
A prediction tells us the model's answer.

What can we measure inside its attention blocks?

Read attention maps, then cover parts of the photograph and measure how the prediction changes.

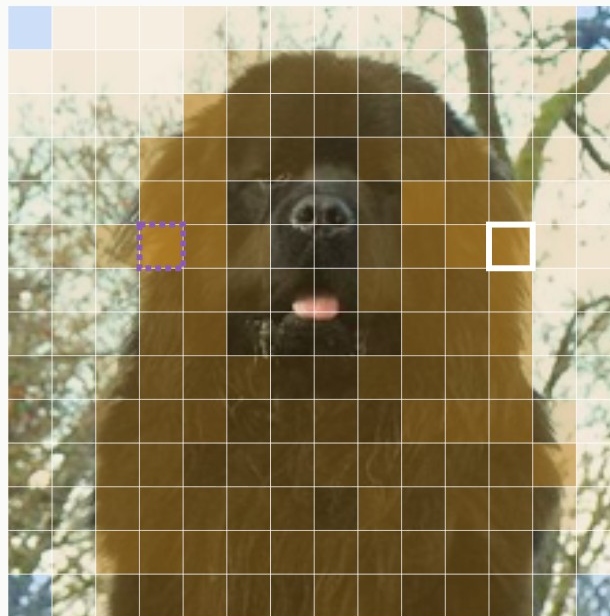
Similar patch features can connect distant image regions

Query · P74



Violet outline: selected patch

Feature similarity · block 12



Gold: positive cosine · fixed -1 to 1

Similar features across the dog

P74 ↔ P82: 0.902

Compare the final
192-number patch vectors.

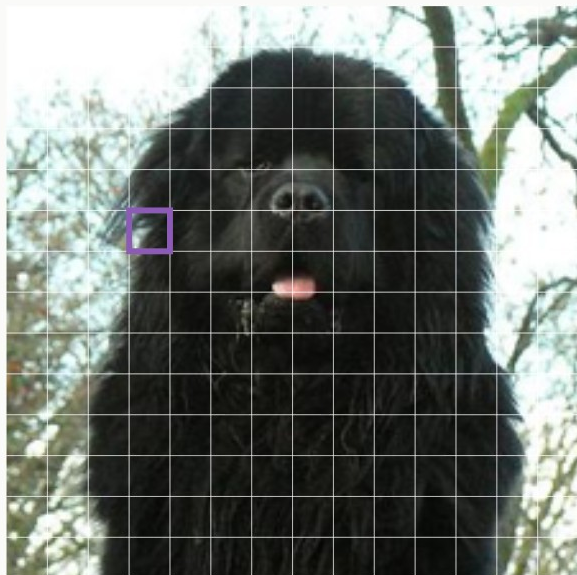
Feature similarity ≠ attention weight

[Explore all nine examples ↗](#)

The selected ear-side patch matches patches on the other side of the dog. This measures representation similarity, rather than which values attention mixes.

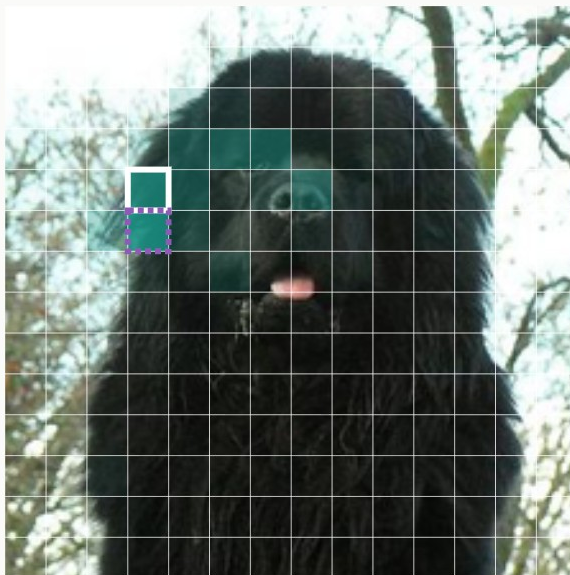
Keep the query fixed; change only the attention head

Query · P74



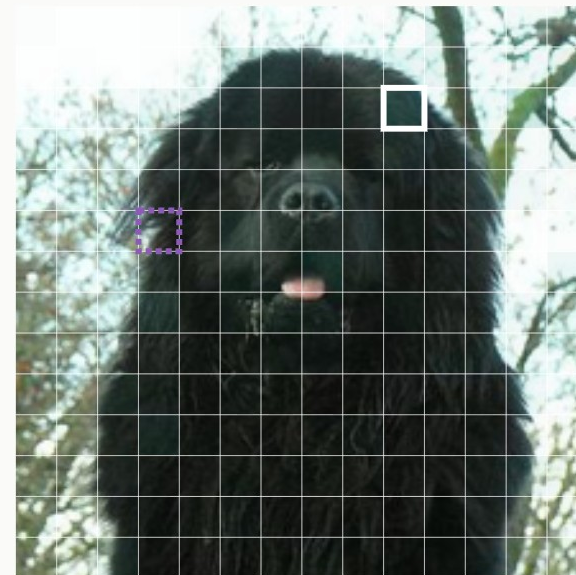
Same query · block 4

Head 1



P60: 9.09%

Head 2



P38: 2.29%

Different heads gather different mixtures.

