Lecture 5:
First contact with Transformers

Advanced deep learning

Rémy Sun
remy.sun@inria.fr

Course organization

- Goal: In-depth understanding of important Deep Learning staples
  - Reinforce what you have already seen
  - Introduce state of the art models

- This is a hands-on course in pytorch
  - Minimal math
    - Enough to understand
  - Quite a bit of coding
  - Get comfortable with the standard pipeline

Evaluation

- Hand in one or two lab notebooks
  - Questions + (clean) code
  - 1st notebook: Lab 5 on transformers (Today!)
    - To hand in after vacation (Nov. 4 AoE)
- Written exam at end of semester
  - Little to no code
  - A few exercises on toy examples
  - Questions on aspects of deep learning

Course organization: 10 Lectures

UNIVERSITÉ CÔTE D'AZUR

- L1-2: Overview of Deep Learning (F. Precioso)
- L3-4: Fundamentals of Deep Learning (R. Sun)

- **L5-6: Transformers (R. Sun)**
- L7: Large models (LLMs, VLMs, Generators) (R. Sun)

- L8: Tricks of the trade (R. Sun)

- L9: Ethics of AI (F. Precioso)

- L10: Intro to generative models (P-A. Mattei)

Today!

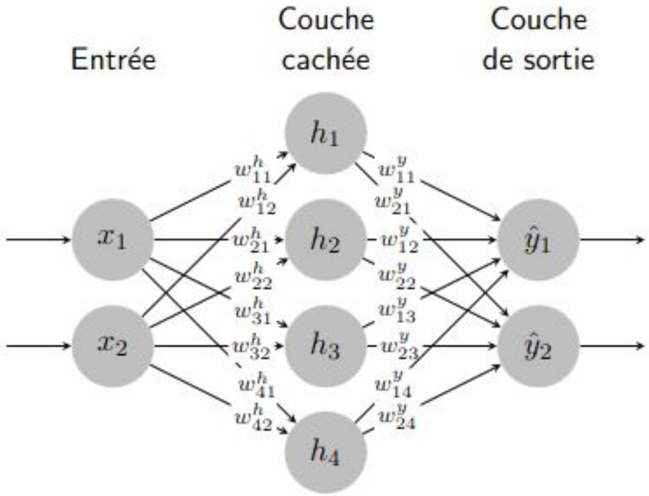
UNIVERSITÉ CÔTE D'AZUR

- Goal: Understand basic deep architectures in-depth
  - Building blocks for everything else in deep learning
  - Deep learning relies on the combination of a lot of very simple blocks

- A few things to take away after these 3 lectures
  - What do we optimize for? How? Why?
  - How do we build and train neural layers?
  - How do they behave?

Refresher on last week

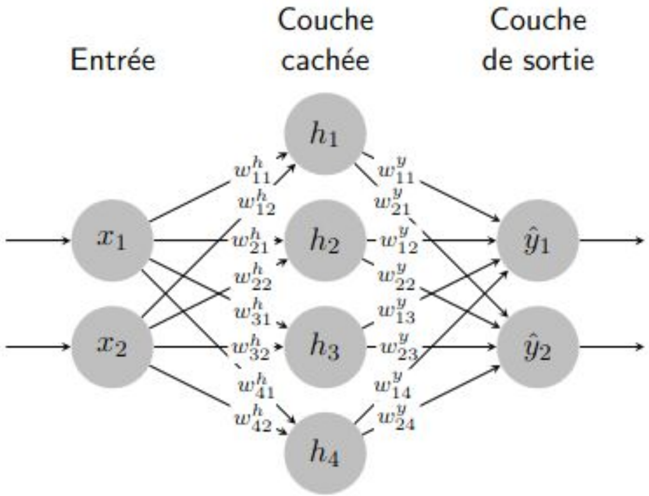
Example: Tanh MLP



- Simple 1 hidden layer MLP
  - 2 inputs
  - 2 outputs
  - 4 hidden activations
- Classification problem
  - Outputs probabilities
  - Cross-entropy loss

$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

TP1: Manual Backprop



$$\begin{cases} \tilde{\mathbf{h}} = \mathbf{x}{\mathbf{W}^h}^\top + \mathbf{b}^h \\ \mathbf{h} = \tanh(\tilde{\mathbf{h}}) \\ \tilde{\mathbf{y}} = \mathbf{h}{\mathbf{W}^y}^\top + \mathbf{b}^y \\ \hat{\mathbf{y}} = \mathrm{SoftMax}(\tilde{\mathbf{y}}) \end{cases} \qquad \begin{cases} \nabla_{\tilde{\mathbf{y}}} = \hat{\mathbf{y}} - \mathbf{y} \\ \nabla_{\mathbf{W}^y} = {\nabla_{\tilde{\mathbf{y}}}}^\top \mathbf{h} \\ \nabla_{\mathbf{b}^y} = \nabla_{\tilde{\mathbf{y}}}^\top \\ \nabla_{\tilde{\mathbf{h}}} = (\nabla_{\tilde{\mathbf{y}}} \mathbf{W}^y) \odot (1 - \mathbf{h}^2) \\ \nabla_{\mathbf{W}^h} = {\nabla_{\tilde{\mathbf{h}}}}^\top \mathbf{x} \\ \nabla_{\mathbf{b}^h} = \nabla_{\tilde{\mathbf{h}}}^\top \end{cases}$$

- Lecture 1 practical correction on Moodle
  - Implement this by hand with basic torch!
  - Careful with batch dimension!

Manual backprop

```python
def init_params(nx, nh, ny):
    """
    nx, nh, ny: integers
    out params: dictionnary
    """
    params = {}

    params["Wh"] = torch.randn((nh, nx))*0.3
    params["Wy"] = torch.randn((ny, nh))*0.3
    params["bh"] = torch.zeros((nh,1))
    params["by"] = torch.zeros((ny,1))

    return params
```

Manual backprop

UNIVERSITÉ CÔTE D'AZUR

```
def forward(params, X):
    """
    params: dictionnary
    X: (n_batch, dimension)
    """
    bsize = X.size(0)
    nh = params['Wh'].size(0)
    ny = params['Wy'].size(0)
    outputs = {}

    outputs["X"] = X
    outputs["htilde"] = torch.mm(X, params["Wh"].T) + params["bh"].T
    outputs["h"] = torch.tanh(outputs["htilde"])
    outputs["ytilde"] = torch.mm(outputs["h"], params["Wy"].T) + params["by"].T
    outputs["yhat"] = torch.exp(outputs["ytilde"])
    outputs["yhat"] = outputs["yhat"] / outputs["yhat"].sum(dim=-1, keepdim=True)


    return outputs['yhat'], outputs
```

Manual backprop

```
def backward(params, outputs, Y):
    bsize = Y.shape[0]
    grads = {}

    Y_tilde_grad = outputs["yhat"] - Y
    h_tilde_grad = torch.mm(Y_tilde_grad, params['Wy']
                            ) * (1 - torch.pow(outputs['h'], 2))

    grads["Wy"] = torch.mm(Y_tilde_grad.T, outputs["h"])
    grads["Wh"] = torch.mm(h_tilde_grad.T, outputs['X'])
    grads["by"] = Y_tilde_grad.sum(dim=0,keepdim=True).T
    grads["bh"] = h_tilde_grad.sum(0, keepdim=True).T

    grads['Wy'] /= bsize
    grads['by'] /= bsize
    grads['Wh'] /= bsize
    grads['bh'] /= bsize

    return grads
```

Manual backprop

```python
def sgd(params, grads, eta):

    params['Wy'] -= eta * grads['Wy']
    params['Wh'] -= eta * grads['Wh']
    params['by'] -= eta * grads['by']
    params['bh'] -= eta * grads['bh']

    return params
```

Manual backprop

UNIVERSITÉ CÔTE D'AZUR

```
for j in range(N // Nbatch):

    indsBatch = range(j * Nbatch, (j+1) * Nbatch)
    X = Xtrain[indsBatch, :]
    Y = Ytrain[indsBatch, :]

    Y_hat, outputs = forward(params, X)
    loss, accuracy = loss_accuracy(Y_hat, Y)
    grads = backward(params, outputs, Y)
    params = sgd(params, grads, eta)
```

Part 2: Autograd backward

UNIVERSITÉ CÔTE D'AZUR

```
def backward(params, outputs, Y):
    bsize = Y.shape[0]
    grads = {}

    Y_tilde_grad = outputs["yhat"] - Y
    h_tilde_grad = torch.mm(Y_tilde_grad, params['Wy']
                            ) * (1 - torch.pow(outputs['h'], 2))

    grads["Wy"] = torch.mm(Y_tilde_grad.T, outputs["h"])
    grads["Wh"] = torch.mm(h_tilde_grad.T, outputs['X'])
    grads["by"] = Y_tilde_grad.sum(dim=0,keepdim=True).T
    grads["bh"] = h_tilde_grad.sum(0, keepdim=True).T

    grads['Wy'] /= bsize
    grads['by'] /= bsize
    grads['Wh'] /= bsize
    grads['bh'] /= bsize

    return grads
```

- Torch.tensor object
    - Np.array like
    - Tracked on a computational graph
    - .grad variable to track gradients
    - .backward to backpropagate gradients through the graph
    - Activate .autograd!

Part 2: Autograd backward

```python
def backward(params, outputs, Y):
    bsize = Y.shape[0]
    grads = {}

    Y_tilde_grad = outputs["yhat"] - Y
    h_tilde_grad = torch.mm(Y_tilde_grad, params['Wy']
                            ) * (1 - torch.pow(outputs['h'], 2))

    grads["Wy"] = torch.mm(Y_tilde_grad.T, outputs["h"])
    grads["Wh"] = torch.mm(h_tilde_grad.T, outputs['X'])
    grads["by"] = Y_tilde_grad.sum(dim=0,keepdim=True).T
    grads["bh"] = h_tilde_grad.sum(0, keepdim=True).T

    grads['Wy'] /= bsize
    grads['by'] /= bsize
    grads['Wh'] /= bsize
    grads['bh'] /= bsize

    return grads
```

```python
params['Wh'] = torch.randn(nh, nx) * 0.3
params['Wh'].requires_grad = True
params['bh'] = torch.zeros(nh, 1, requires_grad=True)
params['Wy'] = torch.randn(ny, nh) * 0.3
params['Wy'].requires_grad = True
params['by'] = torch.zeros(ny, 1, requires_grad=True)
```

```python
with torch.no_grad():
    params['Wy'] -= eta * params['Wy'].grad
    params['Wh'] -= eta * params['Wh'].grad
    params['by'] -= eta * params['by'].grad
    params['bh'] -= eta * params['bh'].grad

    params['Wy'].grad.zero_()
    params['Wh'].grad.zero_()
    params['by'].grad.zero_()
    params['bh'].grad.zero_()
```

```python
Yhat, outputs = forward(params, X)
L, _ = loss_accuracy(Yhat, Y)
L.backward()
params = sgd(params, 0.03)
```

Part 2: Using a computational graph

- Torch keeps track of operations in a dynamic graph
  - Not static like tensorflow
  - Need to tell pytorch to do it
    - .requires_grad=True
  - Have to exclude graph operations
    - With torch.no_grad():

- Gradients are automatically computed
  - L.backward()
    - L is a scalar loss tensor
  - Go back through the graph to fill .grad

