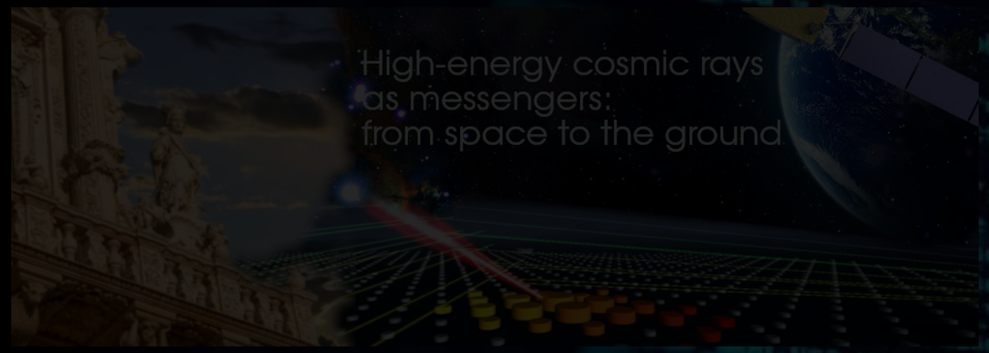


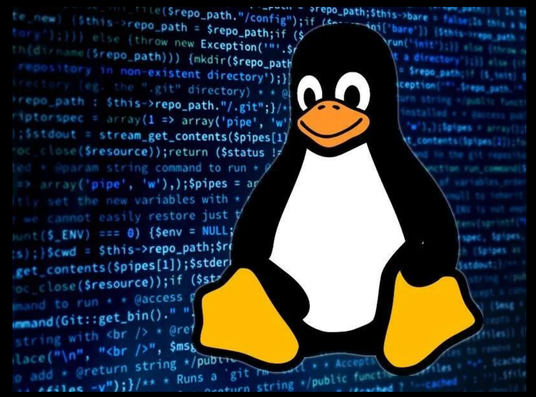
AI Hands-On

By Andrii Tykhonov [andrii.tykhonov@cern.ch]

Code: https://gitlab.cern.ch/andrii/mlregressioncalo/-/tree/tutorial

ISAPP Lecce, Italy. June 12, 2025

Preface: Installing Software



python™

ANACONDA®

TensorFlow

PyTorch

matplotlib

Operating system: Linux or MacOS

Supported and tested operating systems:

- **Linux** (currently tested on Ubuntu, but others like Debian, Fedora etc. should work just fine)

- **Mac OS**

Note on Windows — while it is not forbidden in the tutorial — it is neither tested nor fully supported, so it will be at your own risk. If you have a windows machine, it is adviced to install Linux either as a second operating system or in a virtual environment (e.g. through VirtualBox). Please contact me in advance if you have a Windows machine and never worked with Linux before.

Software prerequisite: Miniconda is (almost) all you need!

Install **Miniconda:**

- Follow the instructions in: https://www.anaconda.com/docs/getting-started/miniconda/install#macos-linux-installation

- Use **Terminal installer** (not graphical one): it allows to easily install/replace and experiment with Miniconda - everything will be placed in your home directory instead of the system one, so you will avoid potential conflicts with already pre-installed python versions etc.

- For example, in linux (basically same in MacOs but using `curl` instead of `wget`):

```
➡ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
➡ bash ~/Miniconda3-latest-Linux-x86_64.sh
```

  - ➡ You will have to agree license agreement etc (press “q” to exit license agreement in the terminal 🙂).
  - ➡ When it prompts “Choose an initialization options:” choose **YES**. If you are worried, you can make a backup of your profile initialization scripts (~/.bashrc or ~/.zshrc depending on the shell you use), but in principle all what conda does is adding one paragraph to the setup script — you can easily remove if you want to delete conda

```
➡ source ~/.bashrc
```

(or ~/.zshrc — depending on which shell you are using)

Setup ML software: Tensorflow

Throughout the tutorial we will use both **Tensorflow** and **Pytorch** frameworks. We will install those in a two separate "environments" of conda and you will be able to easily switch between the two. You will appreciate the convenience and power of Conda — it allows you to install ML software (hopefully) quickly, (hopefully) gracefully, and without experiencing library conflicts etc. (unless something goes really wrong - but we are here to help you in this case …). In the first part of the tutorial we will work with Tensorflow since (arguably) it is more simple/intuitive to use.

- Install **Tensorflow**
  1. Add `conda-forge` channel to look for software:
  2. Create new environment that we will call "tf":
  3. Activate the "tf" environment:

```
conda config --add channels conda-forge
conda create -n tf tensorflow
conda activate tf
```

Comment on activation (step 3): this has to be done in each new shell. If you want it to active the environment automatically at the start of a new shell, you can add the above command to your ~/.bashrc [or ~/.zshrc]

Note on the tensorflow versions: sometimes one needs to experiment with versions, for example I have experienced problems with latest tensorflow versions, so I downgraded to 2.16 on Ubuntu (2.15 on MacOS):

```
conda create -n tf tensorflow=2.16
```

Test Tensorflow #1 …

- Activate the Tensorflow environment in conda (see previous page for mode details):

```
conda activate tf
```

- See if Tensorflow libs are there and working:

```
python
>>> import tensorflow as tf
>>> print (tf.__version__)
2.16.XX

### NOTE USE Tensorflow version 2.15 or higher!
### If nothing crashes so far – things seem to work so far..
### First import of tensorflow on some systems (e.g. MacOS) may take
### a while - be patient (it will cache and work faster afterwards)
```

Test Tensorflow #2…

- Let's run a mock-up model training:

```
python
>>> import tensorflow as tf
>>> import numpy as np
>>> x = np.random.rand(1000,10) # random sample of 1000 sets of numbers
>>> y = np.random.rand(1000,)   # random sample of 1000 set 'labels'
>>> m = tf.keras.models.Sequential([tf.keras.layers.Dense(10)]) # trivial nonsense model
>>> l = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True) # we will learn about it later
>>> m.compile(loss=l, metrics=['accuracy'])    # compile the model
>>> m.fit(x,y,epochs=5)                        # train the model
```

- You should see something like:

```
Epoch 1/5
32/32 [==============================] - 0s 206us/step - loss: 1.5475 - accuracy: 0.0000e+00
Epoch 2/5
32/32 [==============================] - 0s 158us/step - loss: 1.2575 - accuracy: 0.0000e+00
Epoch 3/5
32/32 [==============================] - 0s 136us/step - loss: 1.0151 - accuracy: 0.0000e+00
Epoch 4/5
32/32 [==============================] - 0s 129us/step - loss: 0.8074 - accuracy: 0.0000e+00
Epoch 5/5
32/32 [==============================] - 0s 144us/step - loss: 0.6322 - accuracy: 0.0000e+00
```

If things go wrong ...

- Encountered runtime error on Ubuntu (perhaps other Linux distributives)

```
libdevice not found at ./libdevice.10.bc
         [[{{node StatefulPartitionedCall}}]] [Op:__inference_multi_step_on_iterator_579]
```

- Encountered on MacOs (apple silicon):

```
Assertion failed: (f == nullptr || dynamic_cast<To>(f) != nullptr), function down_cast, file
external/local_tsl/tsl/platform/default/casts.h, line 58.
```

— these are few typical problems I encountered myself … Unfortunately there might be more, but normally with the slight help of google, chatgpt, stackoverflow and a little prayer — things will work ;-)

Possible solutions:

1. Re-install another tenosorflow version (you may have to experiment with a few different versions):

```
conda remove -n tf --all                # delete existing tf environment from conda
conda create -n tf tensorflow=2.16      # install a specific tensorflow version (e.g. 2.16 ubuntu, or 2.15 for MacOs)
```

2. Mask out your GPU*:

```
export CUDA_VISIBLE_DEVICES=""
```

* If you you have an Nvidia GPU and the the solution 1 (re-installing different tensorflow versions) do not help, try forcing tensorflow NOT to use the GPU (it is OK for the sake of this tutorial; in the future you may tweak your software setup to fully profit of your nice GPU hardware)

Choose your weapon (text editor)

- It is perfectly fine to use your favorite python editor throughout the tutorial:

  vim, emacs, nano, eclipse+pydev, ...

- If you don't have one, for sake of simplicity and convenience – it is suggested to install **jupyter**:

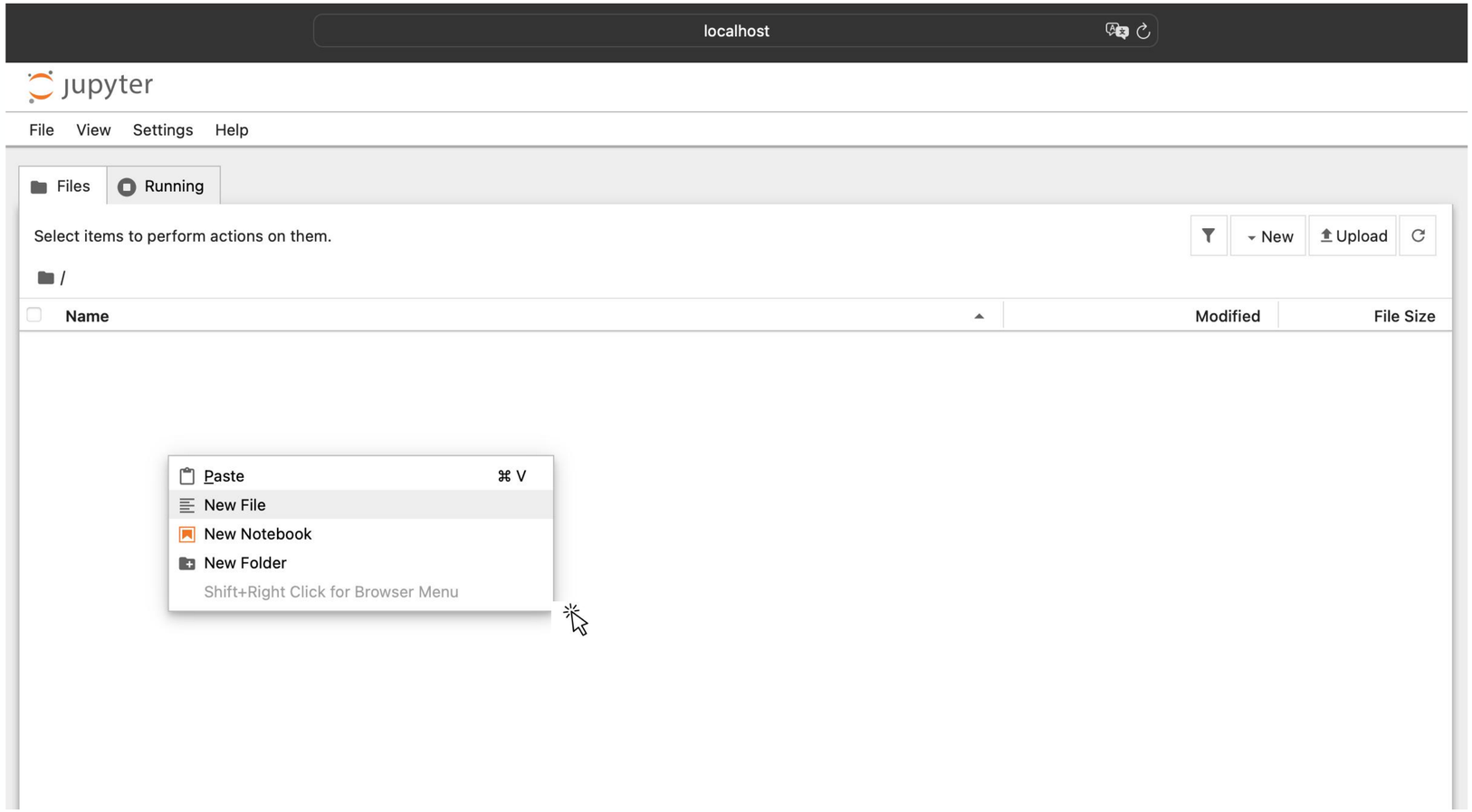
```
# open  a separate terminal window where you will run the editor
conda activate tf
conda install jupyter
```

- Create a directory for the code and run the editor there:

```
mkdir mycode
cd mycode
jupyter notebook
```