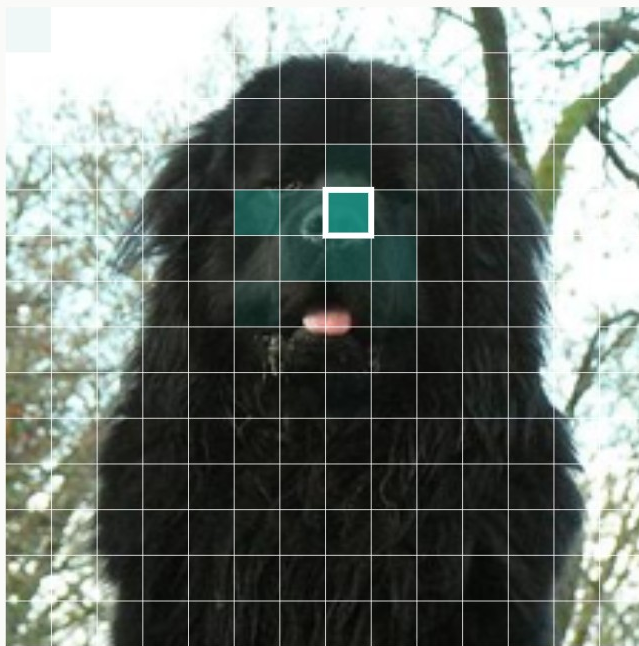
[Explore all nine examples ↗](#)

Teal shows source weights. Both maps use the same 0-10.35% scale. Within each head, all 197 source weights—including CLS—sum to 100%.

CLS gathers a message for the image summary

Attention · block 12 · head 1

Query = CLS
current image state



**CLS reads
patch information**

P64: 24.86%

of this head's weight

Teal: 0 → 24.86%

One attention head is one part of the classifier.

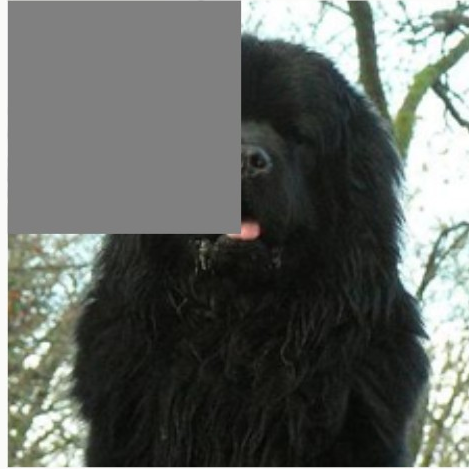
[Explore all nine examples ↗](#)

The query is CLS. Its weights select a mixture of all source value rows. Later operations and the classifier turn the final CLS features into class scores.

"Cover" means replace these pixels with gray



Original input x



Copy with gray pixels

112 × 112 pixels replaced

RGB fill = (0.5, 0.5, 0.5)

After normalization:

$$(0.5 - 0.5) / 0.5 = 0$$

Same 224 × 224 image size.

All 196 patches remain.

```
x = transform(photo).unsqueeze(0) # (1, 3, 224, 224)
covered = x.clone() # a separate copy
covered[:, :, :112, :112] = 0 # all RGB channels, top left
```

Copy the normalized input, then overwrite one quadrant. The rest of the image stays unchanged; no patch rows are deleted.

Run the covered image through the same trained model

Two independent forward passes



Original x



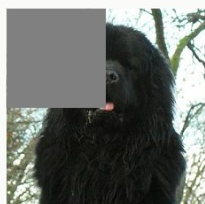
Same trained ViT
patches → blocks → CLS → head

Same weights; recompute all activations



95.73%

P(Newfoundland)



Covered copy



Same trained ViT
patches → blocks → CLS → head



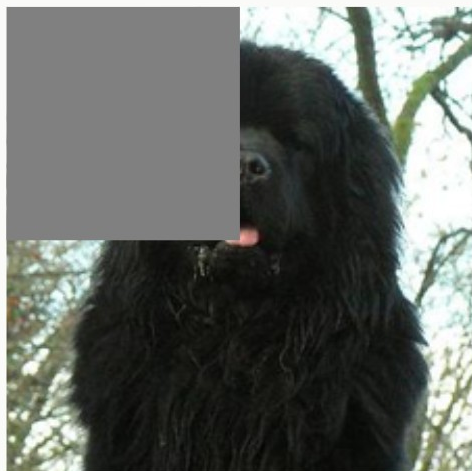
83.00%

```
model.eval()  
with torch.inference_mode():  
    p_before = model(x).softmax(-1)[0]          # (1000,)  
    p_after = model(covered).softmax(-1)[0]      # (1000,)  
    before, after = p_before[256], p_after[256]  # Newfoundland
```

Recompute patch features, attention, CLS and class scores. Softmax gives 1,000 probabilities; compare the Newfoundland entry in both runs. The weights stay fixed.