Part 3: torch.nn instantiation

```
params = {}

params["Wh"] = torch.randn((nh, nx))*0.3
params["Wy"] = torch.randn((ny, nh))*0.3
params["bh"] = torch.zeros((nh,1))
params["by"] = torch.zeros((ny,1))
```

```
def forward(params, X):
    """
    params: dictionnary
    X: (n_batch, dimension)
    """
    bsize = X.size(0)
    nh = params['Wh'].size(0)
    ny = params['Wy'].size(0)
    outputs = {}

    outputs["X"] = X
    outputs["htilde"] = torch.mm(X, params["Wh"].T) + params["bh"].T
    outputs["h"] = torch.tanh(outputs["htilde"])
    outputs["ytilde"] = torch.mm(outputs["h"], params["Wy"].T) + params["by"].T
    outputs["yhat"] = torch.exp(outputs["ytilde"])
    outputs["yhat"] = outputs["yhat"] / outputs["yhat"].sum(dim=-1, keepdim=True)


    return outputs['yhat'], outputs
```

- Torch.nn.Module objects
    - .__init__ creates weights and initializes them!
    - .forward implements forward operations
        - Default object call
        - model(x)
    - Some global control over model weights

Part 3: torch.nn instantiation

```
params = {}

params["Wh"] = torch.randn((nh, nx))*0.3
params["Wy"] = torch.randn((ny, nh))*0.3
params["bh"] = torch.zeros((nh,1))
params["by"] = torch.zeros((ny,1))
```

```
def forward(params, X):
    """
    params: dictionnary
    X: (n_batch, dimension)
    """
    bsize = X.size(0)
    nh = params['Wh'].size(0)
    ny = params['Wy'].size(0)
    outputs = {}

    outputs["X"] = X
    outputs["htilde"] = torch.mm(X, params["Wh"].T) + params["bh"].T
    outputs["h"] = torch.tanh(outputs["htilde"])
    outputs["ytilde"] = torch.mm(outputs["h"], params["Wy"].T) + params["by"].T
    outputs["yhat"] = torch.exp(outputs["ytilde"])
    outputs["yhat"] = outputs["yhat"] / outputs["yhat"].sum(dim=-1, keepdim=True)


    return outputs['yhat'], outputs
```

```
model = torch.nn.Sequential(
    torch.nn.Linear(nx, nh),
    torch.nn.Tanh(),
    torch.nn.Linear(nh, ny)
)
loss = torch.nn.CrossEntropyLoss()
```

```
_, indY = torch.max(Y, 1)
L = loss(Yhat, indY)

_, indYhat = torch.max(Yhat, 1)

acc = torch.sum(indY == indYhat.data) * 100 // indY.size(0);
```

```
with torch.no_grad():
    for param in model.parameters():
        param -= eta * param.grad
    model.zero_grad()
```

```
Yhat = model(X)
L, _ = loss_accuracy(loss, Yhat, Y)
L.backward()
model = sgd(model, 0.03)
```

Part 3: Building a network object

- What is a network?
  - Weights to be learned
  - A forward function that uses those weights

- Torch.nn modules implement this
  - Initialize weights in __init__
  - Implement forward
    - Default call of the object: model(x)
  - Large panel of existing layer implementations!
  - Even losses can be made as modules

Part 4: torch.optim optimization

```
def sgd(params, grads, eta):

    params['Wy'] -= eta * grads['Wy']
    params['Wh'] -= eta * grads['Wh']
    params['by'] -= eta * grads['by']
    params['bh'] -= eta * grads['bh']

    return params
```

- Torch.optim objects
    - Tracks learning rates
    - Tracks weights to optimize
    - Performs SGD steps
    - Even cleans up!

Part 4: torch.optim optimization

```
def sgd(params, grads, eta):

    params['Wy'] -= eta * grads['Wy']
    params['Wh'] -= eta * grads['Wh']
    params['by'] -= eta * grads['by']
    params['bh'] -= eta * grads['bh']

    return params
```

```
optim = torch.optim.SGD(model.parameters(), lr=eta)
```

```
yhat = model(X)
L,acc = loss_accuracy(loss,yhat,Y)
optim.zero_grad()
L.backward()
optim.step()
```

Part 4: Using an optimizer object

- What is an optimizer?
  - A way to update gradients
    - Operation on gradients
  - With parameters
    - (Possibly dynamic)

- Torch.optim optimizer
  - Instantiate as an object
    - With weights to track
    - With parameters of the optimization
  - Call optimizer.zero_grad() and optimizer.step()

Lessons to learn

- Training a network requires
  - Weights
  - A forward function
  - A backward function
  - Gradient steps

- Nice pytorch tools
  - Torch.tensor and torch.autograd
  - Torch.nn
  - Torch.optim

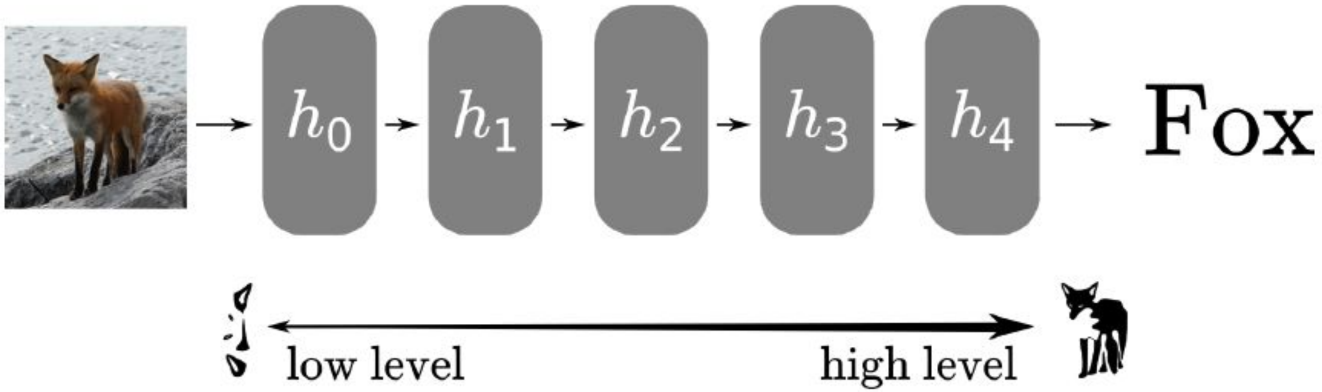
How does pytorch work?

- Create neural network
  - Use the torch.nn modules
    - Can use torch.nn.Modules
    - Or create a new class inheriting from torch.nn modules

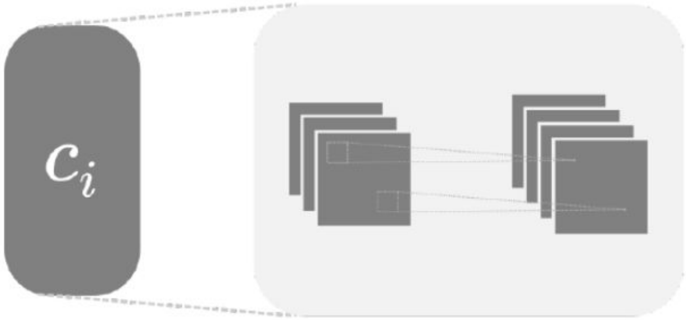
- In training loop

```
yhat = model(X)
L,acc = loss_accuracy(loss,yhat,Y)
optim.zero_grad()
L.backward()
optim.step()
```

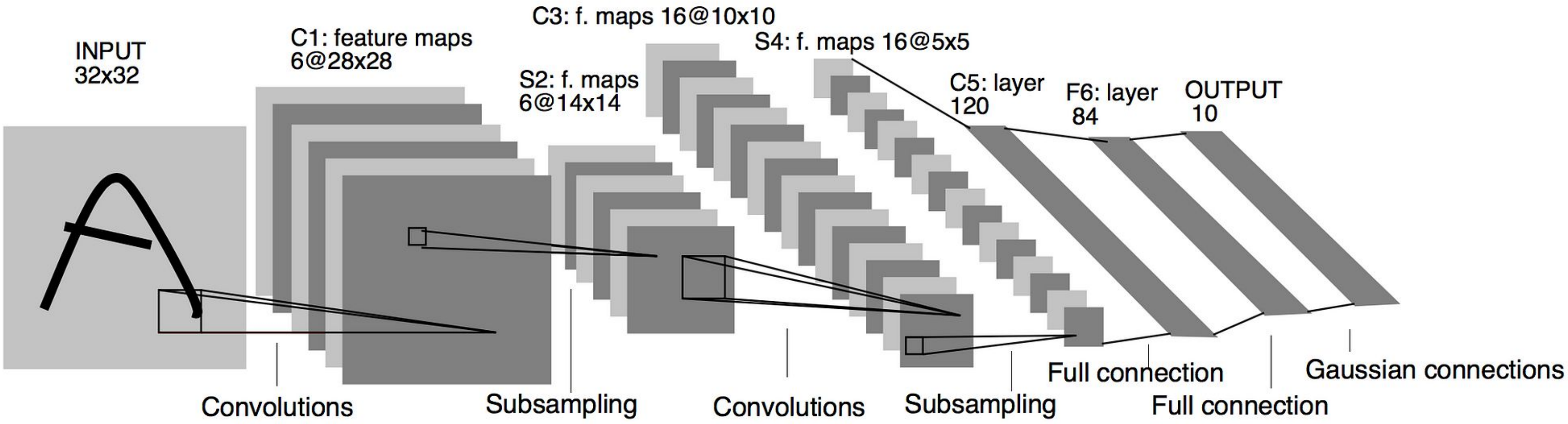
ConvNets for computer vision



- Convolutional layers
  - Local correlations
  - Well suited to images
  - Used sometimes with Transformers now



ConvNets for computer vision

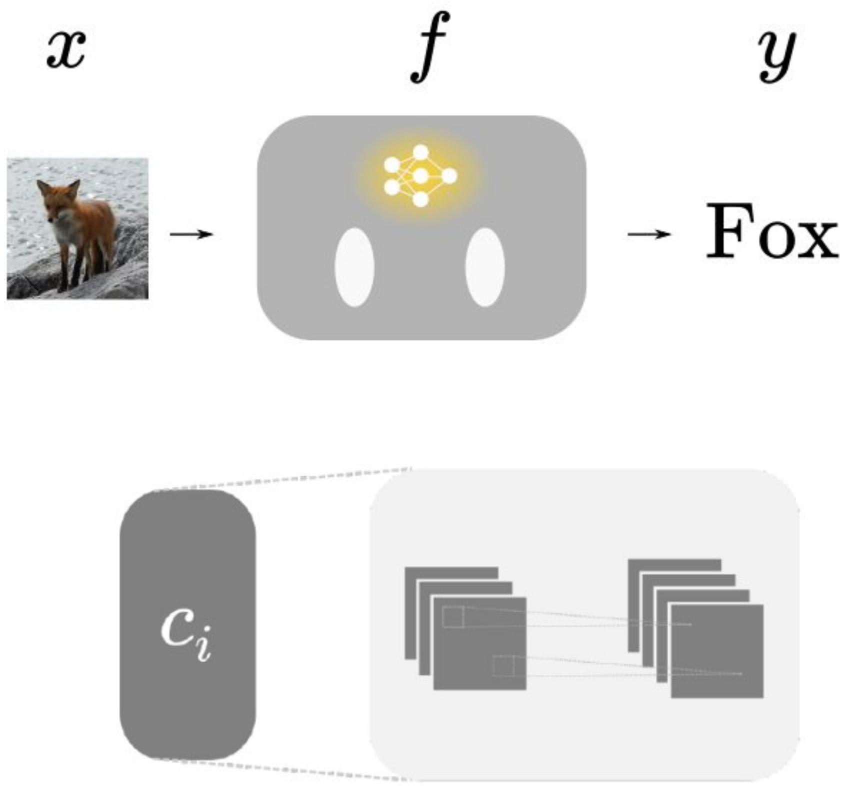


- Classical architecture of computer vision
  - Convolutional layers for feature extraction
  - Dense/linear layers to make decisions from features
  - E.g. LeNet5 (Before 2000!)