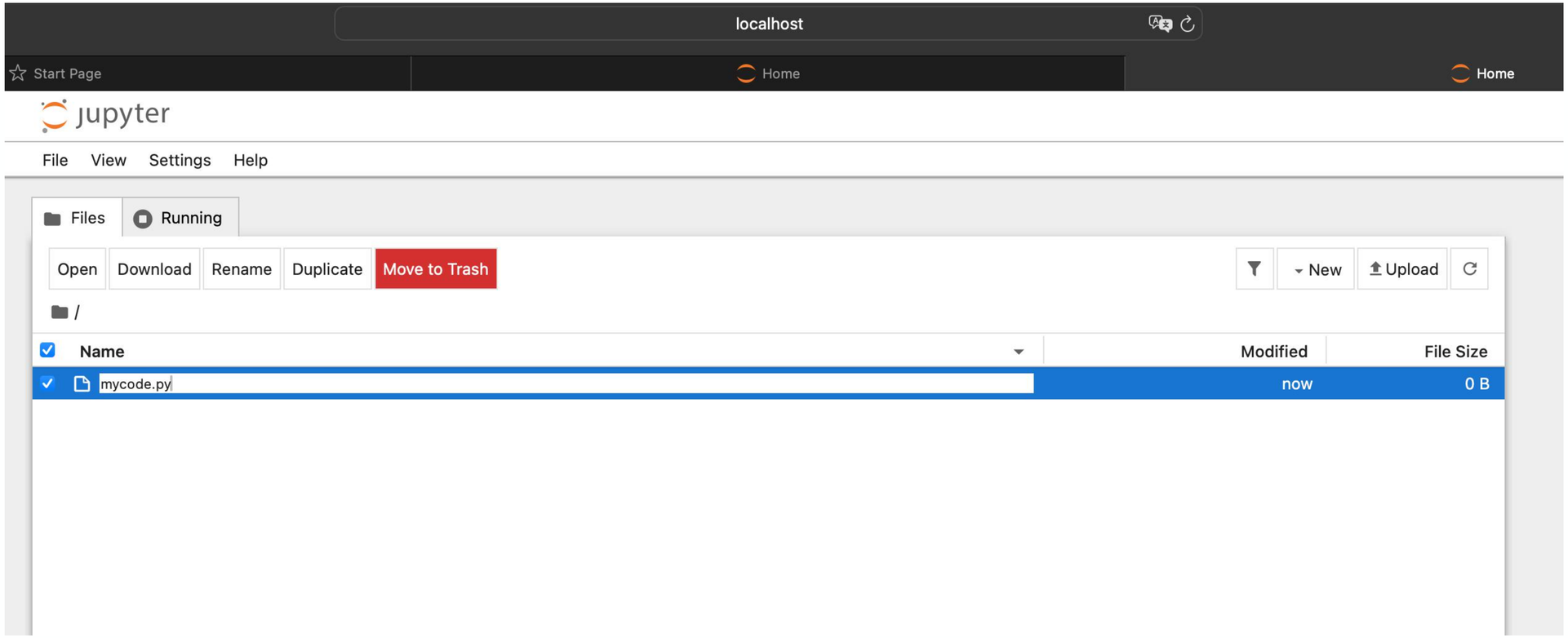
Using jupyter as text editor

- Jupyter will run a python text editor in your browser, there you can create new (click right mouse button)



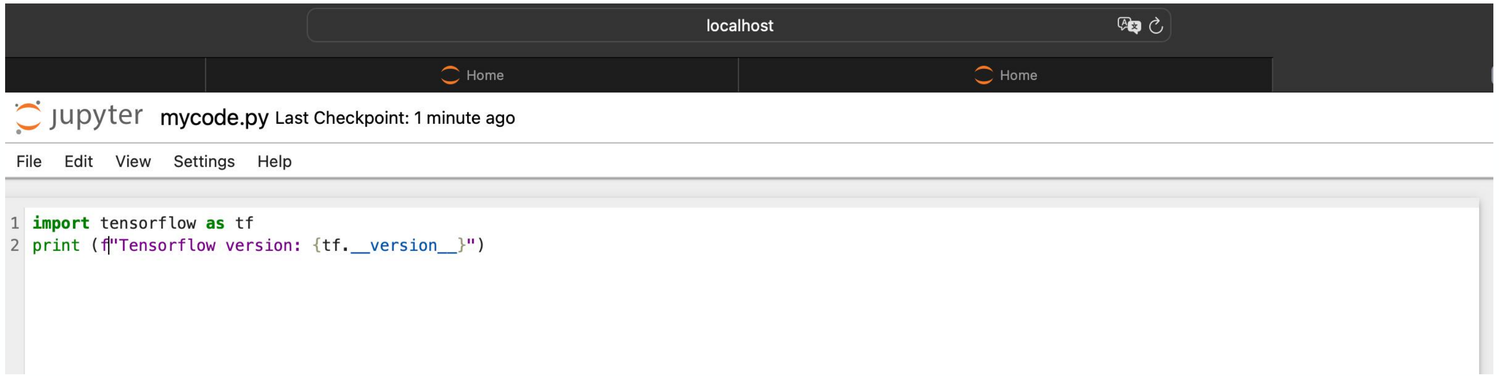
Using jupyter as text editor

- Let's call our file `mycode.py` – double click on it and you will enter the editor



Using jupyter as text editor

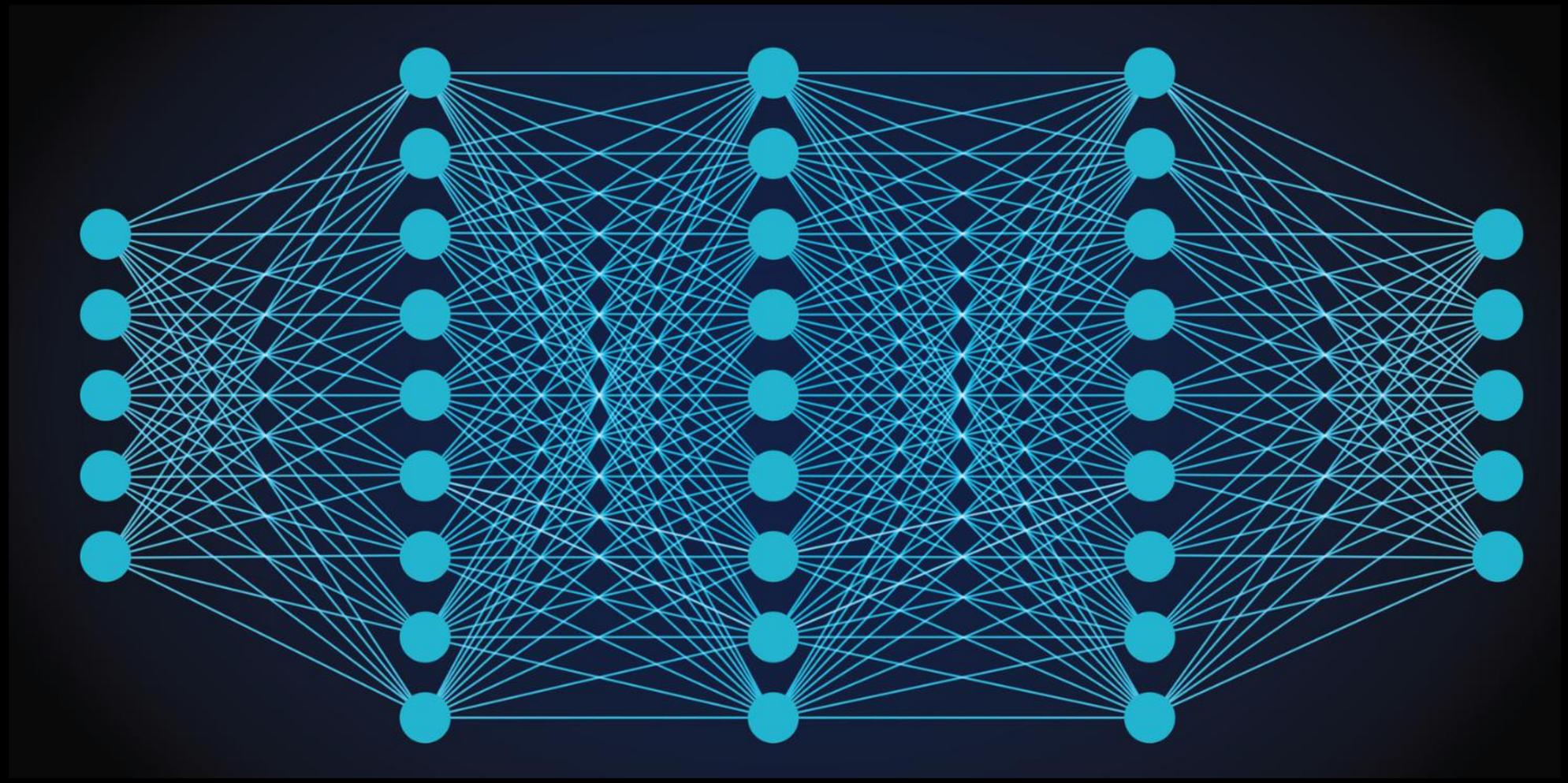
- Et voilà!



- Keep a separate terminal window to run your code (essentially you have two terminal windows, in one of those you run the editor — jupyter, in the other one — the code itself):

```
>>> python mycode.py
>>> Tensorflow version 2.16.1
```

Part I: Our first Neural Network



Adapted from: https://www.tensorflow.org/tutorials/quickstart/beginner

Our first dataset: MNIST

First we get some data that we want to train our NN ... Create a new file: `first_nn.py`

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Let's have a look inside the data…

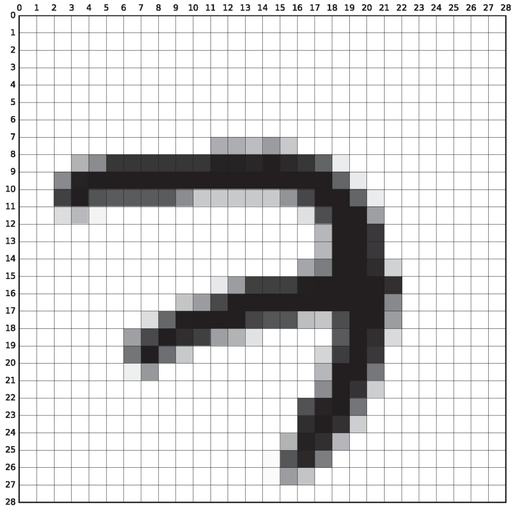
```
python -i first_nn.py
>>> x_train.shape
(60000, 28, 28) # - 60000 images of digits 28x28 pixel each
>>> y_train.shape
(60000,) >>>    # - 60000 labels corresponding to a number (from 0 to 9)
```

Reference: https://www.tensorflow.org/tutorials/quickstart/beginner

Our first dataset: MNIST

MNIST dataset *(Modified National Institute of Standards and Technology database)*:

- A database of handwritten digits in the format of 28 x 28 pixel b/w images (pixel intensity encoded in 1 byte, from 0 to 255)



(**a**) MNIST sample belonging to the digit '7'.

(**b**) 100 samples from the MNIST training set.

Our goal – develop a NN that can classify a hand-written image telling wether it corresponds to 0, 1, … or 9

Reference: doi.org/10.3390/app9153169

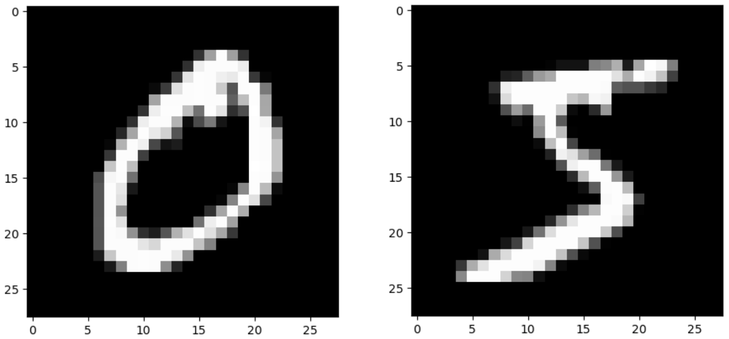
Our first dataset: MNIST

- Let's get a habit of doing visualization/debugging of our data — we will need `marplotlib` library:

```
conda install matplotlib
```

- Add visualization code to our `first_nn.py` and run it:

```
import matplotlib.pyplot as plt
...
plt.imshow(data_train[0], interpolation='none', cmap='gray')  # comment out this line afterwards
plt.show()                                                    # comment out this line afterwards
plt.imshow(data_train[1], interpolation='none', cmap='gray')  # comment out this line afterwards
plt.show()                                                    # comment out this line afterwards
```

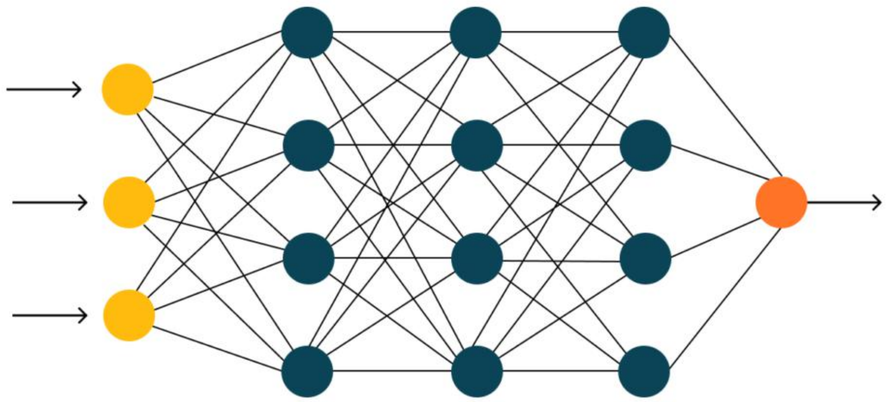


On the Neural Networks (NN) in Tensorflow & Keras...