Four covers, four new forward passes

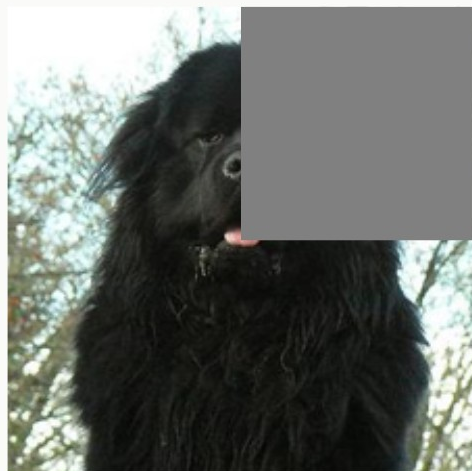
Original P(Newfoundland): 95.73%



Top left

83.00%

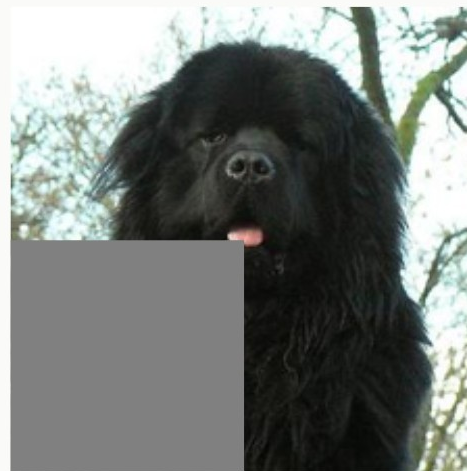
−12.73 points



Top right

79.62%

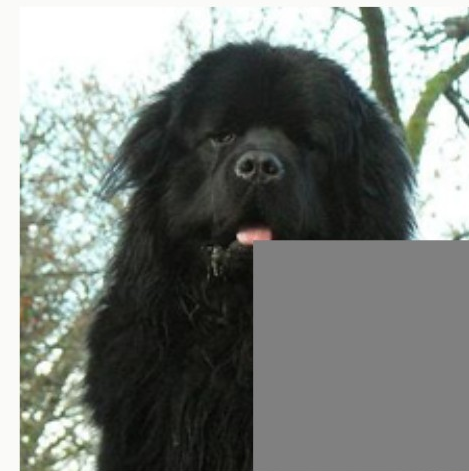
−16.11 points



Bottom left

84.54%

−11.19 points



Bottom right

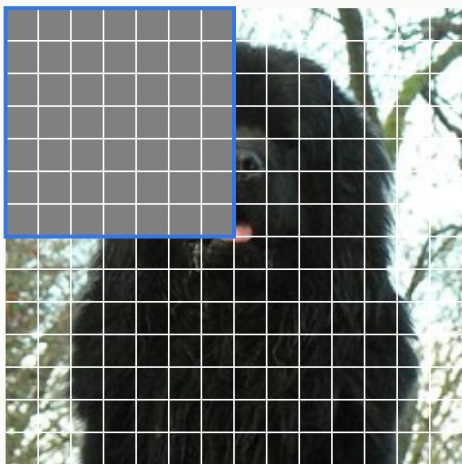
82.16%

−13.57 points

Top-right covering causes the largest drop: 16.11 percentage points. All four images still predict Newfoundland. Attention shows internal mixing; occlusion tests how a changed input changes the prediction.

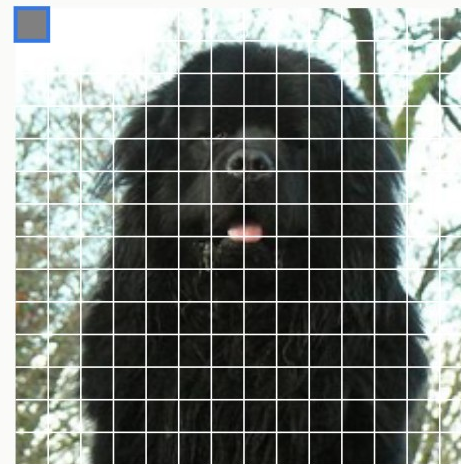
Use smaller covers to ask a more local question