TP4b: Computer vision

- A few milestones
  - Load image data
  - Load a classic model
  - Train it!
  - (Finetune a strong model)

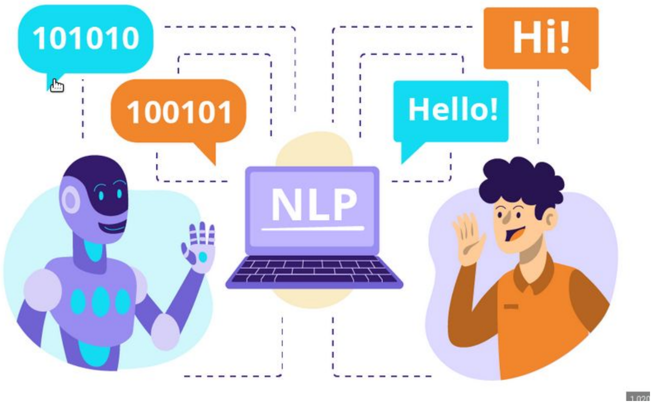
- Apply knowledge from TP4a!

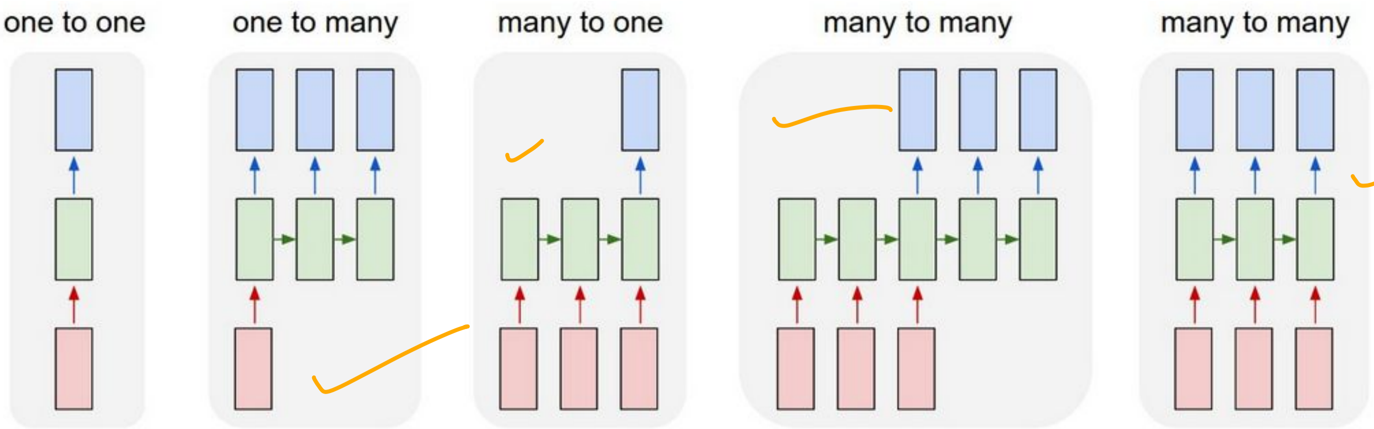


RNNs for Language

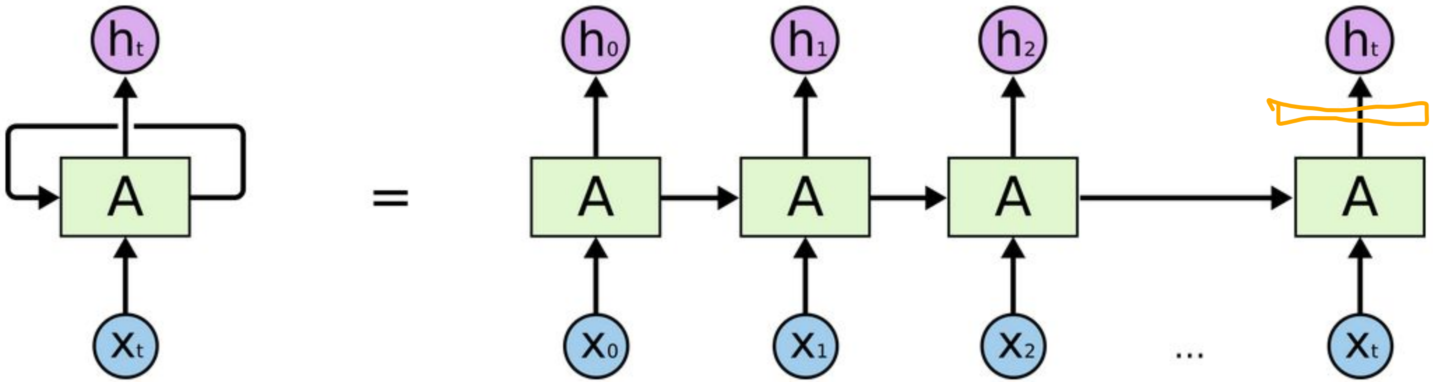
UNIVERSITÉ CÔTE D'AZUR

- Recurrent networks
  - Temporal correlations
  - How to take the past into account?
  - Recent resurgence (SSMs, …)

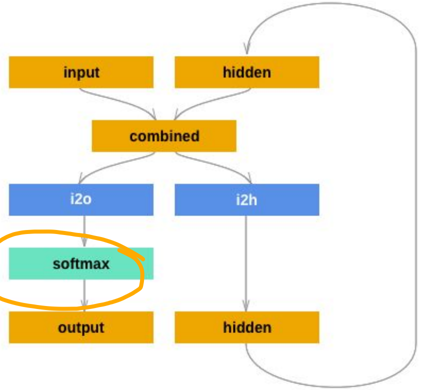




RNNs for Language



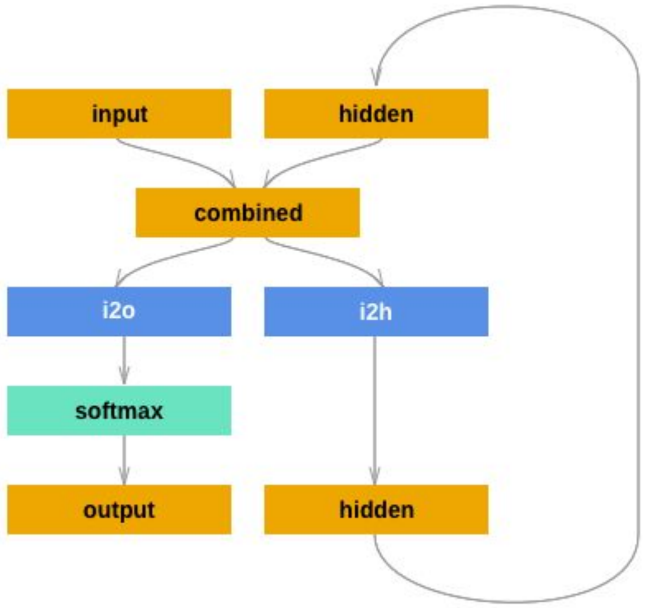
- Basic RNN model
  - Input + hidden state
  - Hidden state remains from input to input



TP4c: NLP

- Making a language processor
  - Tokenize words
  - Create network
  - Train network
  - Apply network

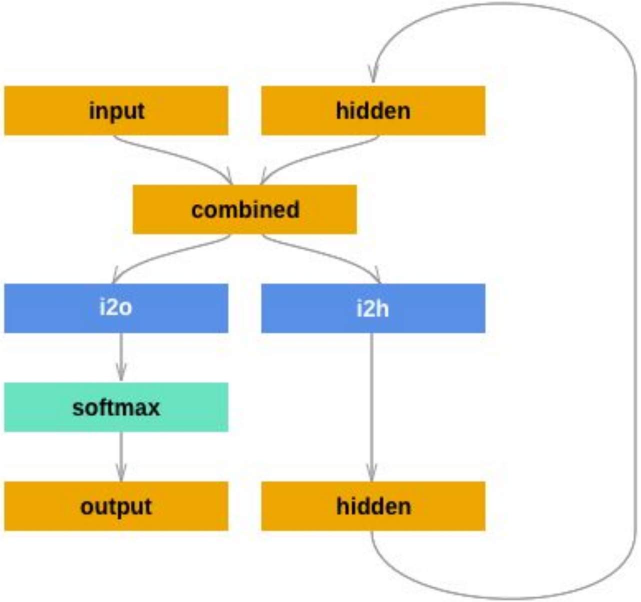
- Apply knowledge from TP4a!



TP4c: NLP

UNIVERSITÉ CÔTE D'AZUR

```python
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size

        ###################
        ## Your code here ##
        ###################

        # Init hidden state

        # Create Linear layers
        self.i2o = None
        self.i2h = None
        # Create activation function
        self.activation = None

    def forward(self, input):

        # Concatenate input and current hidden state
        concat = None

        # Pass through the two hidden layers the concatenation
        output = None
        hidden = None

        # Save the hidden state for later
        self.hidden = None

        # Compute the output
        output = None

        ###################
        ##      FIN       #
        ###################

        return output

    def init_hidden(self): # to reset hidden state
        self.hidden = torch.zeros(1, self.hidden_size)
```



Transformers: A new neural architecture

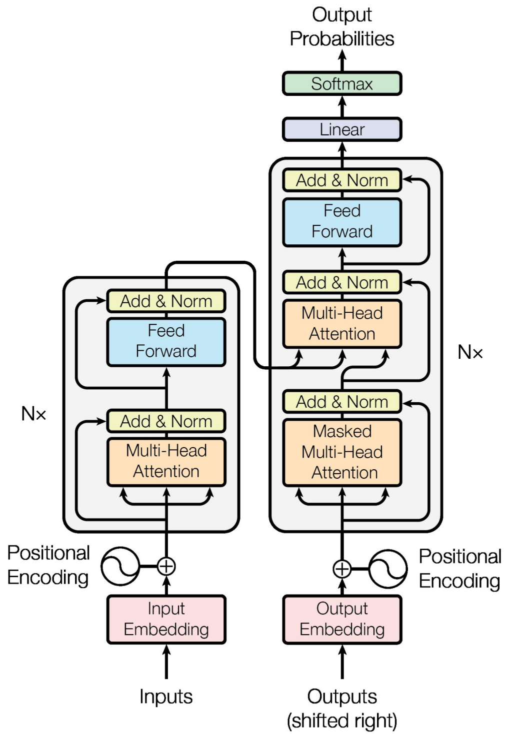
A new type of neural layer

UNIVERSITÉ CÔTE D'AZUR

- What is the state of the art in
  - Computer vision?
    - CNNs
  - Natural language processing?
    - RNNs
  - Time series?
    - RNNs or TCNs
  - Multimodal problems?
    - Hybrid?