Tensorflow is normally used with **Keras wrapper API** (comes as a part of tensorflow installation) which allows to create and manipulate Neural Networks in intuitive way, composed of “layers” stacked one after another

`tf.keras.layers.Dense(4,...)` `...` `...`

`tf.keras.layers.Input(3,..)`



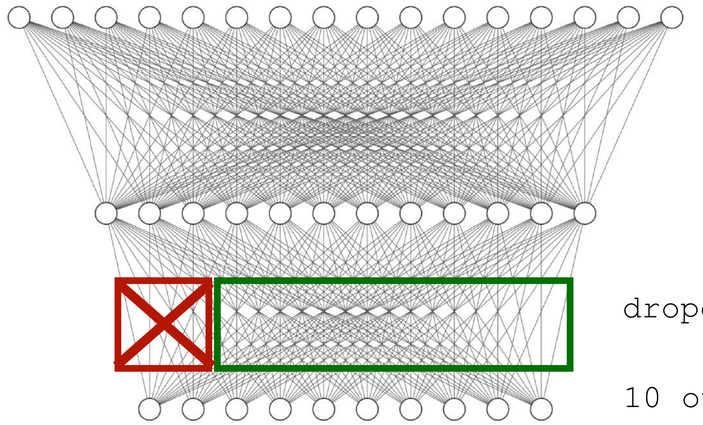
`tf.keras.layers.Dense(1,…)`

Image: qtravel.ai

Constructing our first model (NN) in Tensorflow

Add NN model to our `first_nn.py`:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)), # First we convert input image into a flat array of numbers
  tf.keras.layers.Dense(128, activation='relu'), # Next, we add a layer of 128 neurons
  tf.keras.layers.Dropout(0.2),                  # Dropout randomly removes 20% neurons from the above layer
  tf.keras.layers.Dense(10)                      # Final layer will correspond to 10 probabilities
])
```



Input image converted into a flat array

128 neurons (activation function Relu)

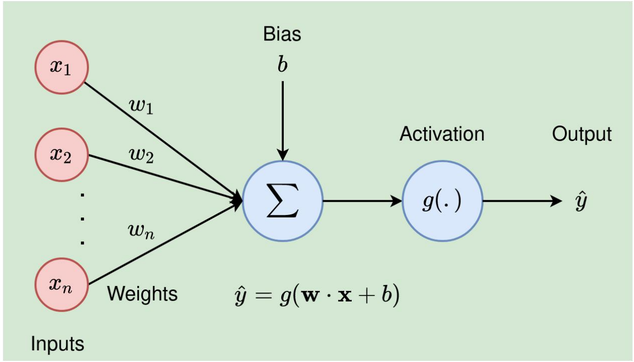
dropout - randomly removes 20% (0.2) of neurons from the above layer during training

10 outputs — will be converted into probabilities of an image representing 0,1,2,...,9

Let’s inspect our model (NN)


```
>>> python -i first_nn.py
>>> model.summary()
```

Remember neuron structure & parameters:



Model: "sequential"

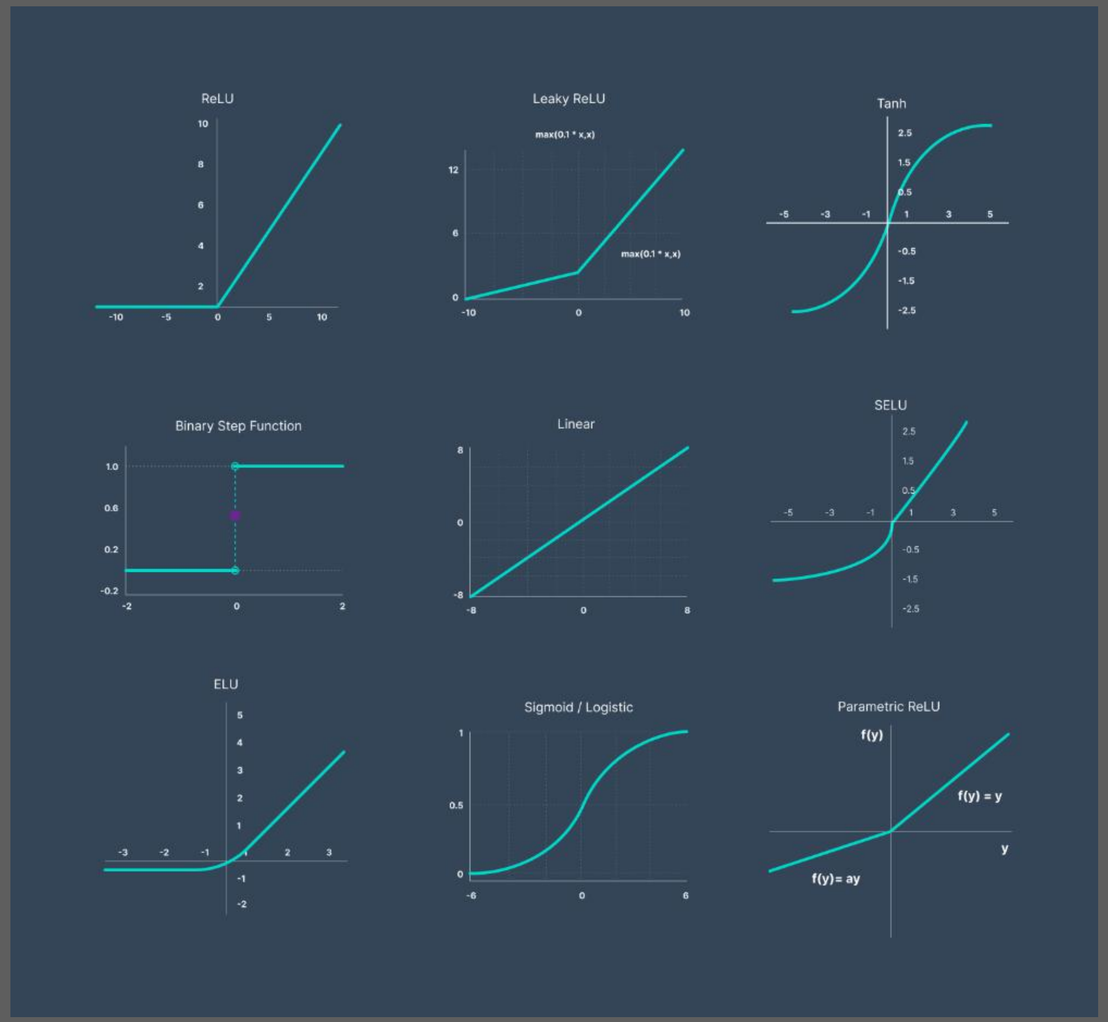
Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

?

?

Total params: 101,770 (397.54 KB)
Trainable params: 101,770 (397.54 KB)
Non-trainable params: 0 (0.00 B)

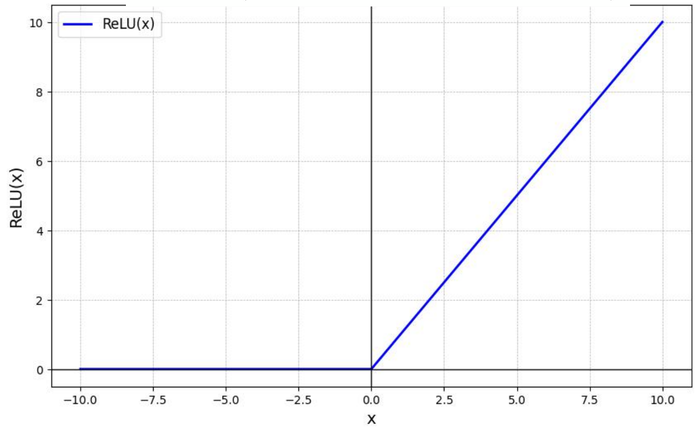
Side note on activation functions ...



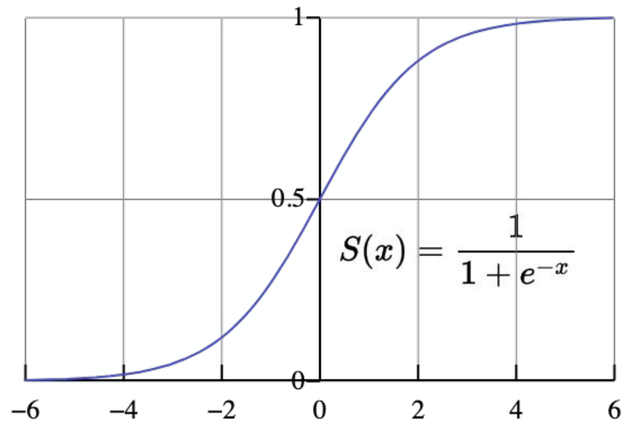
Why do we need activation functions?

- Primary reason for activation functions is to introduce a **non-linearity** in the model, otherwise the output will be a simple linear combination of inputs!
- Activations are also required in classification tasks in order to convert continue outputs into contained [0,1] probabilities

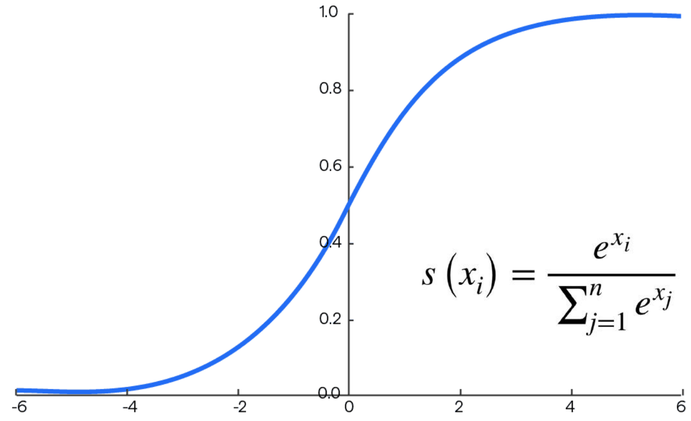
ReLU (Rectified Linear Unit)



Sigmoid

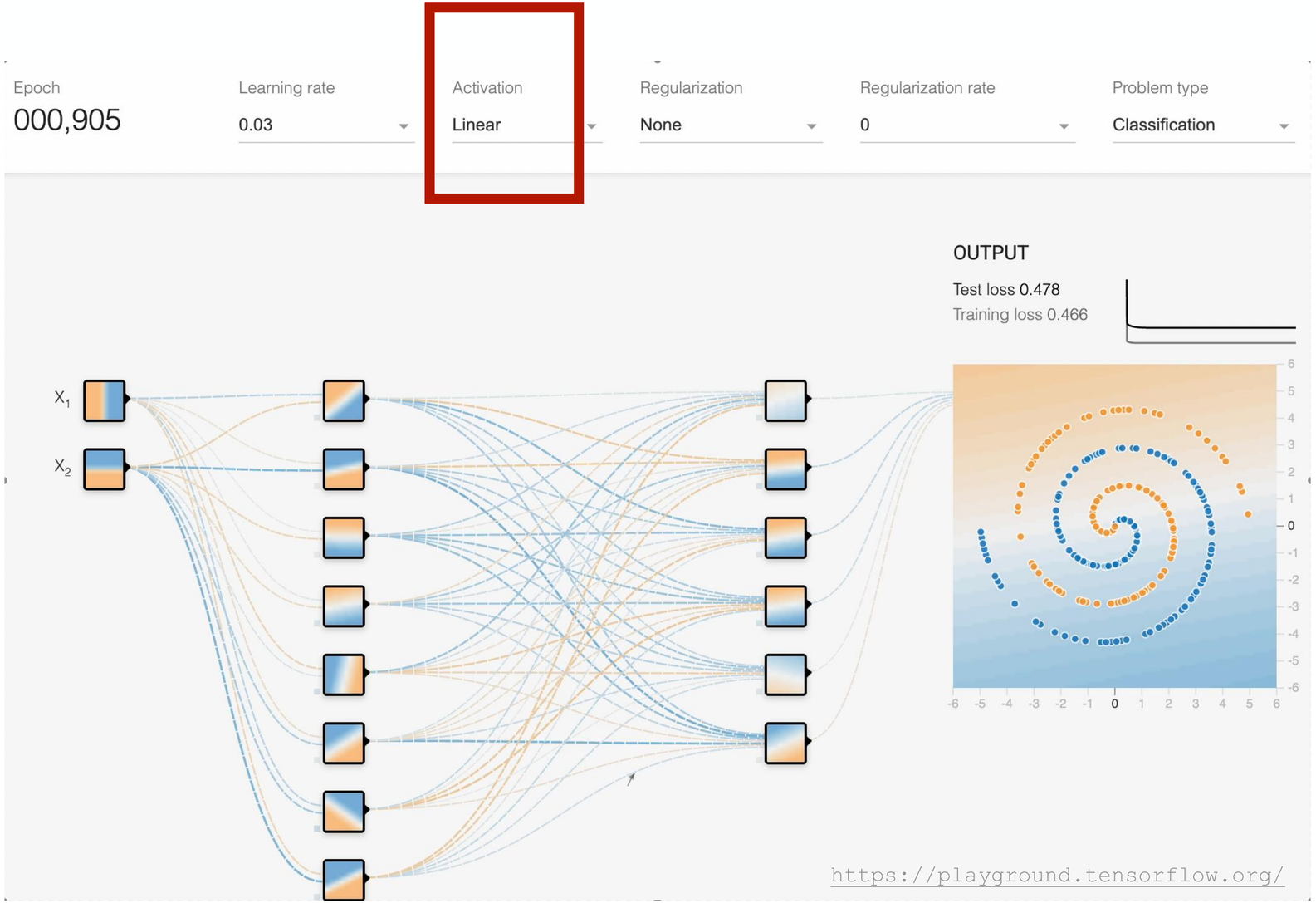


Softmax



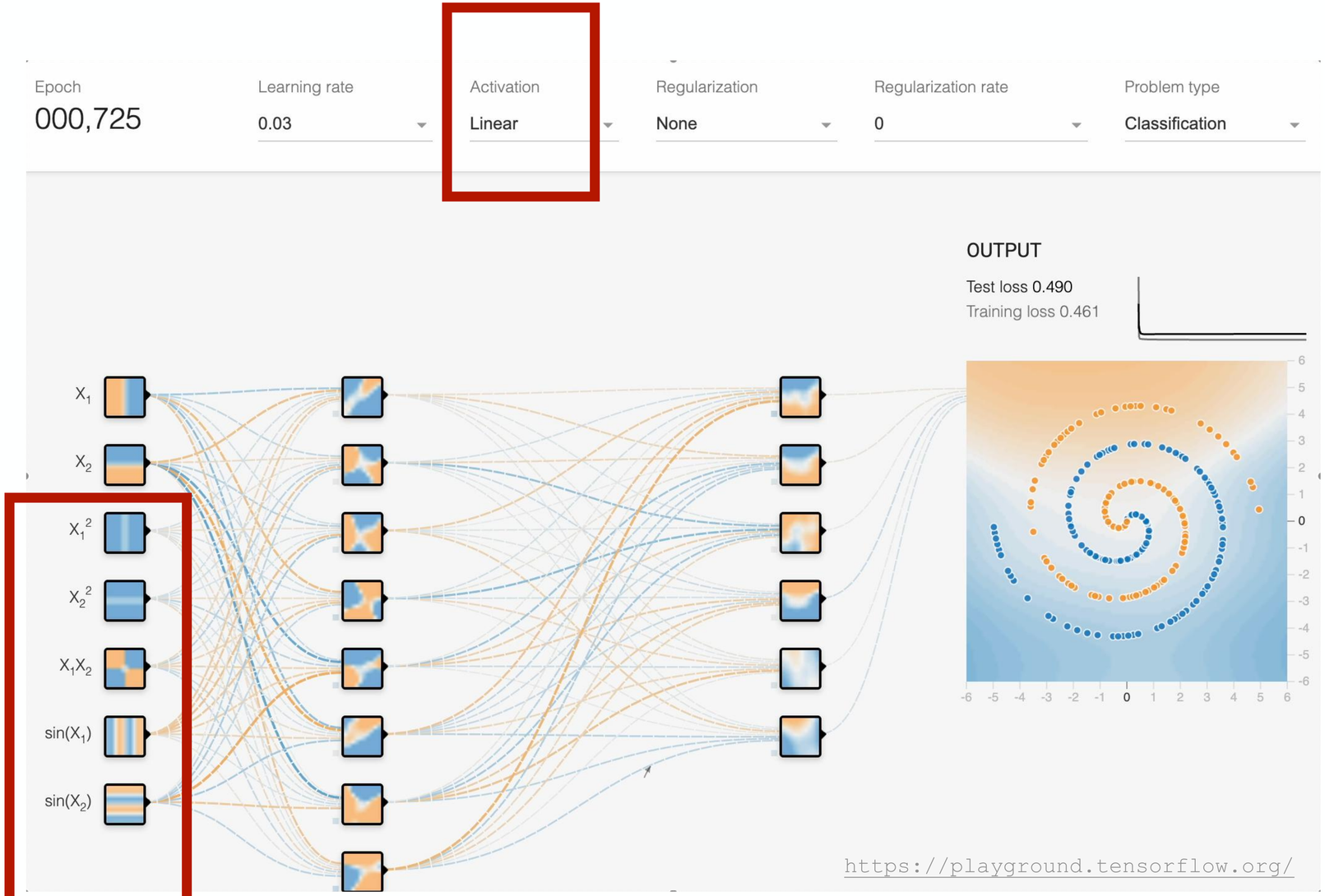
And there are more (leaky ReLU, tanh etc.)

Example of NN without activation function



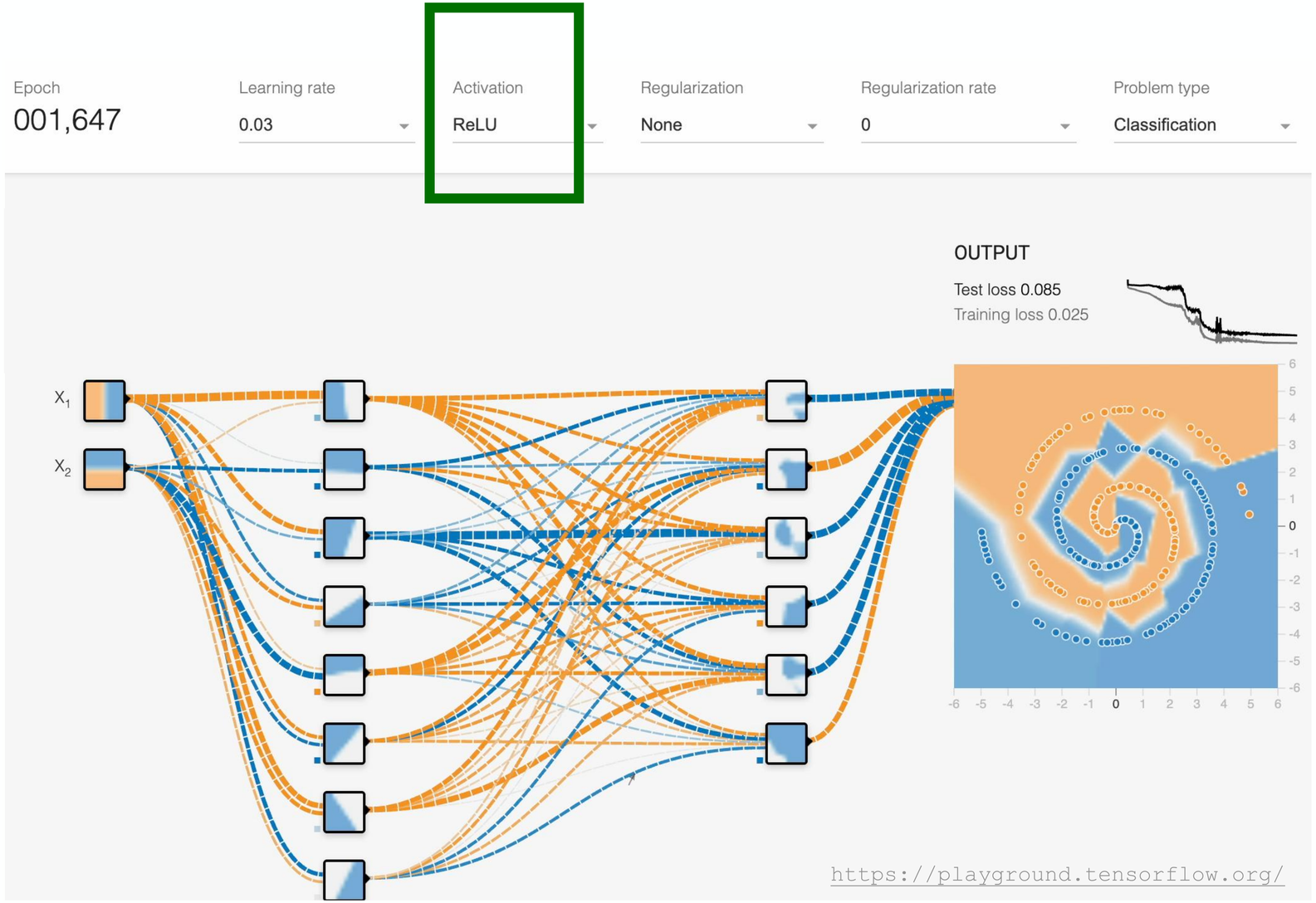
In this example model is not able to learn the data representation (identify two classes of orange and blue points) ...

Example of NN without activation function



… adding non-linear function of inputs as additional inputs does not help …

Activation function (ReLU) added

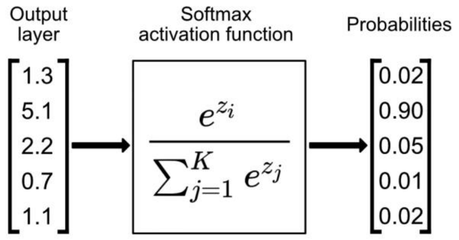


Now getting back to our code…

What does our model do?

Try yourself:

```
python -i first_nn.py
>>> predictions = model(x_train[:1]).numpy() # let's process a first training image with our model
>>> print (predictions)

[[ 0.37128526 -0.10725151  0.18021962  0.10659367  0.20349179  0.00092683
  -0.45577508 -0.23256837  0.02991931 -0.14916359]]

# ... these do not look like probabilities
# To interpret model output as probabilities we process the output with Softmax function:
```




```
>>> predictions = tf.nn.softmax(predictions).numpy()
>>> print (predictions)

[[0.11647741, 0.05115878, 0.09075072, 0.15573394, 0.10448843,
        0.07079597, 0.05859897, 0.12428293, 0.14535967, 0.08235319]]

# ... now it looks like probabilities, but they are clearly wrong, because we need to train our model first!
```

Adding loss function and compiling the model

Add to `first_nn.py`:

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
# Note that from_logits=True informs the loss function that the model output is not reduced
# to probabilities [0 to 1], hence the loss will apply softmax function to the model output

# example of loss calculation
predictions = model(x_train[:1]).numpy()                      # comment me out later
print ('loss=',loss_fn(y_train[:1], predictions).numpy())     # comment me out later

model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])                  # accuracy = N_correct_guesses / N_total
```

Note on the loss function

Categorial cross-entropy:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

C – number of classes (10 in our case)

y_i – true label for class i (either 1 or 0)

$\hat{y}_i$ – predicted label for class i (in the 0 to 1 range)

Try yourself:

```
>>> from tensorflow import keras
>>> import numpy as np
>>> y_true = np.array([1, 2])
>>> y_pred = np.array([[0.05, 0.95, 0], [0.1, 0.8, 0.1]])
>>> scce = keras.losses.SparseCategoricalCrossentropy()
>>> print (scce(y_true, y_pred))
```

Space Categorial cross-entropy:

$$\text{Loss} = -\log(\hat{y}_c)$$

c – correct class index (from 0 to 9 in our case)

$\hat{y}_c$ – predicted label for class c

Try yourself:

```
>>> from tensorflow import keras
>>> import numpy as np
>>> y_true = np.array([[0,1,0], [0,0,1]])
>>> y_pred = np.array([[0.05, 0.95, 0], [0.1, 0.8, 0.1]])
>>> cce = keras.losses.CategoricalCrossentropy()
>>> print (cce(y_true, y_pred))
```

The two examples correspond to identical cases, the only difference is in the format of the true labels!

Training and testing the model

Add training part to the `first_nn.py` and run it:

```
history = model.fit(x_train, y_train, epochs=5)
```

.. you will get something like:

```
Epoch 1/5
1875/1875 [==============================] - 1s 339us/step - loss: 0.3030 - accuracy: 0.9123
Epoch 2/5
1875/1875 [==============================] - 1s 333us/step - loss: 0.1431 - accuracy: 0.9577
Epoch 3/5
1875/1875 [==============================] - 1s 464us/step - loss: 0.1061 - accuracy: 0.9679
Epoch 4/5
1875/1875 [==============================] - 1s 327us/step - loss: 0.0880 - accuracy: 0.9728
Epoch 5/5
1875/1875 [==============================] - 1s 338us/step - loss: 0.0750 - accuracy: 0.9768
...
```

Add testing part:

```
model.evaluate(x_test,  y_test, verbose=2)
```

.. you will get something like:

```
313/313 - 0s - loss: 0.0760 - accuracy: 0.9762
```

That's it - we trained our first NN model!

Recap

first_nn.py

```
import tensorflow as tf
#import matplotlib.pyplot as plt

# get the dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# example of input data
#plt.imshow(x_train[0], interpolation='none',cmap='gray')  # comment me out later
#plt.show()                                                # comment me out later
#plt.imshow(x_train[1], interpolation='none',cmap='gray')  # comment me out later
#plt.show()                                                # comment me out later

# create the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)), # First we convert input image into a flat array of numbers
  tf.keras.layers.Dense(128, activation='relu'), # Next, we add a layer of 128 neurons
  tf.keras.layers.Dropout(0.2),                  # Dropout randomly removes 20% neurons from the above layer
  tf.keras.layers.Dense(10)                      # Final layer will correspond to 10 probabilities
])

# example of how model process the intput data
#predictions = model(x_train[:1]).numpy()
#print (predictions)

# define the loss function
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# example of loss calculation                              # comment me out later
#predictions = model(x_train[:1]).numpy()                  # comment me out later
#print ('loss=',loss_fn(y_train[:1], predictions).numpy())  # comment me out later

# compile the model
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy']) # accuracy = N_correct_guesses / N_total

# train the model
history = model.fit(x_train, y_train, epochs=5)

# test the model
model.evaluate(x_test,  y_test, verbose=2)
```

Recap (remove non-essential commented parts)


```
import tensorflow as tf

# get the dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# create the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)), # First we convert input image into a flat array of numbers
  tf.keras.layers.Dense(128, activation='relu'), # Next, we add a layer of 128 neurons
  tf.keras.layers.Dropout(0.2),                  # Dropout randomly removes 20% neurons from the above layer
  tf.keras.layers.Dense(10)                      # Final layer will correspond to 10 probabilities
])

# define the loss function
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# compile the model
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])

# train the model
history = model.fit(x_train, y_train, epochs=5)

# test the model
model.evaluate(x_test,  y_test, verbose=2)
```

Recap (remove non-essential commented parts)


```
import tensorflow as tf

# get the dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# create the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)), # First we convert input image into a flat array of numbers
  tf.keras.layers.Dense(128, activation='relu'), # Next, we add a layer of 128 neurons
  tf.keras.layers.Dropout(0.2),                  # Dropout randomly removes 20% neurons from the above layer
  tf.keras.layers.Dense(10)                      # Final layer will correspond to 10 probabilities
])

# define the loss function
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# compile the model
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])

# train the model
history = model.fit(x_train, y_train, epochs=5)

# test the model
model.evaluate(x_test,  y_test, verbose=2)
```

This is the core part of tensorlfow (or any other ML framework) – when the model is built, tensorflow becomes aware of its trainable parameters (wights and bias values of every neuron in every layer)

Recap (remove non-essential commented parts)


```
import tensorflow as tf

# get the dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

This is the core part of tensorlfow (or any other ML framework) – when the model is built, tensorflow becomes aware of its trainable parameters (wights and bias values of every neuron in every layers)

```
# create the model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)), # First we convert input image into a flat array of numbers
  tf.keras.layers.Dense(128, activation='relu'), # Next, we add a layer of 128 neurons
  tf.keras.layers.Dropout(0.2),                  # Dropout randomly removes 20% neurons from the above layer
  tf.keras.layers.Dense(10)                      # Final layer will correspond to 10 probabilities
])

# define the loss function
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# compile the model
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

The partial derivatives w.r.t. trainable parameters are computed during the execution of “fit” given the input data. The parameters are updated in every “epoch” following the gradient descent method (e.g. “adam” is one of the most common types of gradient descent algorithms)

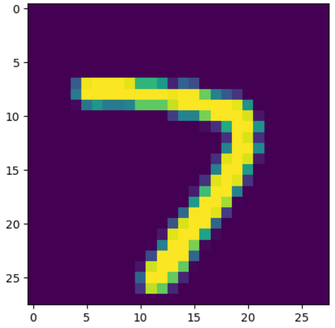
```
# train the model
history = model.fit(x_train, y_train, epochs=5)

# test the model
model.evaluate(x_test,  y_test, verbose=2)
```

Fun part: let's run some predictions!