Large cover: 112×112



49 patches covered · 4 tests

Small cover: 16×16



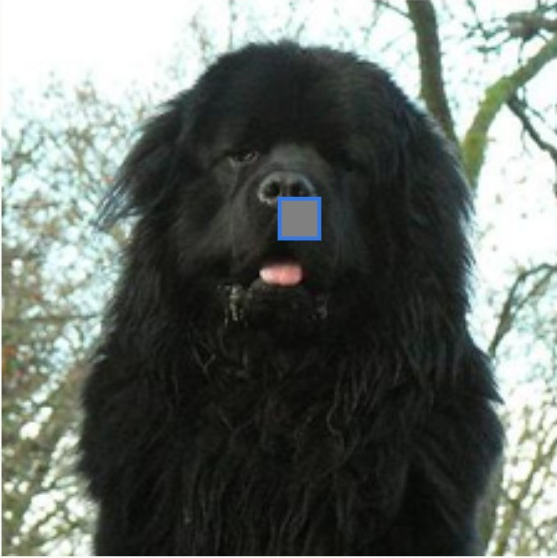
1 patch covered · 196 tests

Fresh original → cover one region → rerun → measure the drop

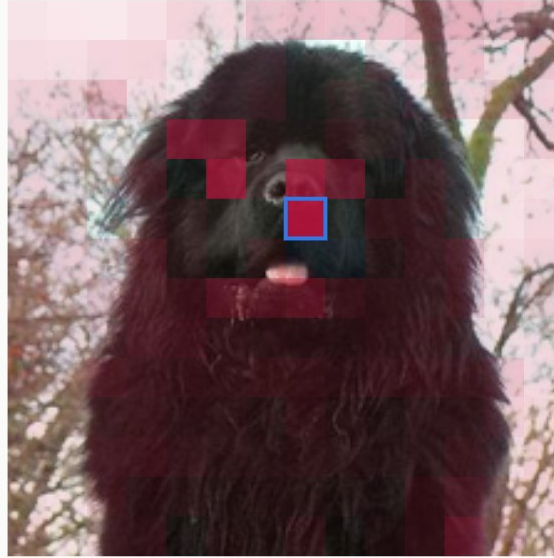
Only the cover size changes. The trained model still uses 16×16 patches and a 224×224 input. Every test starts from the original image.

Smaller covers reveal local sensitivity

One test: cover P78



All 196 test results



Largest drop: P78

95.73% → 91.89%

Down 3.84 percentage points

Red: probability falls

Blue: probability rises

Drop scale: -4 to +4 points

Each square = one separate test

All 196 tests still predict Newfoundland.

Smaller covers localize sensitivity. P78 causes the largest drop here, but no single-patch cover changes the top label. This map measures probability changes, not attention weights.

SECTION

10

The cost of smaller patches

Every query compares all source rows.

What happens when we make the patches smaller?

Halve the patch width and height. Predict the change in attention work before calculating it.

What changes when the patch size is halved?

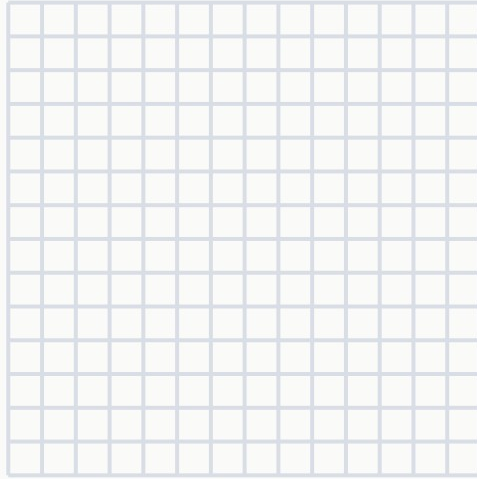
32 × 32 patches



49 patches + CLS

$50^2 = 2,500$ scores / head

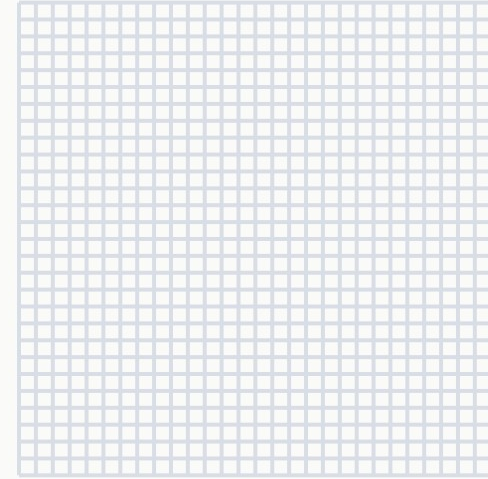
16 × 16 patches



196 patches + CLS

$197^2 = 38,809$ scores / head

8 × 8 patches



784 patches + CLS

$785^2 = 616,225$ scores / head

At fixed image size, half the patch width gives four times as many patch rows.

How much matching happens inside the tiny real model?

196 patch rows + CLS

$197 \times 197 = 38,809$ scores per head

3 heads $\times$ 12 blocks

1,397,124 scores for one image

Even this small ViT compares many pairs of rows.

What happens if we use a larger image?

image width / height

patch width / height

Predict the token count first.

Image size Patch size

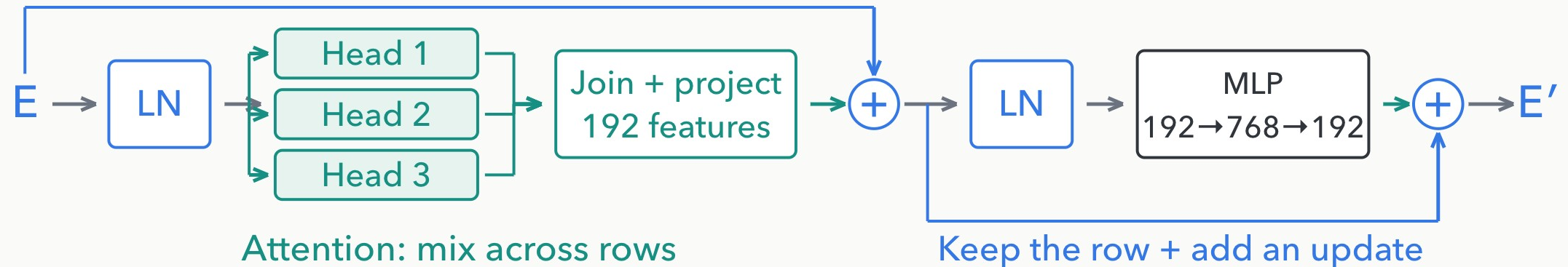
$14 \times 14 = 196$ patches $\cdot$ 197 tokens with CLS $\cdot$ 38,809 scores per head

Work out the token count before you change the controls.