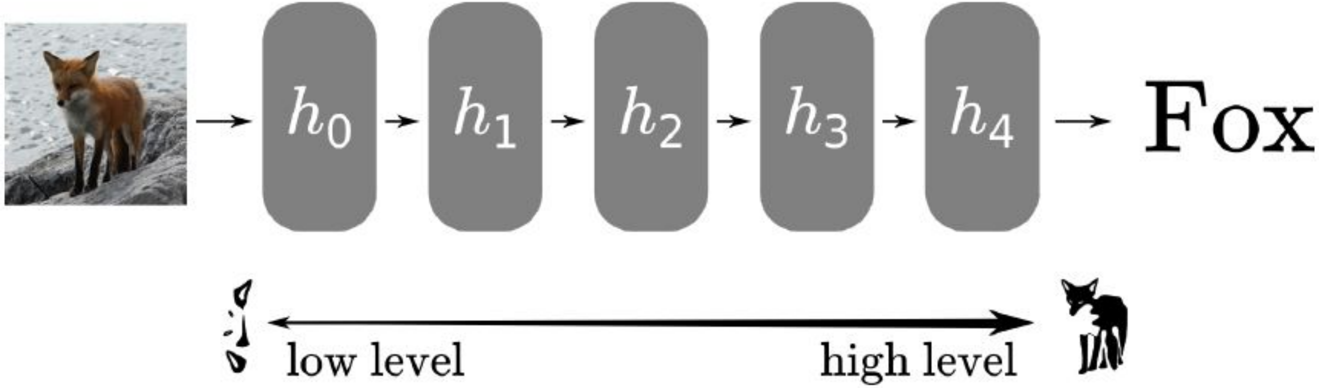
A new type of neural layer for everying

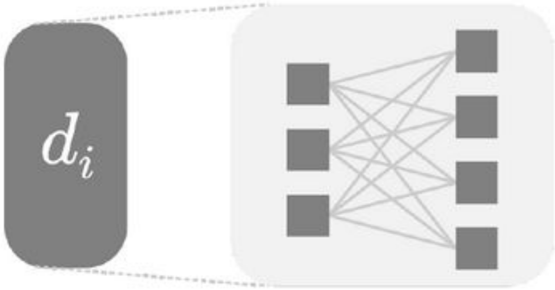
- What is the state of the art in
  - Computer vision?
    - ~~CNNs~~ -> **Transformers**
  - Natural language processing?
    - ~~RNNs~~ -> **Transformers**
  - Time series?
    - ~~RNNs or TCNs~~ -> **Transformers**
  - Multimodal problems?
    - ~~Hybrid?~~ -> **Transformers**



- Transformers use keeps increasing over time

Existing layer: Dense



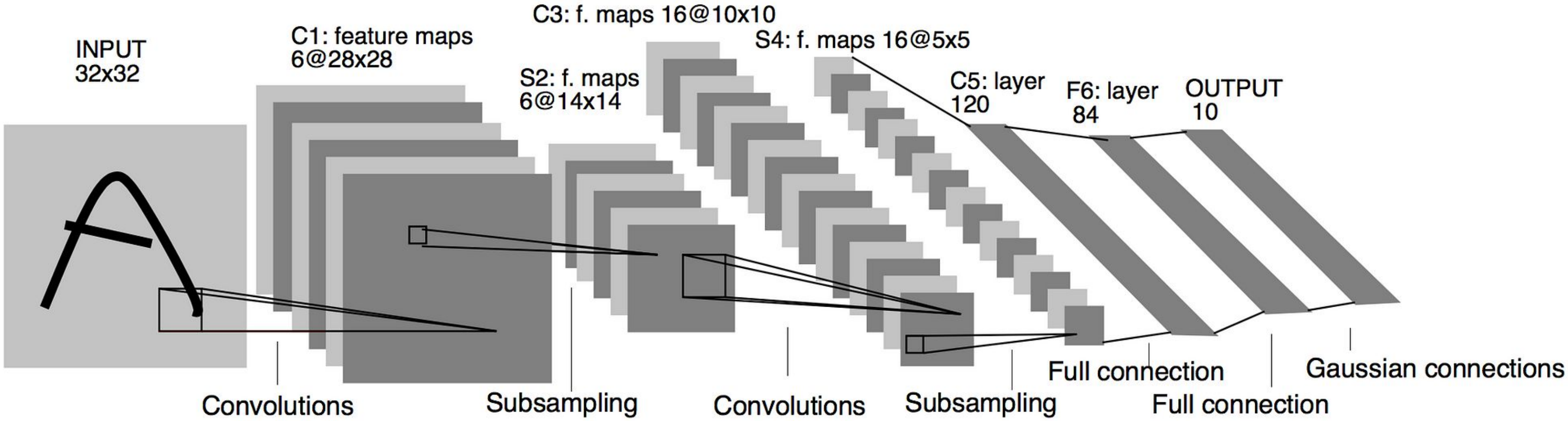


(a) Dense layer

$$d_\theta(x) = \sigma(W_\theta x^T + b_\theta)$$

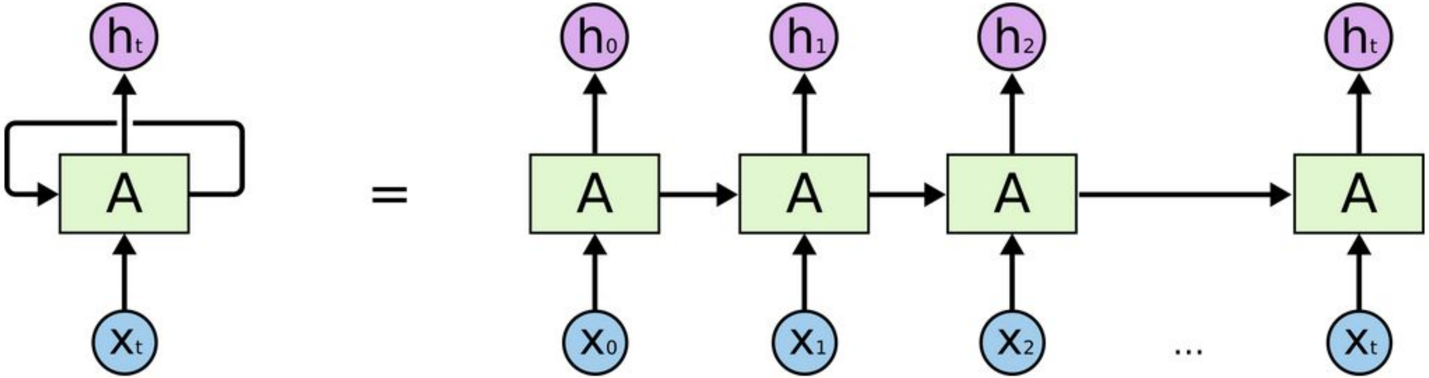
$$\sigma(x) = ReLU(x) = max(0, x)$$

Classic layer: CNN



Classic layer: RNN

UNIVERSITÉ CÔTE D'AZUR



A new challenger appears!

UNIVERSITÉ CÔTE D'AZUR

- RNNs have issues with long term memory
  - Try to allow it to look at past hidden states
    - That is a lot
  - **Attention** as a solution (~2015)
    - Only look at some past states

- Attention is all you need (Vaswani et al. 2017)
  - **The** transformer
    - Only attention and dense layers
  - Outperforms RNNs substantially

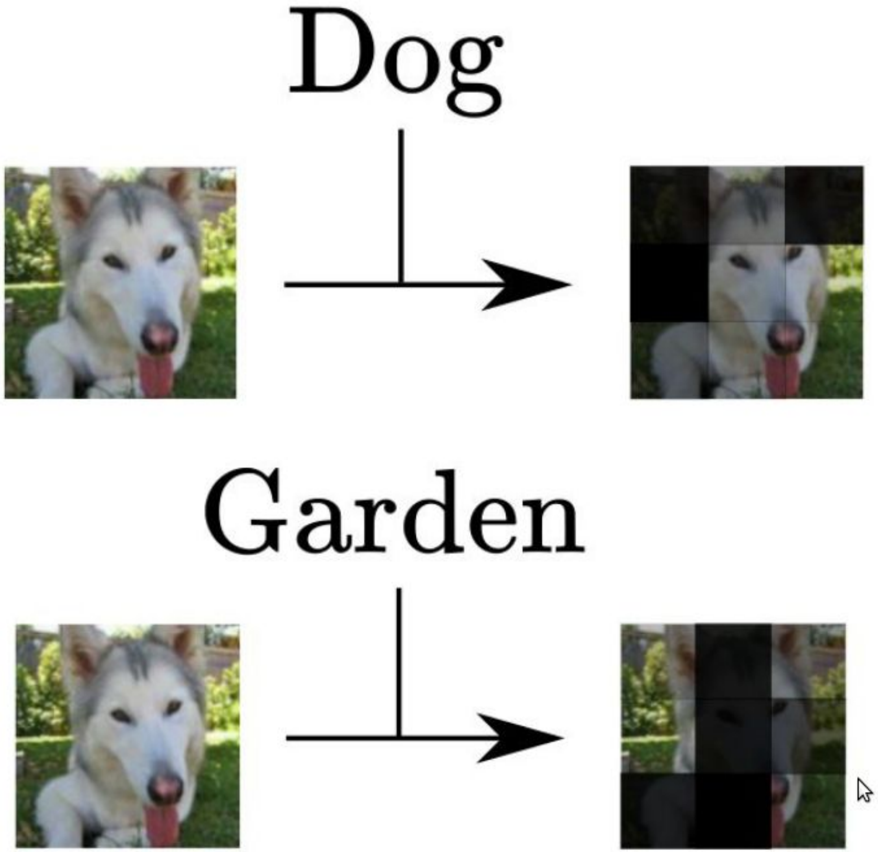
A new challenger appears!