First, remember that our model produces a set of 10 numbers as the input, however they are not constrained to [0,1] range since the constraining part (Softmax activation) was included in the loss function, but not in the model itself.

Hence, we need to convert our 10 number into 10 probabilities, in `first_nn.py` add:

```
probability_model = tf.keras.Sequential([
  model,
  tf.keras.layers.Softmax()
])
```



Now let's run the predictions:

```
python -i first_nn.py
>>> import numpy as np                                  # we need numpy for some array manipulations
>>> import matplotlib.pyplot as plt
>>> plt.imshow(x_test[17], interpolation='none')        # pick some random image from the data
>>> plt.show()                                         # let's first look at that image ourselves
>>> model_guss = probability_model(x_test[17:18])      # run the prediction
>>> print ("The guessed number is:", np.argmax(model_guss))  # ??? Did it guess?
```

Try with the other images and see for yourself how good (or maybe not) our model is? :-)

Don't close the console with the `first_nn.py` yet! (See next slide...)

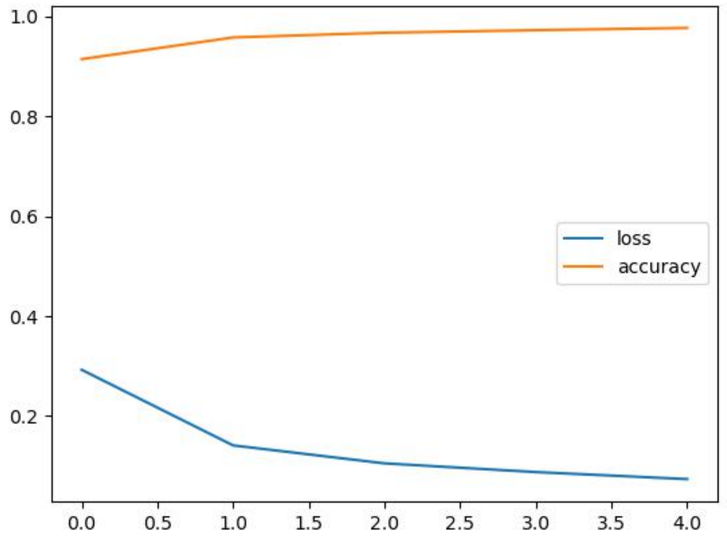
Visualizing the training process

Don't close your console with `first_nn.py` yet, let's examine the `history` object:

```
>>> print (history.history)
{'loss': [0.29248857498168945, 0.1410820633172989, 0.10512221604585648, 0.08767301589250565,
0.07357344776391983], 'accuracy': [0.914900004863739, 0.9583333134651184, 0.9677833318710327,
0.9729166626930237, 0.9773499965667725]}
```

Now let's do some plots:

```
>>> plt.plot(history.history['loss'])
>>> plt.plot(history.history['accuracy'])
>>> plt.legend(['loss', 'accuracy'])
>>> plt.show()
```



Let’s complicate things a bit ...

In your `first_nn.py` modify the `fit` function to the following and re-run the code:

```
history = model.fit(x_train, y_train, epochs=50, validation_split=0.1)
```

Now let’s inspect the `history` object again and plot the numbers there:

```
>>> print (history.history.keys())
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

>>> plt.plot(history.history['loss'])
>>> plt.plot(history.history['val_loss'])
>>> plt.plot(history.history['accuracy'])
>>> plt.plot(history.history['val_accuracy'])
>>> plt.legend(['loss','val_loss','accuracy','val_accuracy'])
>>> plt.show()
```

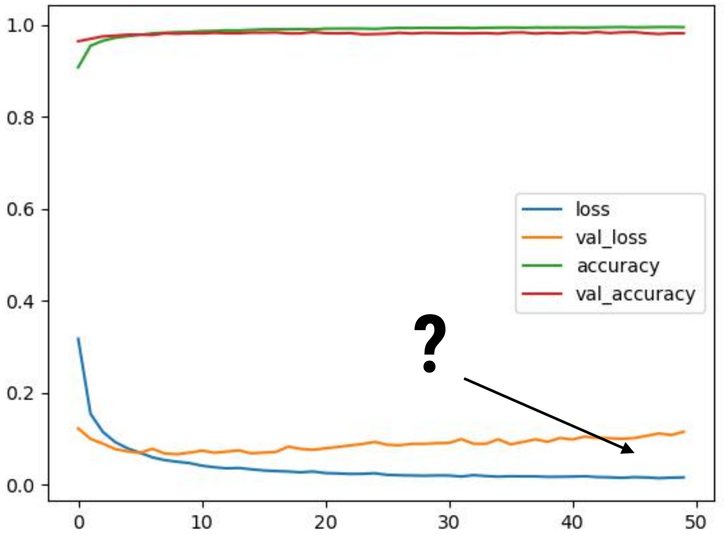
Let's complicate things a bit (adding validation)...

In your `first_nn.py` modify the `fit` function to the following and re-run the code:

```
history = model.fit(x_train, y_train, epochs=50, validation_split=0.1)
```

Now let's inspect the `history` object again and plot the numbers there:

```
>>> print (history.history.keys())
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

>>> plt.plot(history.history['loss'])
>>> plt.plot(history.history['val_loss'])
>>> plt.plot(history.history['accuracy'])
>>> plt.plot(history.history['val_accuracy'])
>>> plt.legend(['loss','val_loss','accuracy','val_accuracy'])
>>> plt.show()
```



Try to experiment a bit, consider modifying model "horizontally" or "vertically" (adding/removing layers, number of neurons etc.). Can we arrive with a better (more accurate model) that, at the same time, does not overfit? ...

Put our code in order

For convenience of further work, let's re-structure our code a bit:

```
cp first_nn.py run_fit.py
```

In `run_fit.py`:

```
from model_nn import model                    # <-- add this in the beginning
...
#model = tf.keras.models.Sequential([...]) # <-- remove or comment out this part
```

Create an empty `__init__.py` along with the following `model_nn.py`:

```
import tensorflow as tf

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

We didn't do anything new, we just re-structured the code for easier manipulation with difference models. If you run `run_fit.py` you will get the same result as with `first_nn.py`

Another way of defining Tensorlfow (TF) model

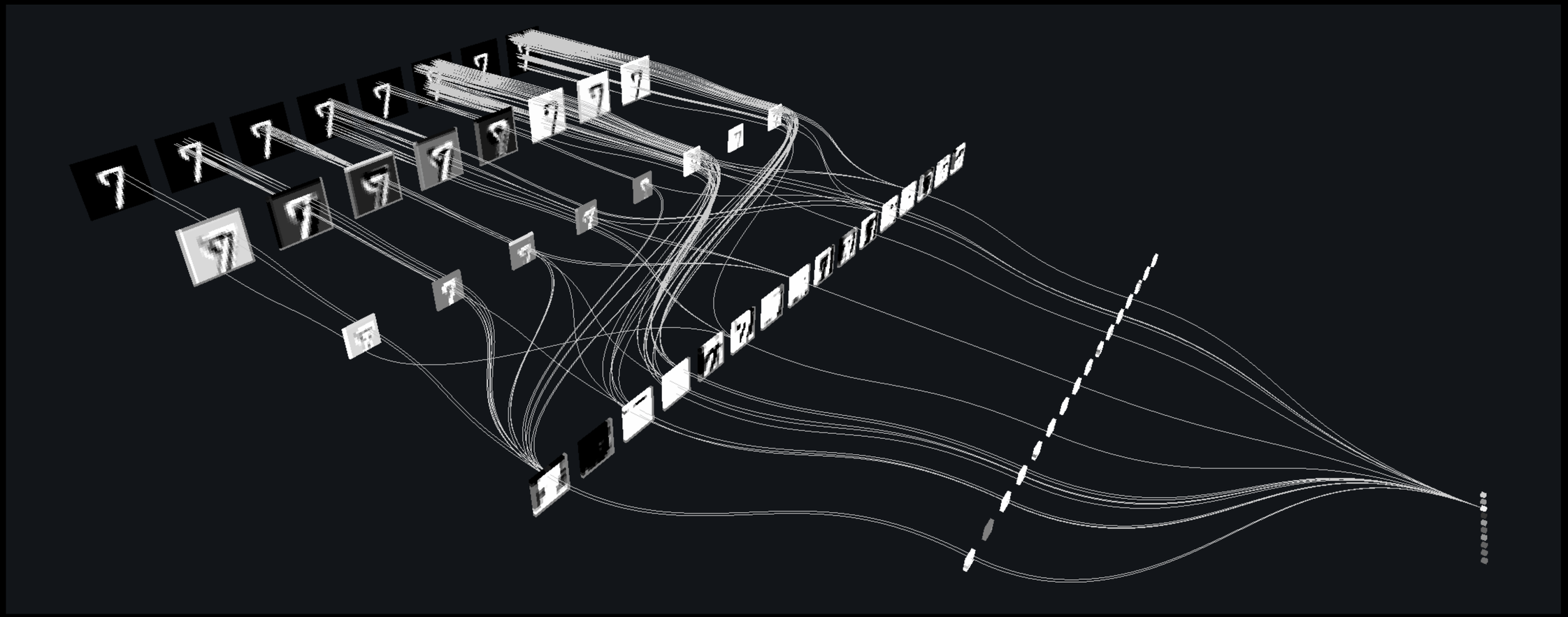
Let's rewrite our NN model in slightly different format (it will get useful for us later ...). Create **`model_nn_new.py:`**

```
import tensorflow as tf

input_ = tf.keras.Input(shape=(28 , 28))

layer = tf.keras.layers.Flatten()(input_)
layer = tf.keras.layers.Dense(128, activation='relu')(layer)
layer = tf.keras.layers.Dropout(0.2)(layer)
layer = tf.keras.layers.Dense(10)(layer)