The whole ViT: pixels $\rightarrow$ context $\rightarrow$ one label



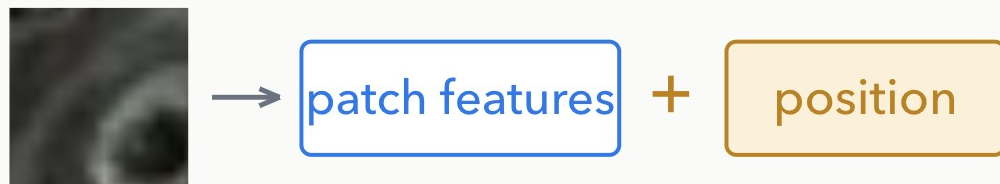
INSIDE EACH BLOCK



One shared patch layer builds the rows. Twelve blocks refine every row. The classifier reads final CLS to score image labels.

Four ideas to carry forward

1 • Content + location



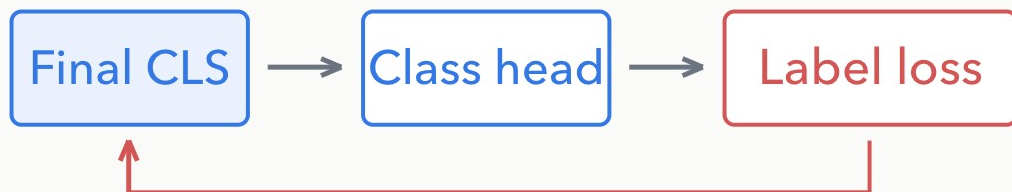
Shared projection; each slot has a position.

2 • Every row gains context



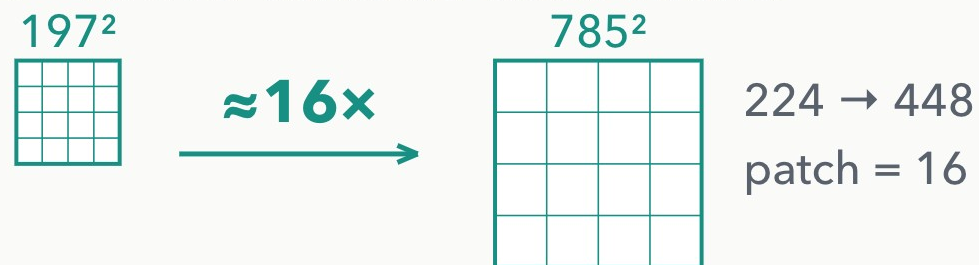
Attention mixes; the MLP transforms.

3 • The task teaches the summary



Train with loss; infer with fixed weights.

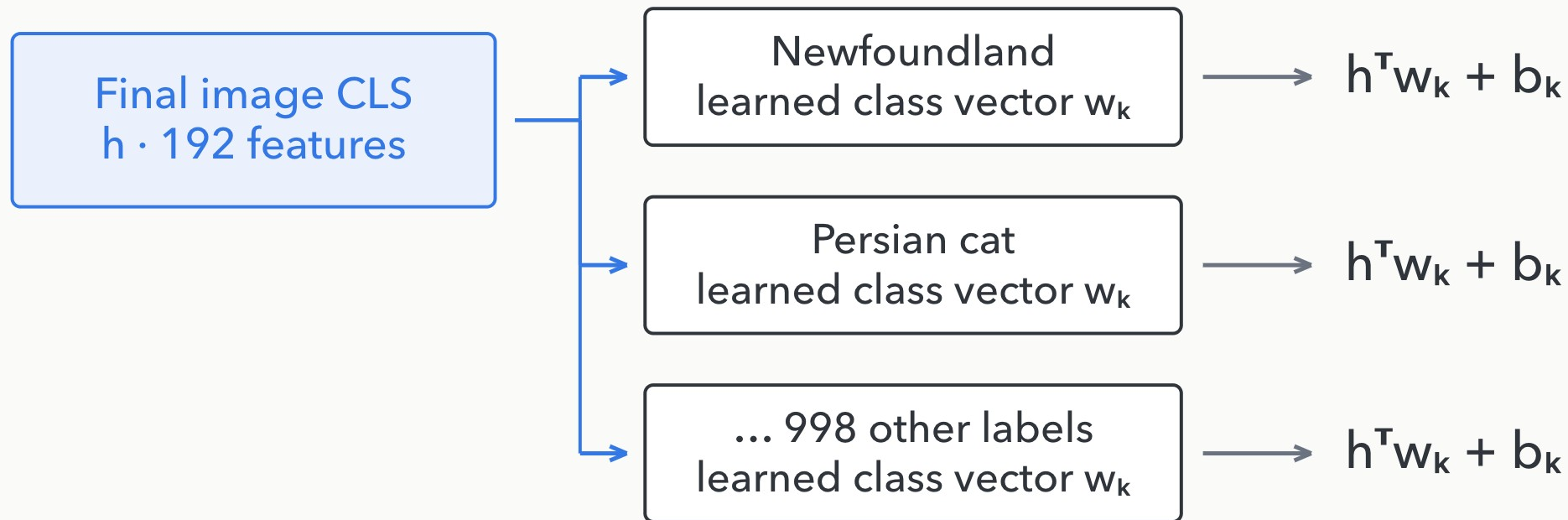
4 • More tokens cost more



Twice the side length: $\approx 16\times$ scores.