- Attention is all you need (Vaswani et al. 2017)
  - **The** transformer
    - Only attention and dense layers
  - Outperforms RNNs substantially

- It gets harder in Computer Vision
  - Multiple attempts since 2017
  - An Image is Worth 16x16 Words (Disovitskyi et al. 2021)
    - (First) Vision Transformer that kind of works
      - With all the data in the world
  - Data efficient Image Transformers (Touvron et al. 2021)
    - Works with reasonable datasets

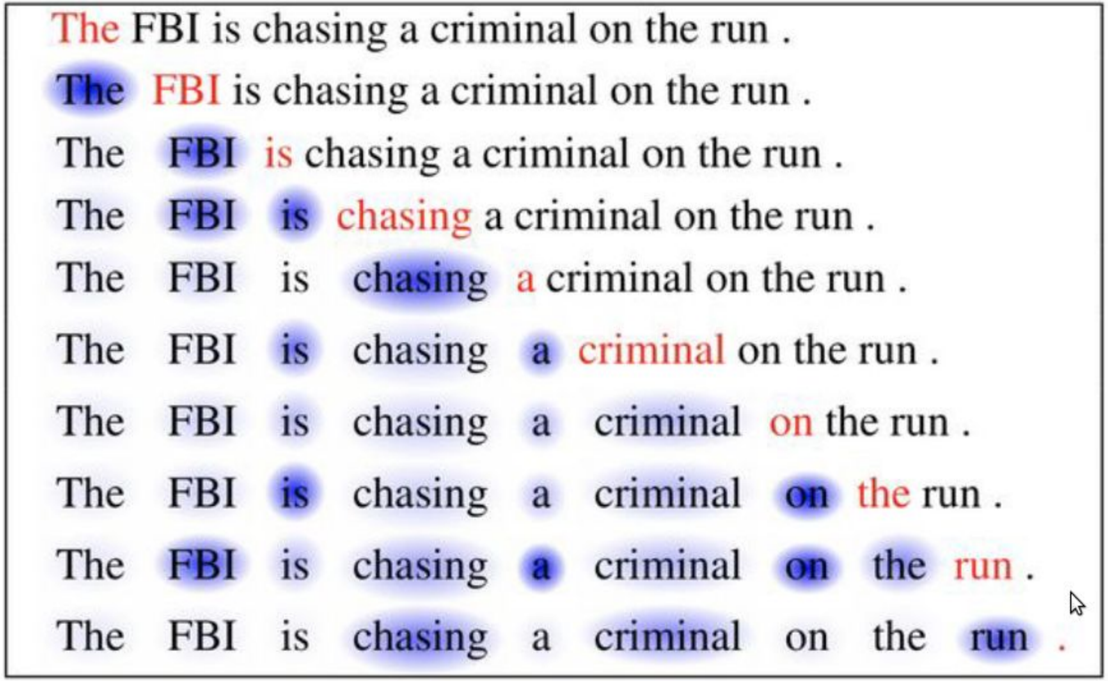
Attention introduction

Main idea



- What is a Dog?
    - It is a 4 legged animal with fur and ears and eyes and a head and …
    - It is on this part of that picture.

NLP example



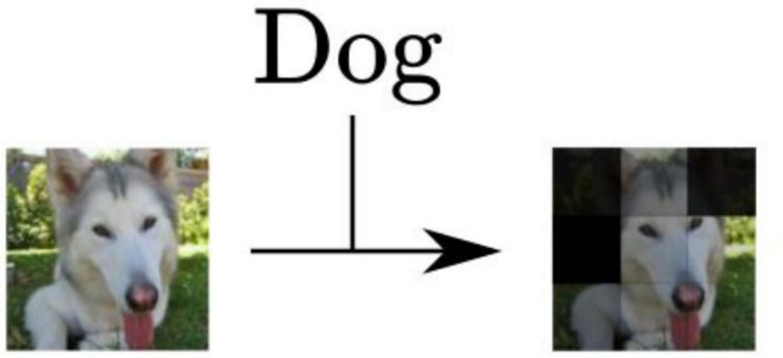
- Define a word by other words it relates to

Multimodal example

- Define a word by a few similar pictures

Attention mechanism

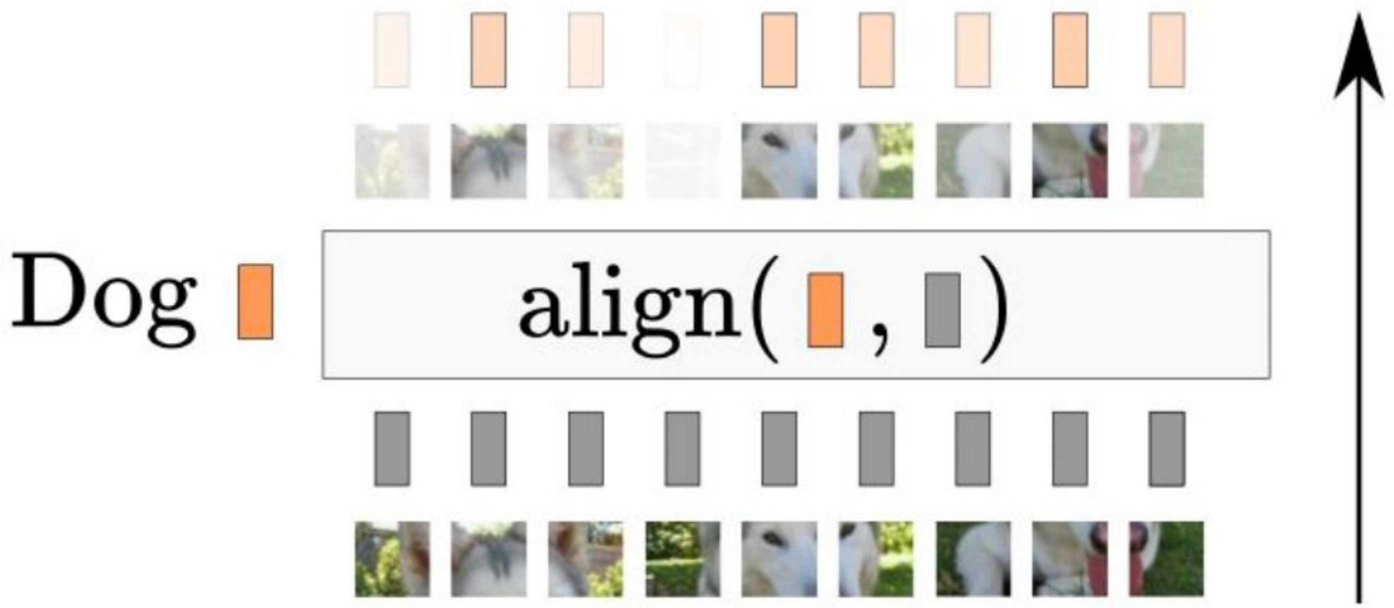
Core question



- Generic example between text and image
  - Could be something else!

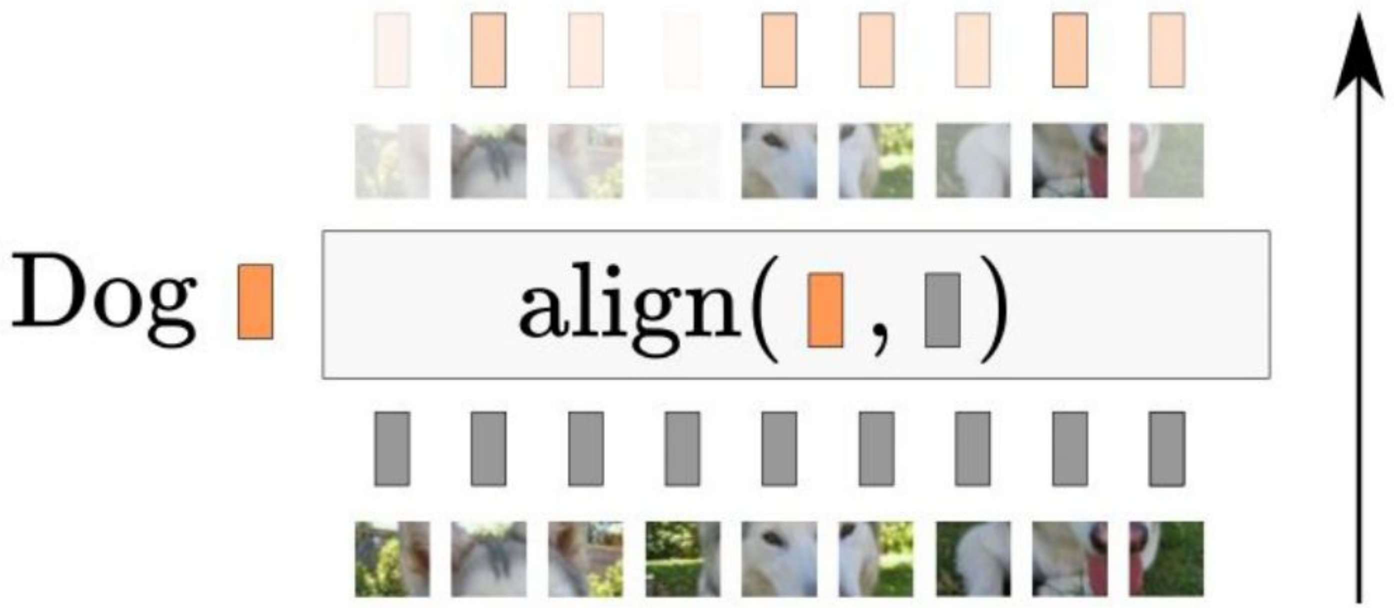
- How do you define the dog with the picture?