model = tf.keras.Model(inputs=input_,outputs=layer)
```

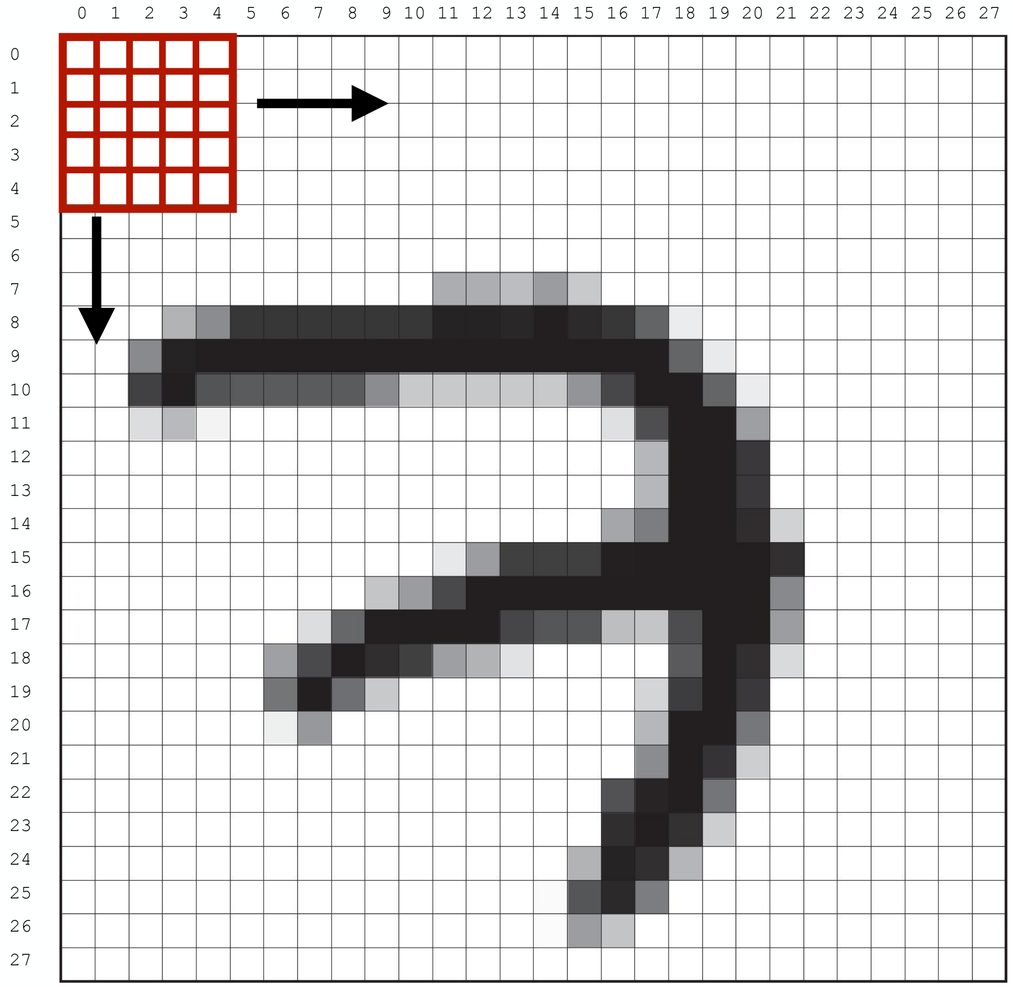
- In the `run_fit.py` file: replace `model_nn` with `model_nn_new` and run the `run_fit.py` again ...

  ➡While nothing really changed in what the code does, the new format/definition of the model will come in hand while working with CNNs, in particular for understanding the dimensionality of tensors etc.

Part II: Convolutional Neural Networks



What are Convolutional Neural Networks?



Convolution is essentially a filter that slides through the image producing one output value per each position

In the example of 5x5 convolution filter, at every position the output value will be:

$$O(i,j) = \sum_{m=0}^{4} \sum_{n=0}^{4} K(m,n) \cdot I(i+m, j+n) + b$$

$O(i,j)$ – output value at filter position (i,j) – in this example from 0 to 23

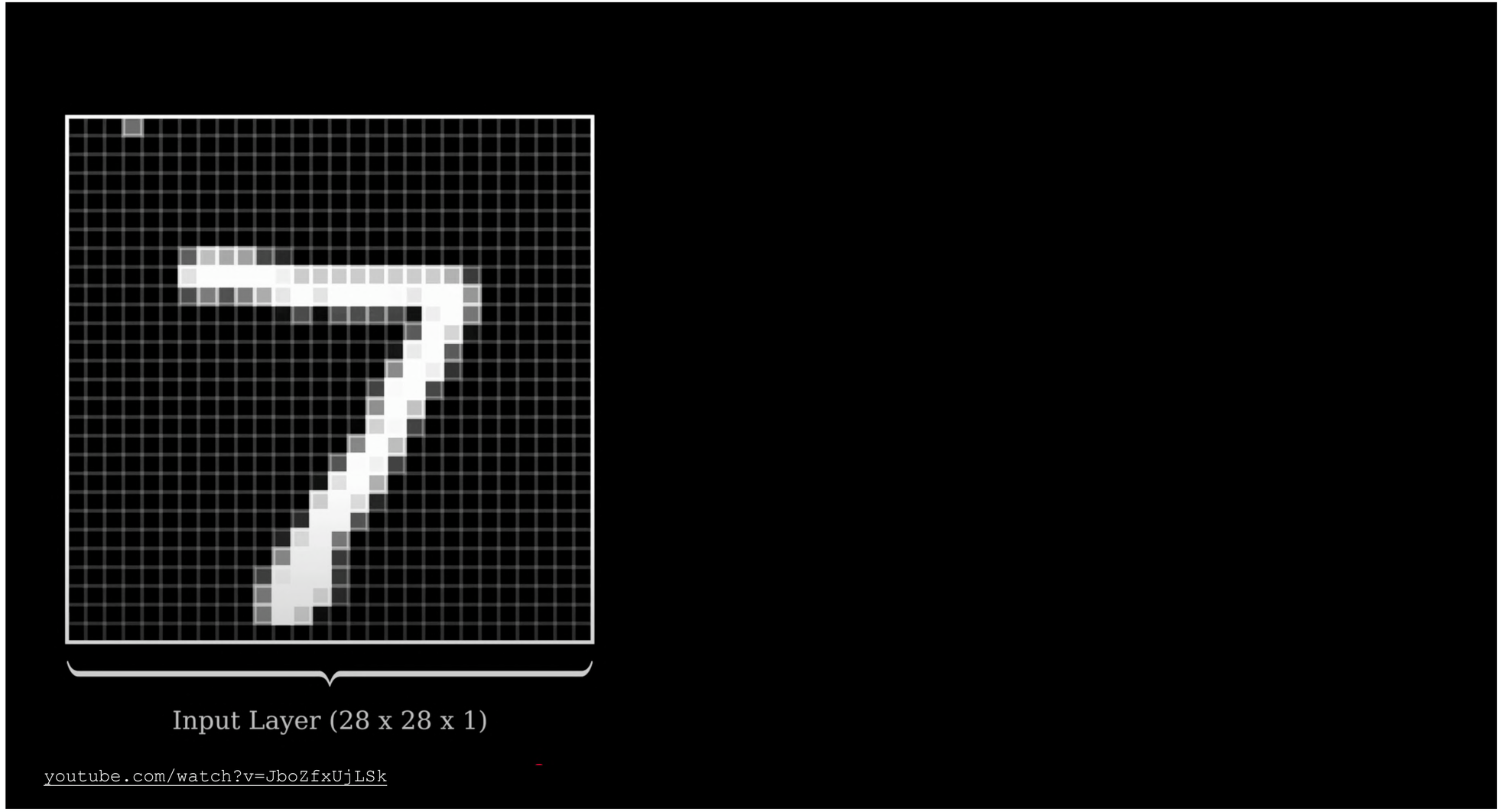
$K(m,n)$ – the convolutional kernel value at row m and column n (from 0 to 4)

$I(i+m, j+n)$ – the corresponding pixel value being convolved

b – bias term

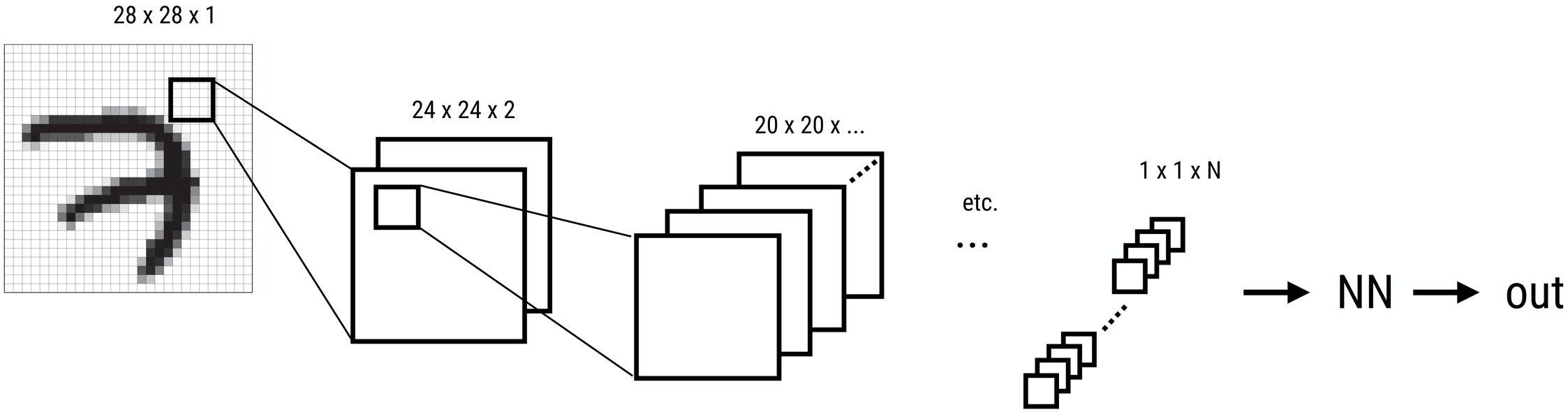
What are Convolutional Neural Networks?

With the 5x5 filter, the output will be another picture of dimension 24 x 24. Now imagine you have two such filters, so that the two output images will be produced:



What are Convolutional Neural Networks?

With the 5x5 filter, the output will be another picture of dimension 24 x 24. Now imagine you have two such filters, so that the two output images will be produced:



- We apply convolutional layers one after another allowing the model to learn some deep features in the data
- The 2D dimensionality necessarily decreases layer after layer
- The 3rd dimension equals the number of Conv filters at every step (usually is set to increase layer after layer, but not necessarily)
- Normally after the last layer will will end up with 1x1xN images, i.e. N numbers which we can then process with the usual NN

CNN with our MNIST example

Let's get back our number classification example: copy `model_nn_new.py` to `model_cnn.py` and let's do some changes there:

```
import tensorflow as tf

#input_ = tf.keras.Input(shape=(28,28))
input_ = tf.keras.Input(shape=(28,28,1))# <-- third dimension specify number of channels in the image
                                        # <-- for example, b/w has only 1 channel, colored has 3 (RGB)

#layer = tf.keras.layers.Flatten()(input_) # <-- we don't need to flatten at this point...

# convolutional part
layer = tf.keras.layers.Conv2D(32,(4,4),strides=(4,4), activation='relu')(input_)
layer = tf.keras.layers.Conv2D(64,(7,7),strides=(1,1), activation='relu')(layer)
layer = tf.keras.layers.Flatten()(layer)

# usual NN part remains the same
layer = tf.keras.layers.Dense(128, activation='relu')(layer)
layer = tf.keras.layers.Dropout(0.2)(layer)
layer = tf.keras.layers.Dense(10)(layer)

model = tf.keras.Model(inputs=input_,outputs=layer)
```

CNN with our MNIST example

Let's get back our number classification example: copy `model_nn_new.py` rewrite `model_cnn.py` and let's do some changes:

```
import tensorflow as tf

#input_ = tf.keras.Input(shape=(28,28))
input_ = tf.keras.Input(shape=(28,28,1))# <-- third dimension specify number of channels in the image
                                        # <-- for example, b/w has only 1 channel, colored has 3 (RGB)

#layer = tf.keras.layers.Flatten()(input_) # <-- we don't need to flatten at this point...

# convolutional part
layer = tf.keras.layers.Conv2D(32,(4,4),strides=(4,4), activation='relu')(input_)
print('Conv layer 1 shape:', layer.shape)
layer = tf.keras.layers.Conv2D(64,(7,7),strides=(1,1), activation='relu')(layer)
print('Conv layer 2 shape:', layer.shape)
layer = tf.keras.layers.Flatten()(layer)

# usual NN part remains the same
layer = tf.keras.layers.Dense(128, activation='relu')(layer)
layer = tf.keras.layers.Dropout(0.2)(layer)
layer = tf.keras.layers.Dense(10)(layer)

model = tf.keras.Model(inputs=input_,outputs=layer)
```

?

Let's add some print statements to look inside our model. In this way it is very easy to experiment with layers, for example if you are not sure what the layer does etc…

```
python model_cnn.py
```

Check the output for yourself and let's see if we understand it ...

CNN with our MNIST example

Open `run_fit.py` and replace the NN with the CNN; re-run the training:

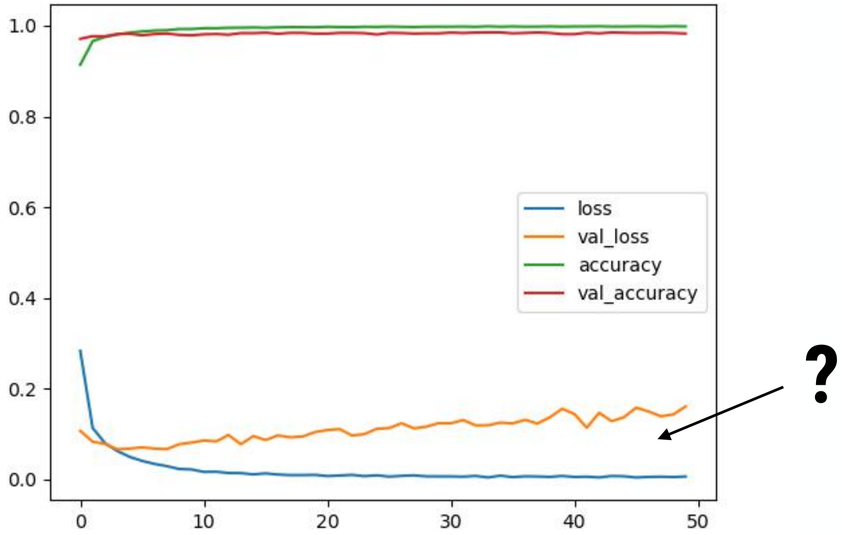
```
#from model_nn_new import model  # comment out the NN
from model_cnn import model      # import CNN instead
```

```
python run_fit.py
```

```
Epoch 1/50
1688/1688 [==============================] - 1s 789us/step - loss: 0.2831 - accuracy: 0.9139 - val_loss: 0.1069 - val_accuracy: 0.9710
Epoch 2/50
1688/1688 [==============================] - 1s 765us/step - loss: 0.1134 - accuracy: 0.9660 - val_loss: 0.0829 - val_accuracy: 0.9768
Epoch 3/50
1688/1688 [==============================] - 1s 761us/step - loss: 0.0798 - accuracy: 0.9753 - val_loss: 0.0782 - val_accuracy: 0.9765
Epoch 4/50
1688/1688 [==============================] - 1s 762us/step - loss: 0.0624 - accuracy: 0.9807 - val_loss: 0.0663 - val_accuracy: 0.9817
```

Comped to NN model, the CNN model appears to have similar performance (check the loss and accuracy metrics of both models)…

However, keep in mind that we deal with very small images (28 x 28) — things will be different if we consider e.g. Megapixel scale pictures…



Inspect our first CNN model


```
python -i model_cnn.py
>>> model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 7, 7, 32)	544
conv2d_1 (Conv2D)	(None, 1, 1, 64)	100,416
flatten (Flatten)	(None, 64)	0
dense (Dense)	(None, 128)	8,320
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

?