Pixels supply content. Position supplies location. Attention builds context. Training makes the final representation useful for the task.

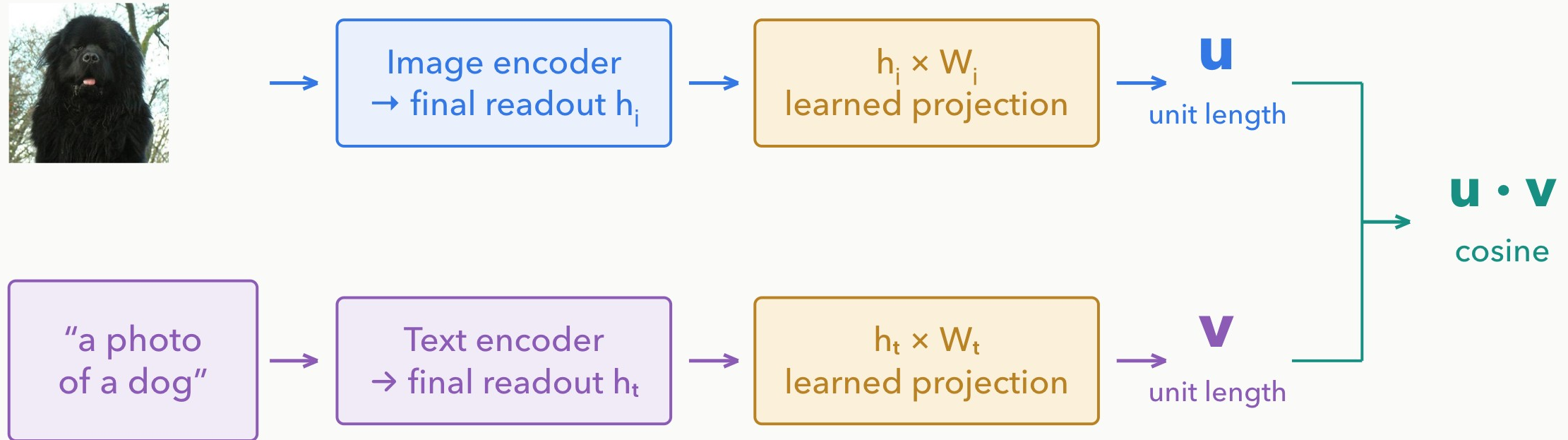
Our classifier stores a vector for each known label



The head stores one learned vector and one bias per label.

A class score is a dot product with its learned weight vector, plus a bias. The current head has 1,000 fixed label slots. New tasks can train a replacement head.

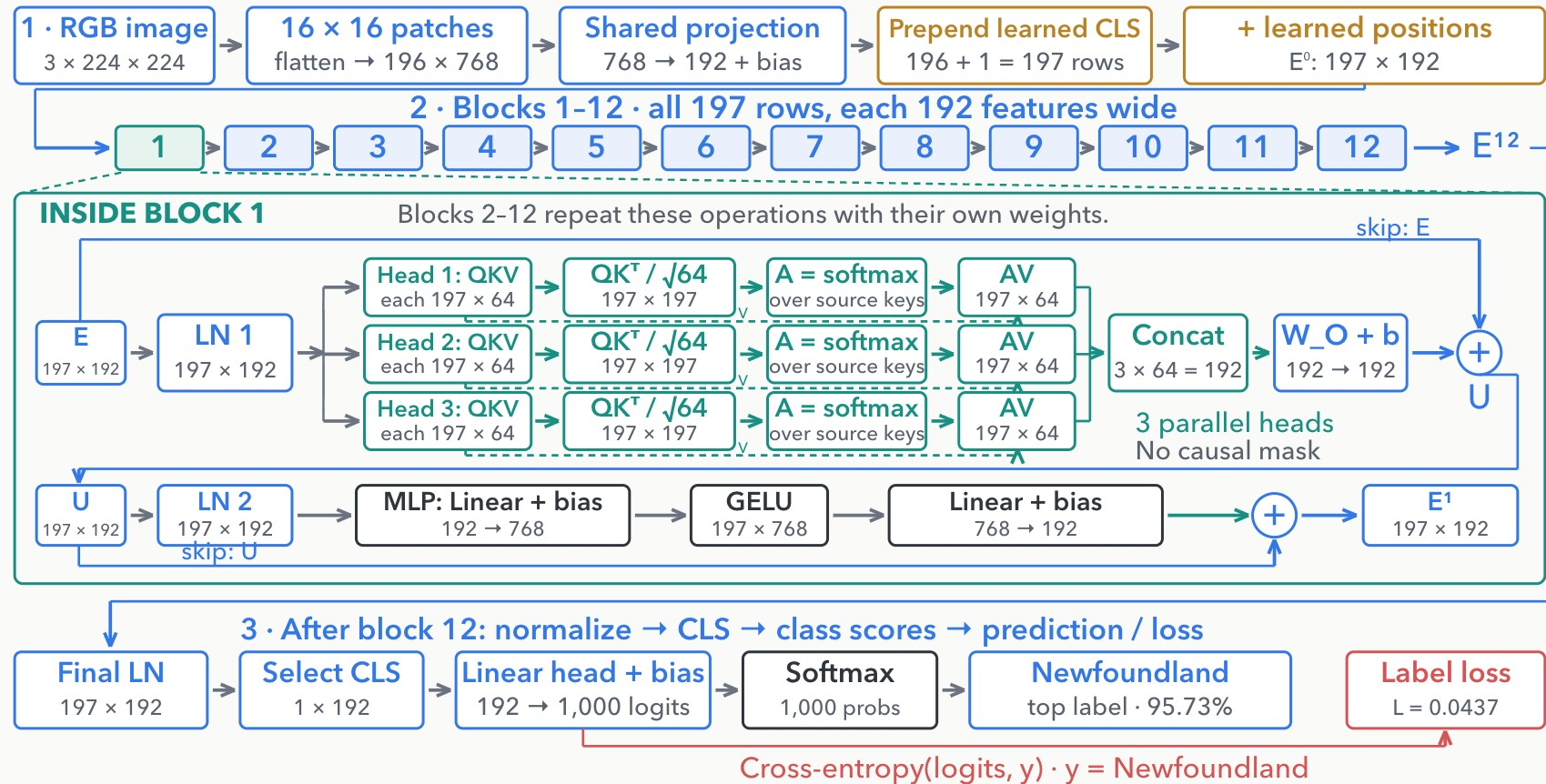
What if a class vector could come from language?