Alignment



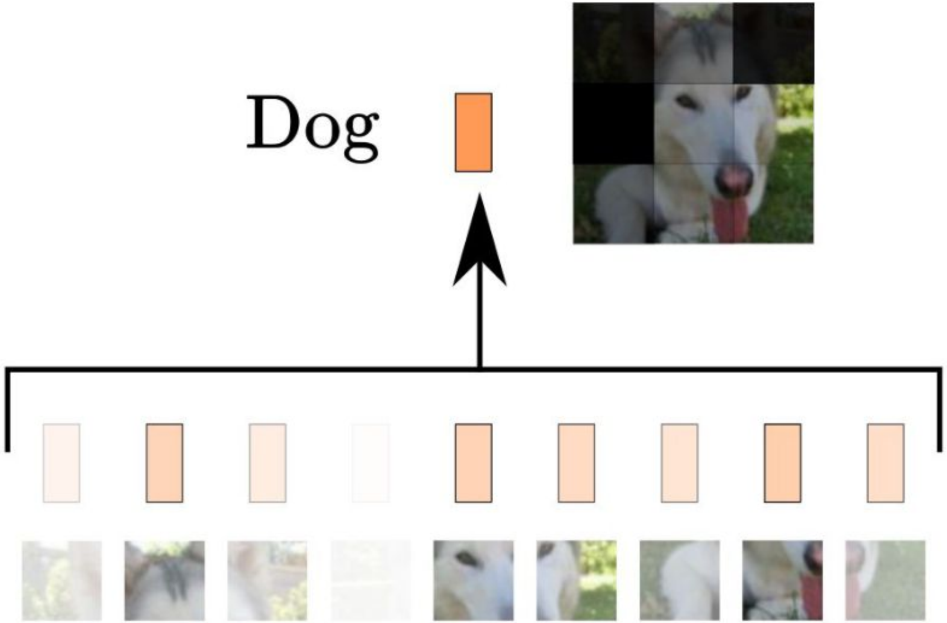
- Find the relevant parts of th picture

Alignment



- Find the relevant parts of the picture

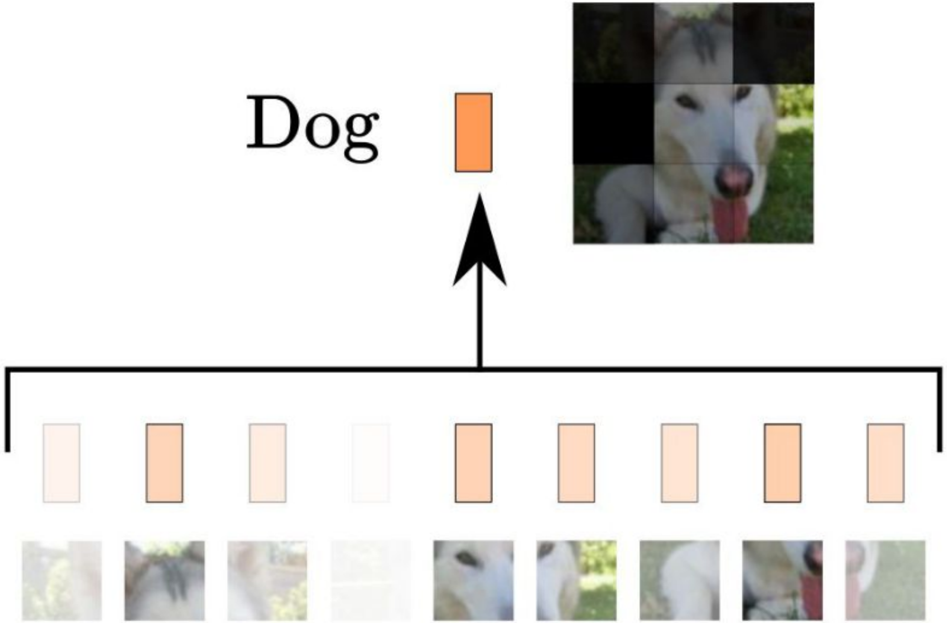
“Expected” representation



- Describe “Dog” by combining similar patch tokens

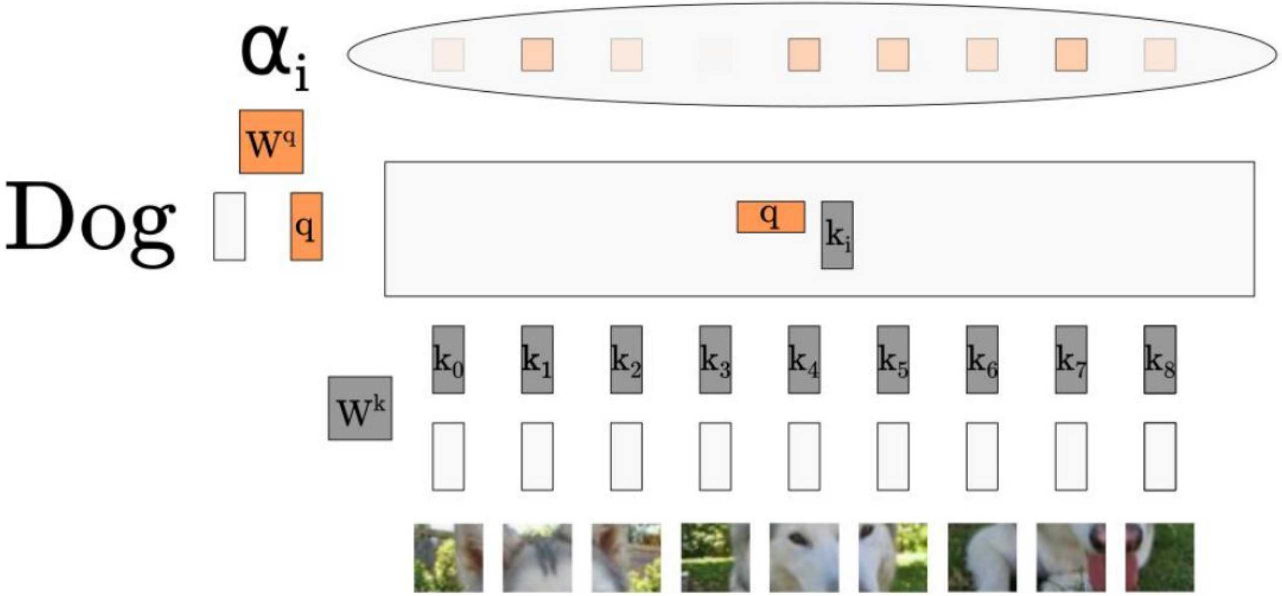
“Expected” representation

How?