?

?

?

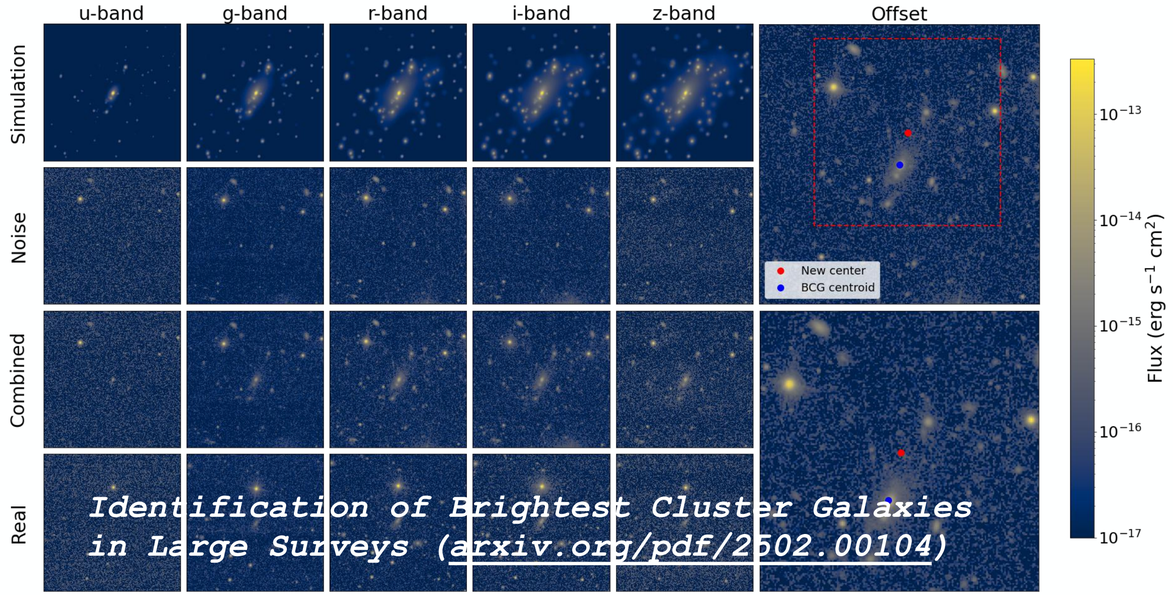
Total params: 110,570 (431.91 KB)
Trainable params: 110,570 (431.91 KB)
Non-trainable params: 0 (0.00 B)

Do the math yourself, do you arrive to the same numbers?

On the power of CNNs...

Consider 4K image classification(~10 megapixels):

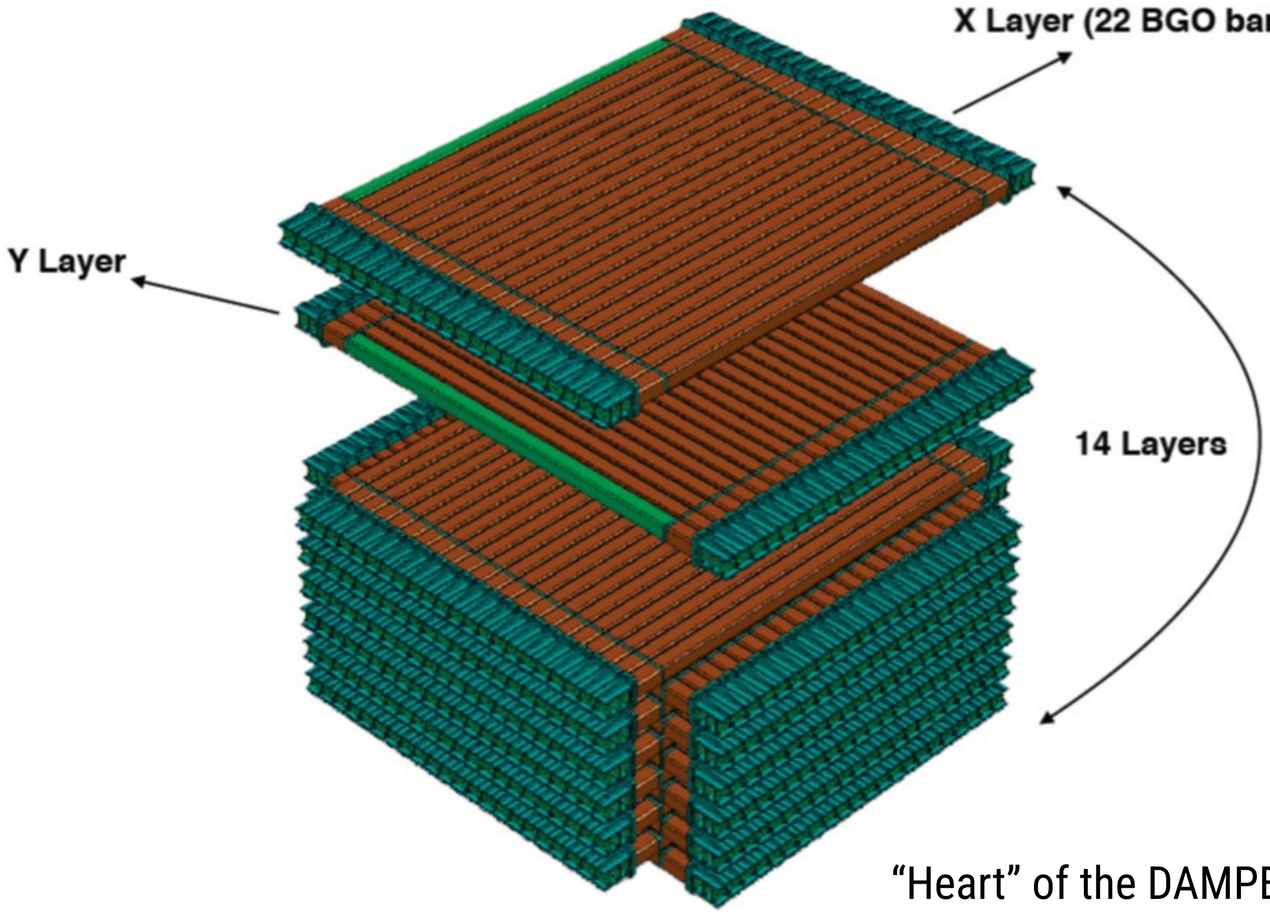
- Take a simple NN (not CNN) model architecture from the MNIST example (with ~100 neurons in the first layer) — **how many trainable parameters do you get?** Compare it with the number of trainable parameters we had in our NN and CNN models (~100’000). The number of parameters in NN model looks big, right? (**Comparable to GPT-2!**)
- CNNs, on the other hand, allow to extract features from the images of arbitrary size and resolution. The complexity of CNN model is defined only by the filter dimension, their number, and amount of layers — all these can be set to whatever number depending on how complex is the model that you want to build (how much trainable date you have etc.)

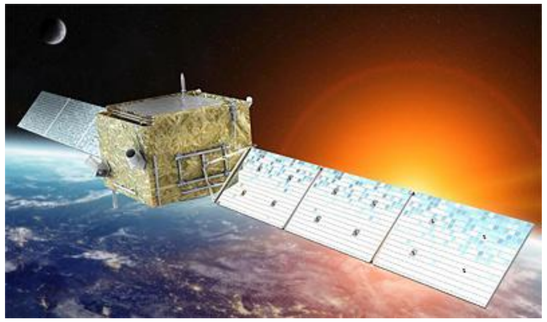


CNN example with the DAMPE cosmic ray detector

DAMPE example

Predicting cosmic ray (or gamma ray) particle direction from a signal (shower) in DAMPE calorimeter:



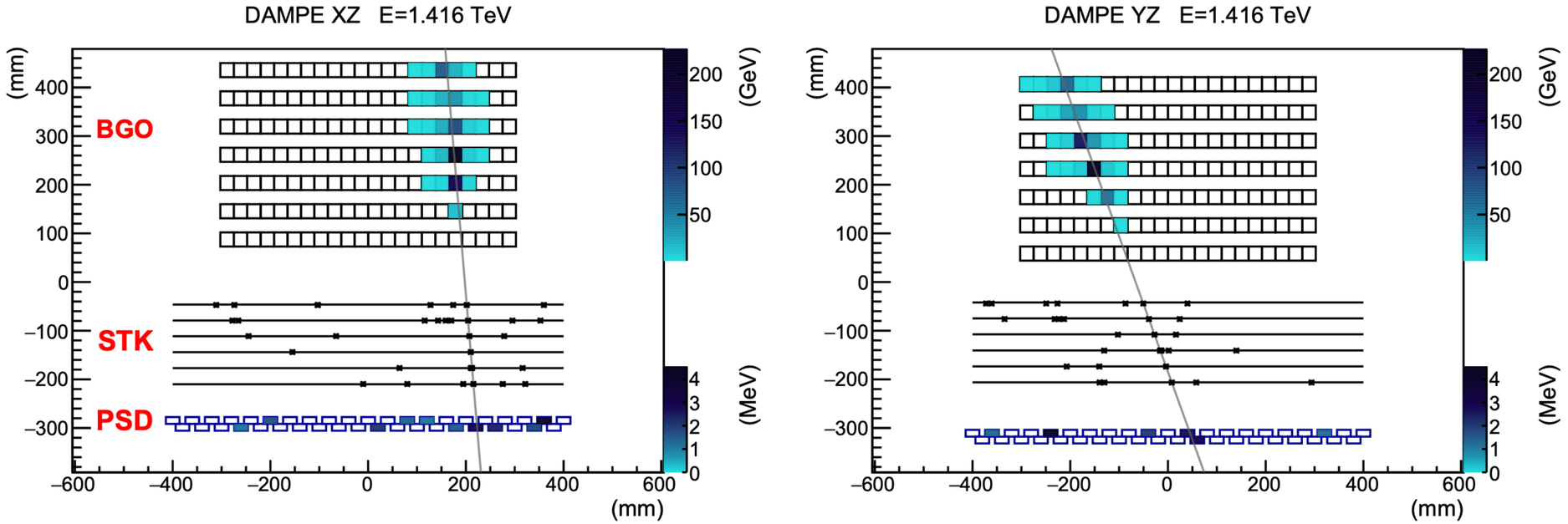


“Heart” of the DAMPE detector - the BGO imaging calorimeter

DAMPE example

Predicting cosmic ray (or gamma ray) particle direction from a signal (shower) in DAMPE calorimeter:

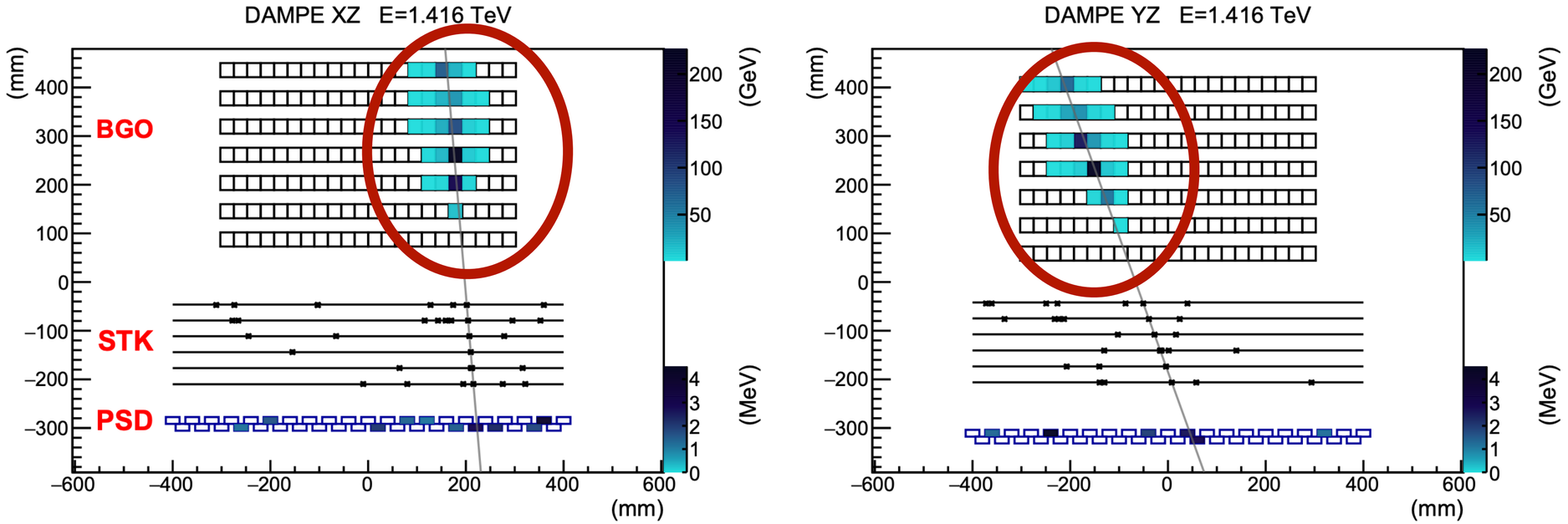
DAMPE detects particle spatial information and energy deposition in two orthogonal projections (XZ and YZ)