Separate encoders. Aligned vectors. Compare in one shared space.

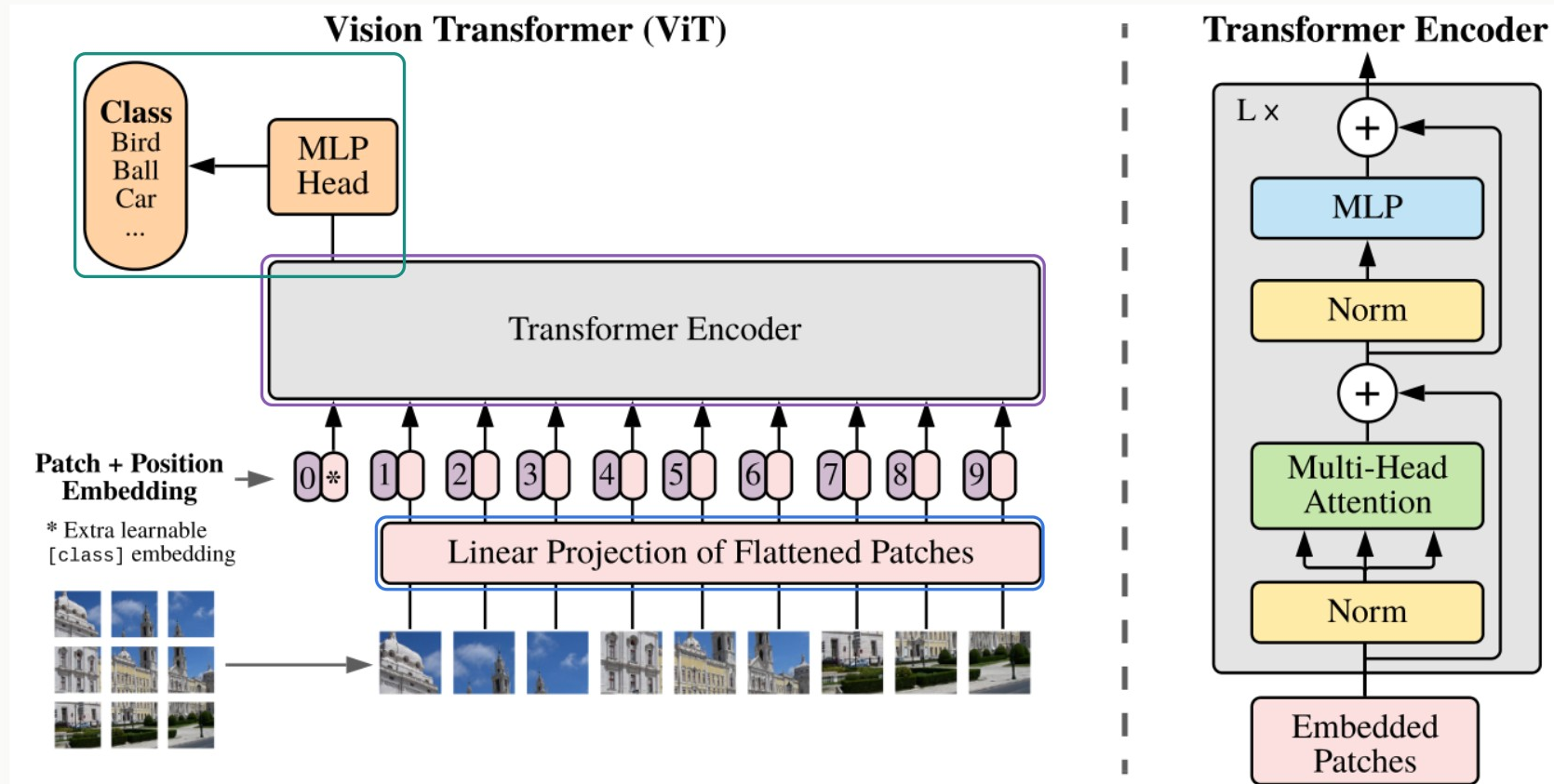
CLIP trains image and text representations to match. Learned projections and unit normalization make their vectors comparable; prompts can then describe candidate classes.