- Describe “Dog” by combining similar patch tokens

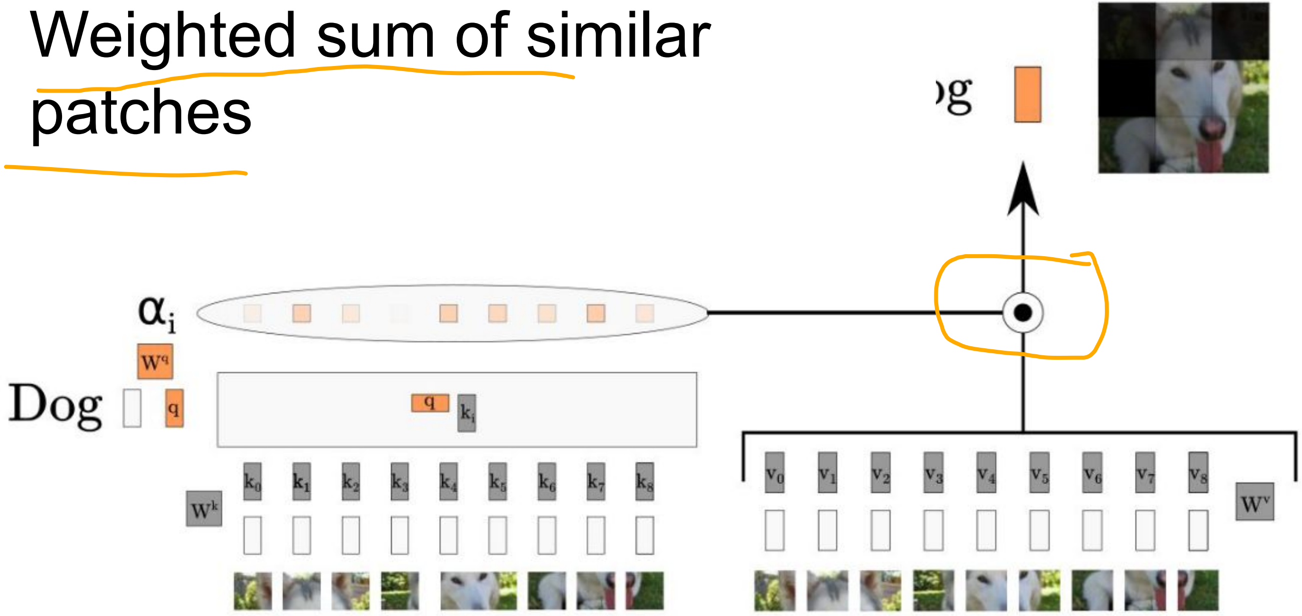
Attention weights



- Find an alignment score through dot product

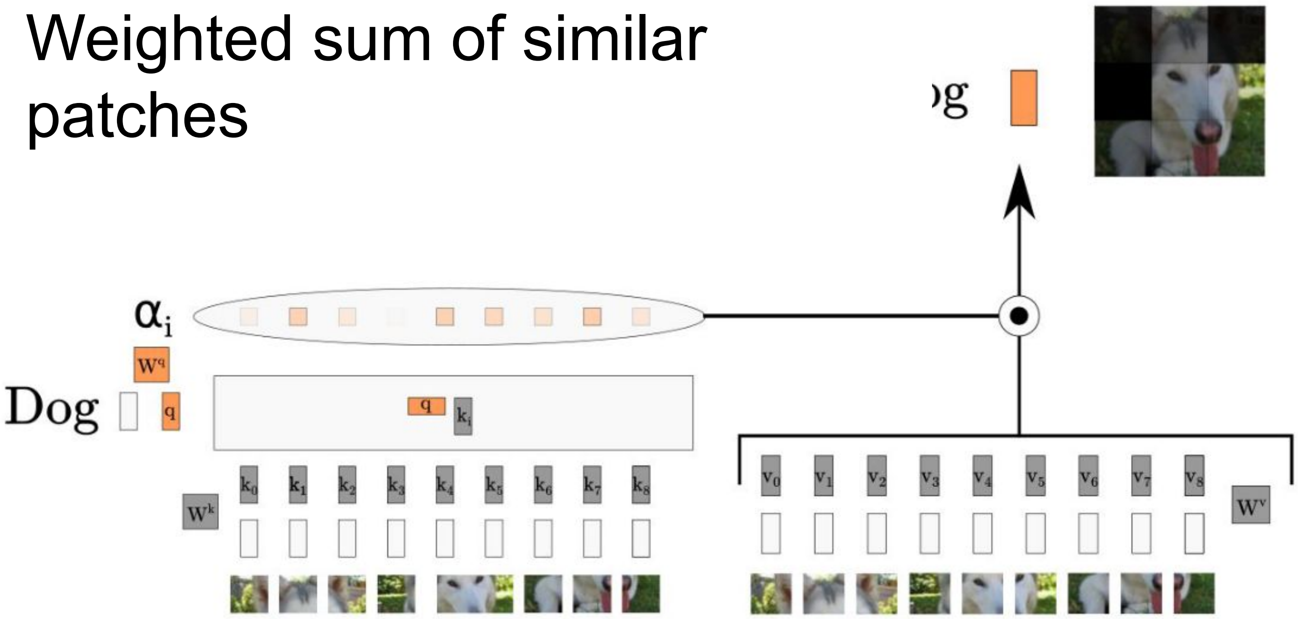
Attended representation

- Weighted sum of similar patches

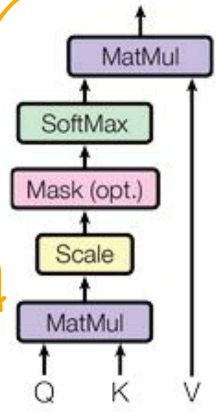


Attended representation

- Weighted sum of similar patches

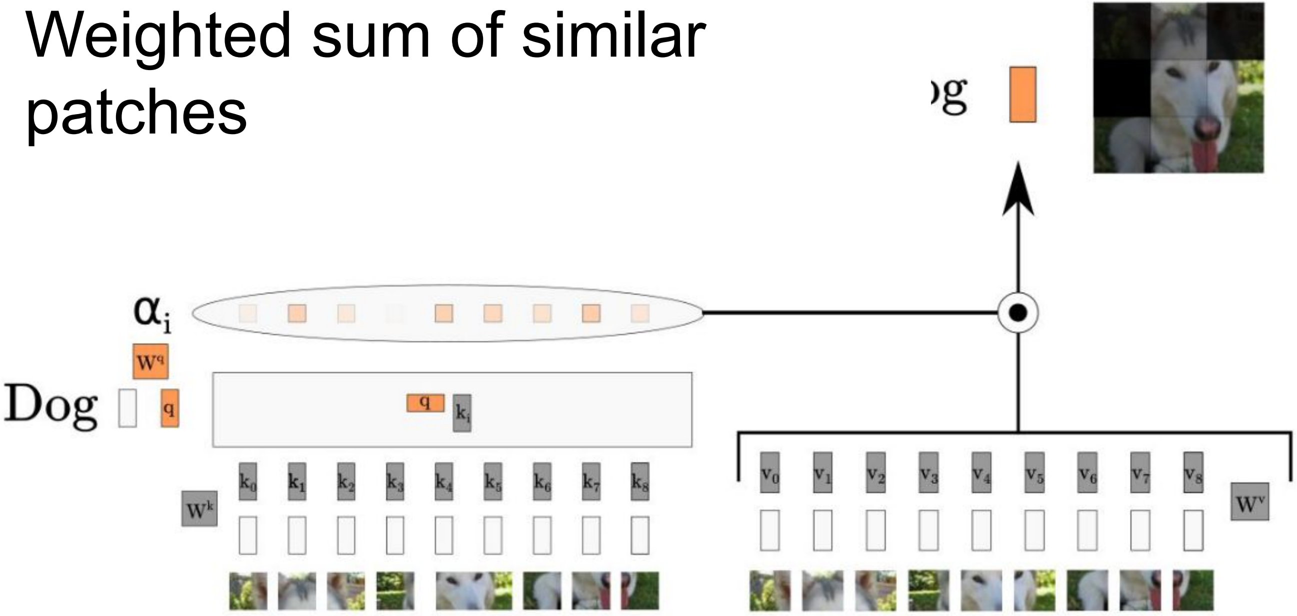


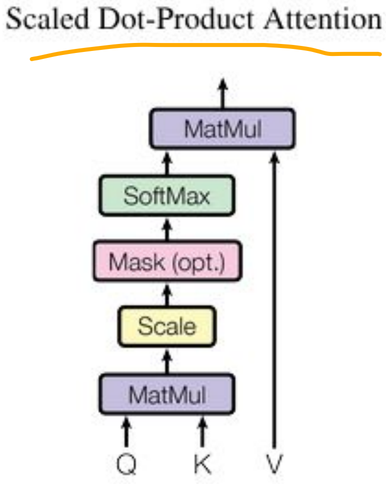
Scaled Dot-Product Attention

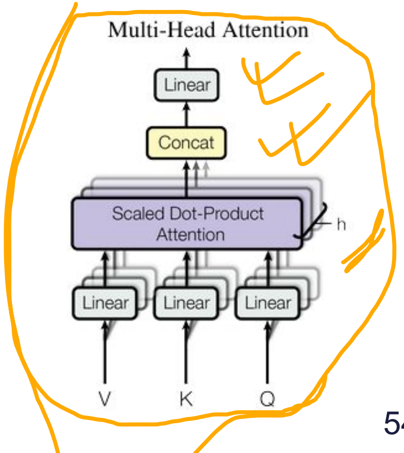


Attended representation

- Weighted sum of similar patches

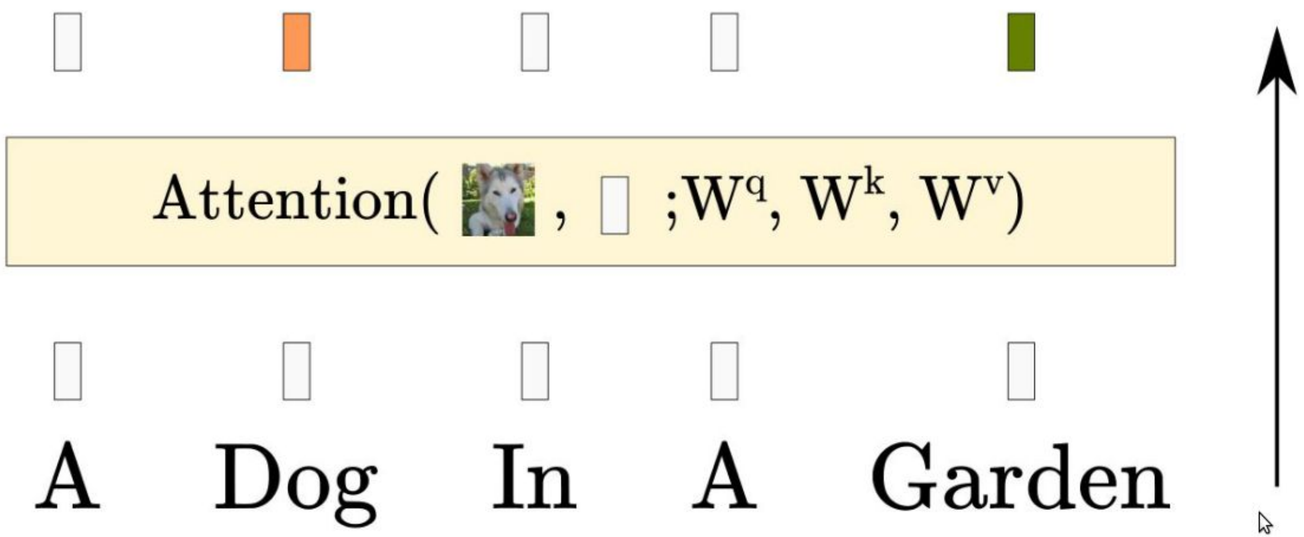






Attended representation

- Train up Weights to get good attention

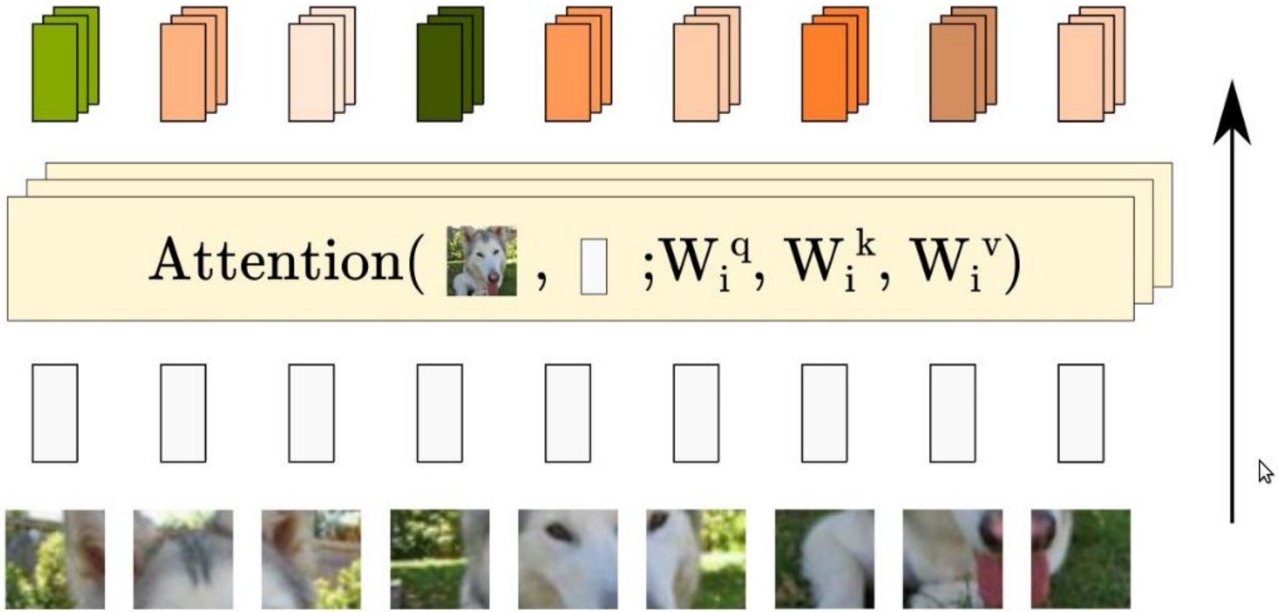


Self-attention

- You can compare a sequence to itself!

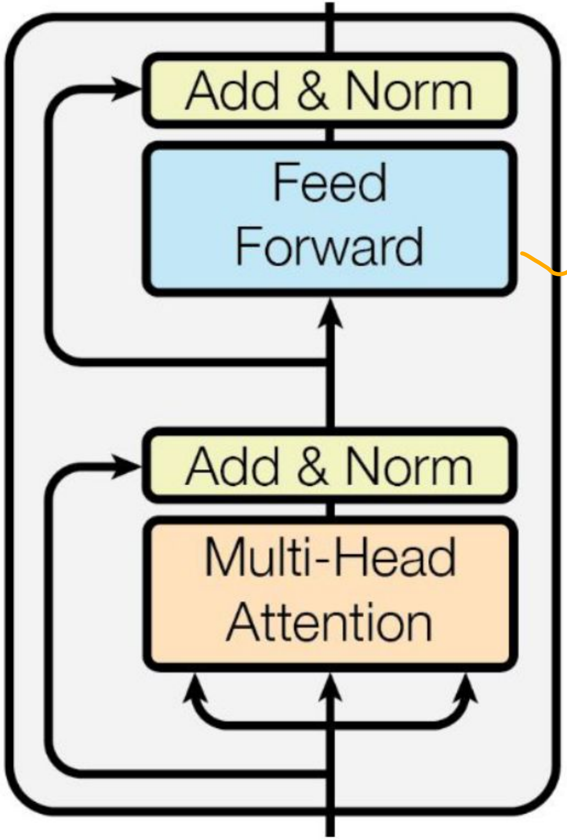
MultiHead

- And even have multiple interpretations



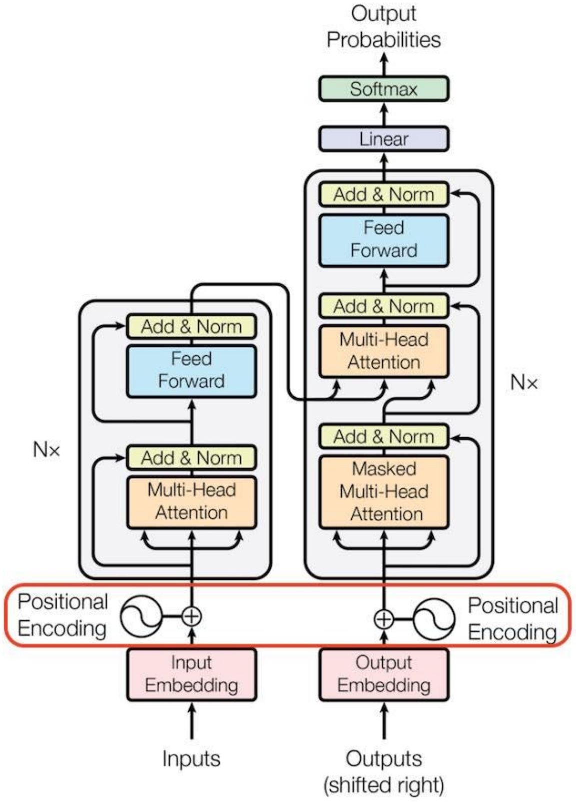
Transformer (Encoder) Block

- Basic encoder block
  - MultiHead attention
  - Skip connection
  - Small MLP applied to each token
  - Skip connection
- Stacked in a transformer



Positional encoding

- What is the order of the tokens?
  - Treated as a set
  - Permutation invariant
- How do keep positional info?
  - Masking
  - Add a positional encoding
    - Sine encoding
    - Learned encoding



Takeaway

- Attention: represent objects by similar things
- Mechanism
  - Compute alignment with product
  - Weighted sum of similar objects
  - Learn projection weights
- Transformer block
  - Attention layer
  - MLP on each token/object

What is great about attention?