The whole Vision Transformer in one figure



Follow the numbered path. The expanded block shows three parallel heads, both residual additions, and the MLP. All 12 blocks keep every row. Final CLS feeds the classifier; the known label enters only at the loss.

To images: An Image is Worth 16×16 Words



Dosovitskiy et al. (2020; ICLR 2021), Figure 1 · arXiv:2010.11929

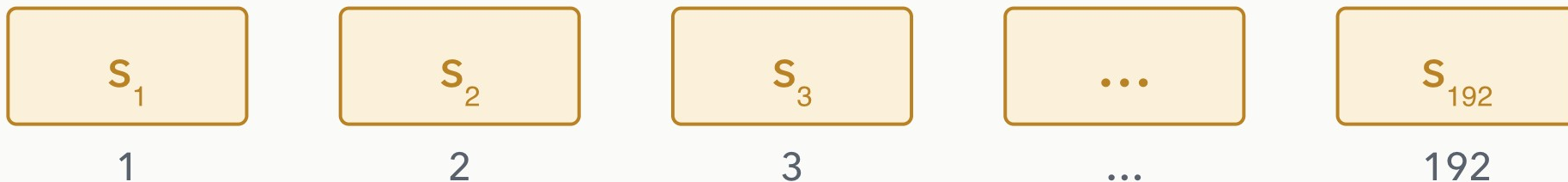
Image patches become tokens. The encoder builds a summary for classification.

Create 192 trainable numbers for CLS



The model uses 192 features per row.

Give the extra row the same width.



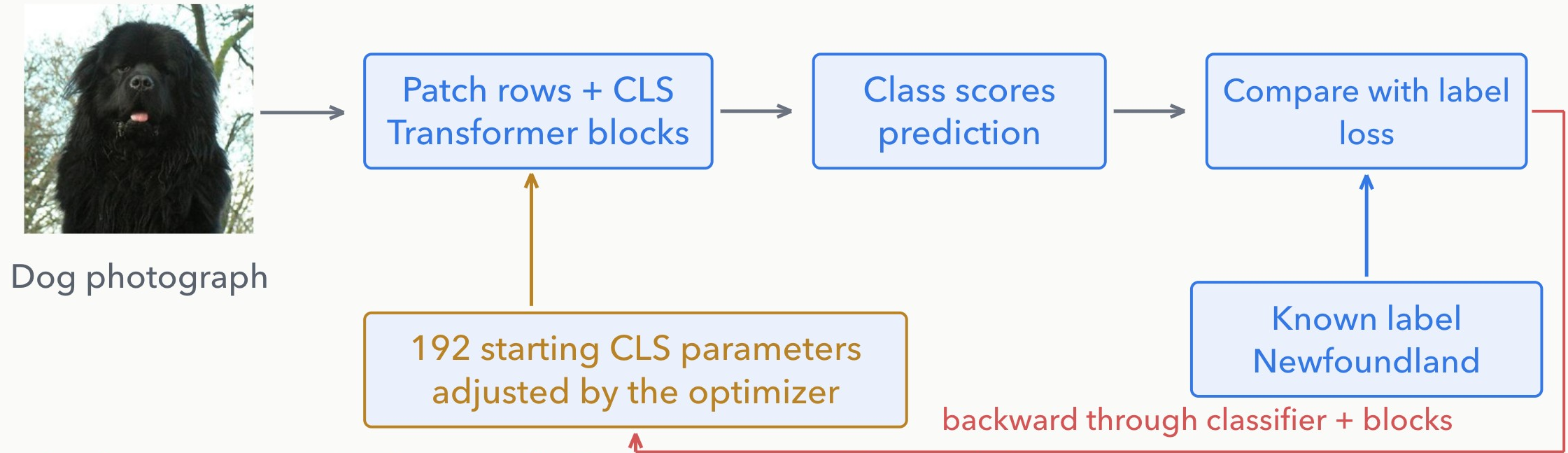
Initialize once with small random values; mark them trainable.

```
cls = nn.Parameter(torch.randn(1, 1, 192) * 0.02)
```

Before training, initialize one 192-number vector. It is stored in the model, like a learned token embedding in text. The numbers do not come from this photograph. Training will adjust them.

The image label teaches the starting CLS numbers

Training idea · imagine this is a labelled training example



The loss trains CLS along with the other trainable model parameters.

The label supplies a loss on the prediction. Backpropagation reaches the starting CLS vector through the classifier and blocks. The optimizer adjusts its 192 parameters, along with other trainable weights, across many labelled images.