A freeform operator

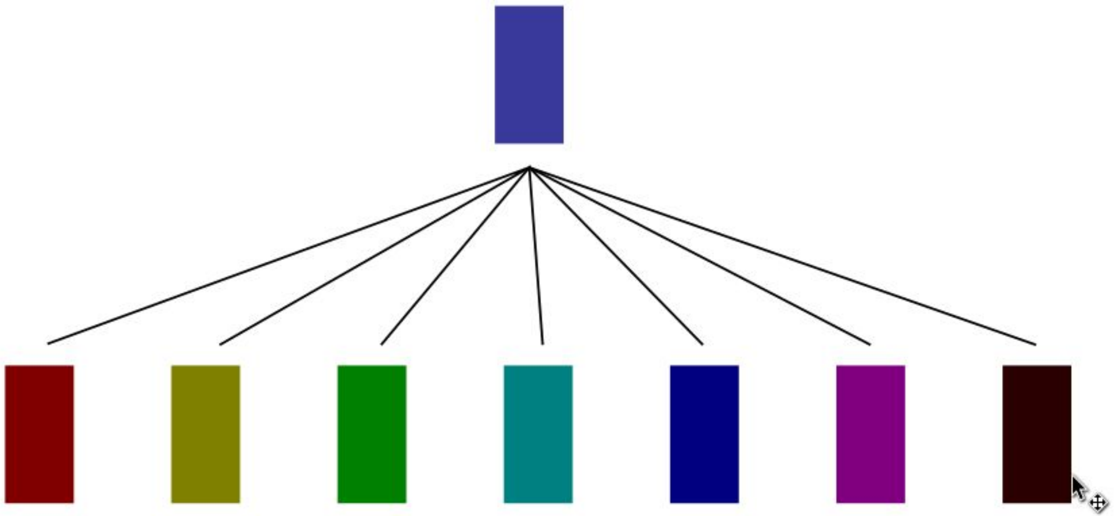
UNIVERSITÉ CÔTE D'AZUR

- Attention can implement a lot of different operations
  - Little built-in bias
    - As opposed to CNNs
  - Adapts to data
    - Can change depending on the input

- A lot of work has been done to “discover” good operators
  - To little avail
  - But attention kind of does that!

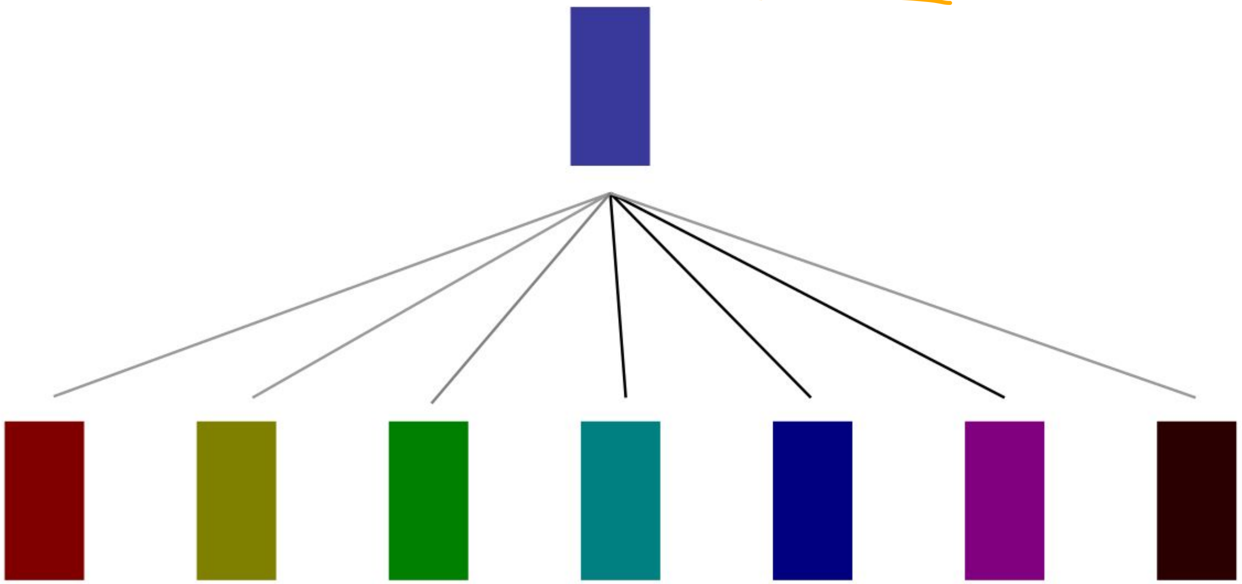
“Dense” Transformers

- Possible to attend to all elements in the sequence
- (Very) loose approximation of dense layer



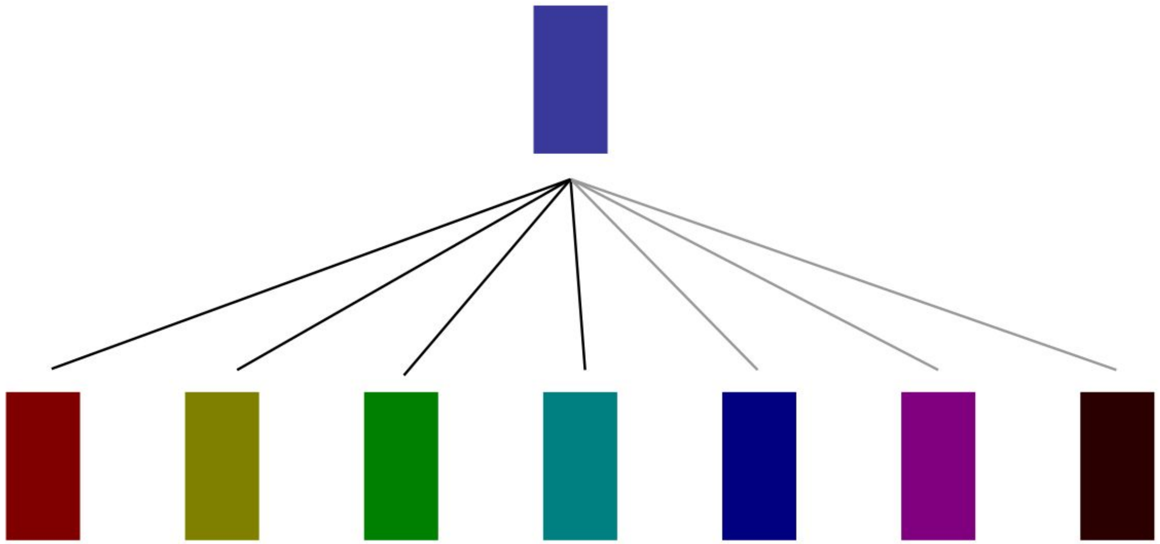
“Convolutional” Transformers

- Possible to use positional info to have convolutions
- Can be exactly approximated with ConViT (Touvron et al. 2021)



“Recurrent” Transformers

- Possible to just keep attention with past elements
- Default mode of transformer decoder



Very strong expressive power

UNIVERSITÉ CÔTE D'AZUR

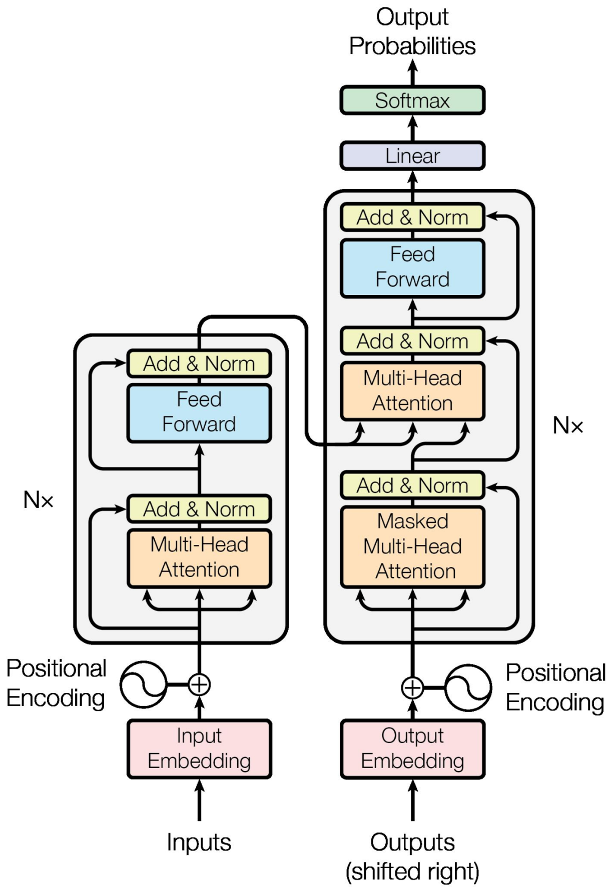
- Transformers are able to leverage large datasets
  - Because they can learn more adapted relations
  - Becoming more and more adopted
  - Scale very well

- Emerging as the dominant neural network type

- Drawbacks
  - Quadratic cost with the number of tokens
  - Need lots of data or strong regularization

Seminal success: Vaswani et al. 2017

- Completely does away with recurrent units
  - Attention as a first class citizen!
  - Introduces element wise MLP for transform
- Transformer
  - *Transforms* the input throughout the layers
  - Also to blame for BERT, ELMO, DALL-E, ...



Seminal success: Vaswani et al. 2017 Res.

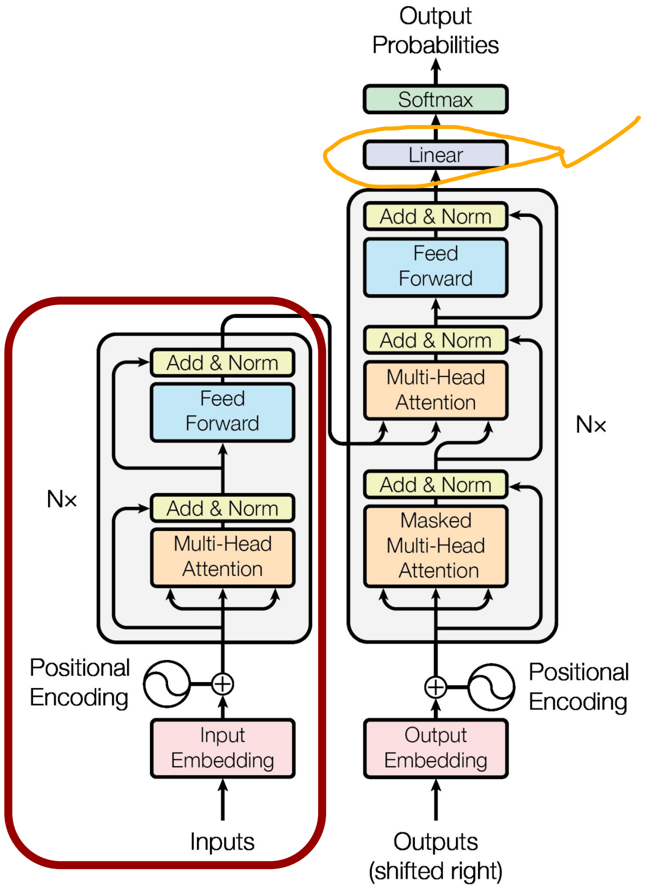
UNIVERSITÉ CÔTE D'AZUR

- Very strong results as soon as 2017!

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

A powerful encoder block

- Encoder yields strong features for the inputs
- Often used as standalone
  - No decoder
  - Linear layer or MLP directly plugged on encoder inputs
- Lots of applications!



TP3: Transformers in details

UNIVERSITÉ CÔTE D'AZUR

- Detailed explanation of all transformers components

- Implement
  - Alignment score
  - Attention block
  - Encoder block
  - Transformer predictor

- Task: Reverse a sequence
  - With attention visualization!