DAMPE example: classical approach (no AI)

Predicting cosmic ray (or gamma ray) particle direction from a signal (shower) in DAMPE calorimeter:

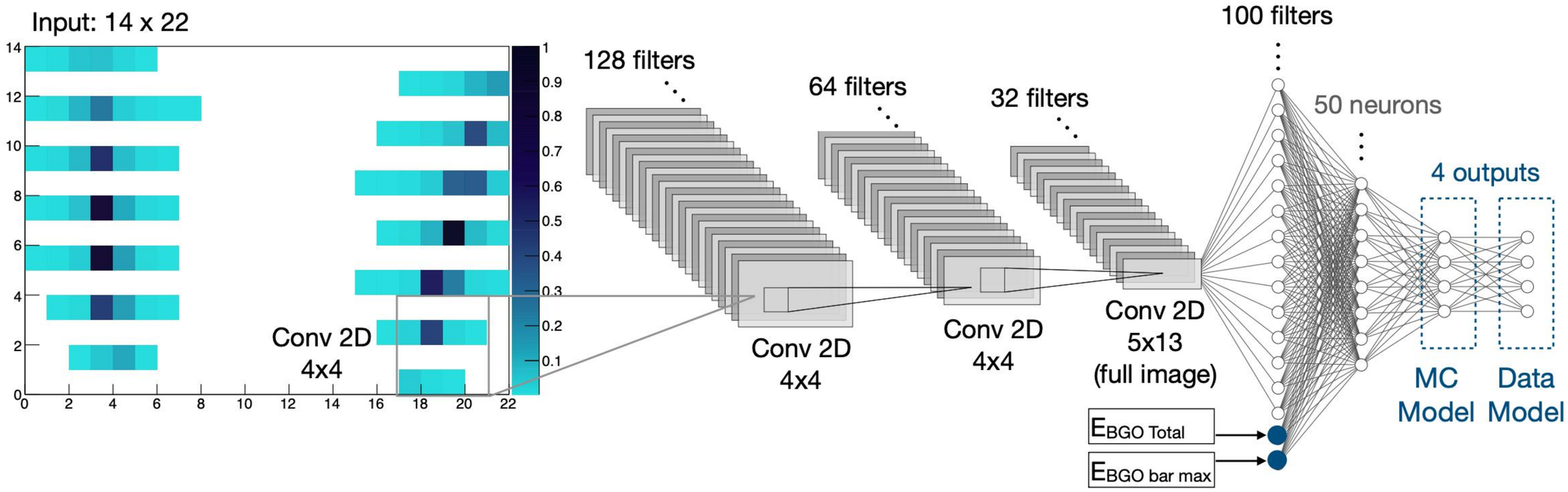
DAMPE detects particle spatial information and energy deposition in two orthogonal projections (XZ and YZ)



In a classical approach, the particle trajectory in the calorimeter is estimated by performing a linear regression fit of the line through the points in calorimeter, where the contribution of every point in the fit is weighted by the energy deposition in this point

DAMPE example: CNN

- We will use CNN instead of the classical linear regression - to predict particle direction
- We will train it on a sample of ~140’000 simulated particle showers in DAMPE



DAMPE example: getting the data

- Download and setup the training DAMPE data and the corresponding software:

```
git clone https://gitlab.cern.ch/andrii/mlregressioncalo.git
cd mlregressioncalo
git switch tutorial
source ./setup.sh      # do it in every new console (cd [path-/mlregressioncalo]; source ./setup.sh)
```

- Inspect the content of DAMPE package & data

```
python
>>> from dampe import get_dampe_data, plot_dampe_event
>>> data = get_dampe_data()
>>> data.keys()
dict_keys(['caloimages', 'truthdata', 'standardrecdata'])
```

DAMPE images on which we will train our network

Array of 4 numbers representing the truth particle direction (two in XZ projection, two in YZ)

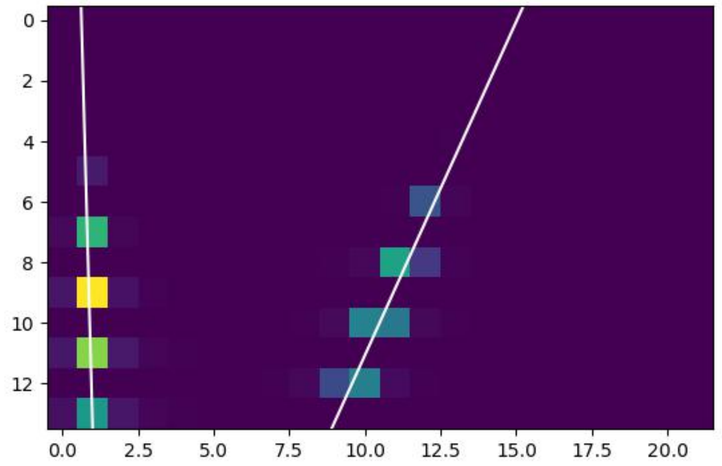
The same 4 numbers but obtained with the standard linear regression algorithm — we will not use it for training, but will keep it for reference when comparing with the CNN predictions

Do yourself — check the dimensions of the data arrays: `data['caloimages'].shape` *etc.*

DAMPE example: visually examining the data

- Let's inspect some of the data events:

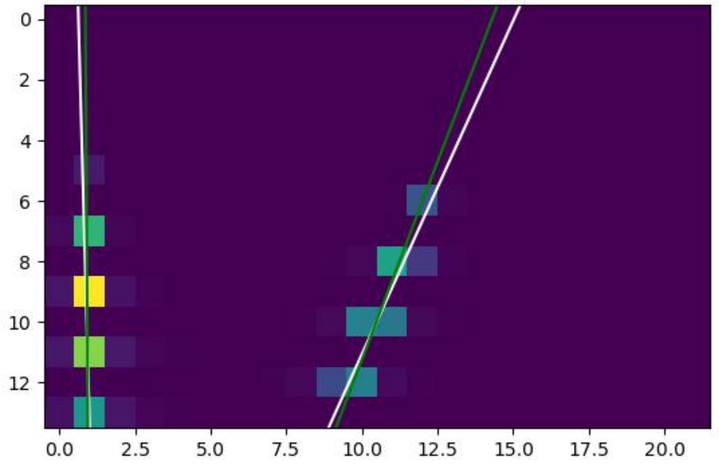
```
>>> plot_dampe_event(data['caloimages'][10],data['truthdata'][10])
```



DAMPE example: visually examining the data

- Let's inspect some of the data events:

```
>>> plot_dampe_event(data['caloimages'][10],data['truthdata'][10],data['standardrecdata'][10])
```



Let's add a prediction with the standard (linear regression) algorithm

You can experiment with the different events in the dataset, for example in the event 19 one can see a very distinct difference between the truth and reconstructed particle direction.

DAMPE CNN training ...

- Let's get back to our CNN training code from the MNIST example, we will use it as a base:

```
cp run_fit.py run_fit_dampe.py
cp model_cnn.py  model_cnn_dampe.py
```

- In `run_fit_dampe.py` change:

```
#from model_cnn import model          # before
from model_cnn_dampe import model     # now
from dampe import get_dampe_data      #
from pickle import dump               #
...

# get the dataset                                          # before
#mnist = tf.keras.datasets.mnist                           #
#(x_train, y_train), (x_test, y_test) = mnist.load_data()  #
#x_train, x_test = x_train / 255.0, x_test / 255.0         #
data = get_dampe_data()                                    # now
x, y = data['caloimages'], data['truthdata']    #
...

#loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)                    # before
loss_fn = tf.keras.losses.MeanAbsoluteError()                                                 # now
model.compile(optimizer='adam',loss=loss_fn, metrics=['mean_squared_error']) #metrics=['accuracy'])     #

history = model.fit(x, y, epochs=5, validation_split=0.1) # training part remains the same
# After the fitting part let's save the model and history for further analysis
model.save_weights('model_cnn_dampe.weights.h5')
dump(history.history, open('history_cnn_dampe.p','wb'))
# comment out the rest of the code afterwards (plt.plot(history.history ... etc.)
```

Note that we reduce number of epochs since training DAMPE models is more time consuming ...

DAMPE CNN model ...

- Now we need to modify the model itself, keeping the changes minimal. In the `model_cnn_dampe.py` do the following:

```
#input_ = tf.keras.Input(shape=(28,28,1)) # before
input_ = tf.keras.Input(shape=(14,22,1))  # now
layer  = tf.keras.layers.ZeroPadding2D(((1,1),(1,1)))(input_)

# convolutional part
#layer = tf.keras.layers.Conv2D(32,(4,4),strides=(4,4), activation="relu")(input_)# before
layer = tf.keras.layers.Conv2D(32,(4,4),strides=(4,4), activation="relu")(layer)  # now
#layer = tf.keras.layers.Conv2D(64,(7,7),strides=(1,1), activation="relu")(layer) # before
layer = tf.keras.layers.Conv2D(64,(6,4),strides=(1,1), activation="relu")(layer)  # now
layer = tf.keras.layers.Flatten()(layer)                                          # same as before (not changed)

# NN part
layer = tf.keras.layers.Dense(128, activation="relu")(layer)     # same as before (not changed)
#layer = tf.keras.layers.Dropout(0.2)(layer)                     # comment out the dropout part
#layer = tf.keras.layers.Dense(10)(layer)                        # before
layer = tf.keras.layers.Dense(4)(layer)                          # now (instead of 10 neurons, now we have 4)

# this remains the same as before ...
model = tf.keras.Model(inputs=input_,outputs=layer)
```

Test/print the layer shapes/dimensions at different steps, let's see if we understand it ...

- We are ready to run the DAMPE model training:

```
python run_fit_dampe.py

...
Epoch 1/50
3993/3993 [==============================] - 3s 684us/step - loss: 76.2474 - mean_squared_error: 11470.5166 - val_loss:
34.6831 - val_mean_squared_error: 2725.1121
...
```

Note the loss value at the first iteration (we will compare it later with the loss of the other model)

DAMPE CNN model from the paper

- Let's try a **deeper** model (from the paper), `cp model_cnn.py model_cnn_dampe_paper.py` and modify it:

```
#input_ = tf.keras.Input(shape=(28,28,1)) # before
input_ = tf.keras.Input(shape=(14,22,1))  # now

# convolutional part
#layer = tf.keras.layers.Conv2D(32,(4,4),strides=(4,4), activation='relu')(input_) # before
#layer = tf.keras.layers.Conv2D(64,(7,7),strides=(1,1), activation='relu')(layer)  #
layer = tf.keras.layers.Conv2D(128,(4,4), activation="relu")(input_)               # now
layer = tf.keras.layers.Conv2D(64,(4,4), activation="relu")(layer)                 #
layer = tf.keras.layers.Conv2D(32,(4,4), activation="relu")(layer)                 #
# ... what is the ouput shape after this layer? (see for yourself with print…)    #
layer = tf.keras.layers.Conv2D(100,(5,13), activation="relu")(layer)               #
# ... why is there (5,13) filter size used at this point? What is the output shape?
layer = tf.keras.layers.Flatten()(layer)

# NN part
#layer = tf.keras.layers.Dense(128, activation='relu')(layer)   # before
#layer = tf.keras.layers.Dropout(0.2)(layer)                    #
#layer = tf.keras.layers.Dense(10)(layer)                       #
layer = tf.keras.layers.Dense(50, activation="relu")(layer)     # now
layer = tf.keras.layers.Dense(4, activation="linear")(layer)    #

# this remains the same as before ...
model = tf.keras.Model(inputs=input_,outputs=layer)
```

Model from:

Astroparticle Physics 146 (2023) 102795

Contents lists available at ScienceDirect

Astroparticle Physics

journal homepage: www.elsevier.com/locate/astropartphys

A deep learning method for the trajectory reconstruction of cosmic rays with the DAMPE mission

Andrii Tykhonov [a,*], Andrii Kotenko [a], Paul Coppin [a], Maksym Deliyergiyev [a], David Droz [a], Jennifer Maria Frieden [b], Chiara Perrina [b], Enzo Putti-Garcia [a], Arshia Ruina [a], Mikhail Stolpovskiy [a], Xin Wu [a]

[a] Department of Nuclear and Particle Physics, University of Geneva, CH-1211, Switzerland
[b] Institute of Physics, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015, Lausanne, Switzerland

DAMPE CNN model from the paper

- In `run_fit_dampe.py` modify the CNN model import and run the script again:

```
#from model_cnn_dampe import model          # before
from model_cnn_dampe_paper import model     # now


# save model under different name
#model.save_weights('model_cnn_dampe.weights.h5')
#dump(history.history, open('history_cnn_dampe.p','wb'))
model.save_weights('model_cnn_dampe_paper.weights.h5')
dump(history.history, open('history_cnn_dampe_paper.p','wb'))
```

```
python run_fit_dampe.py
```

```
...
Epoch 1/50
3993/3993 [==============================] - 44s 11ms/step - loss: 27.7825 - mean_squared_error: 2802.2505 - val_loss: 10.1922 -
val_mean_squared_error: 229.4982
...
```

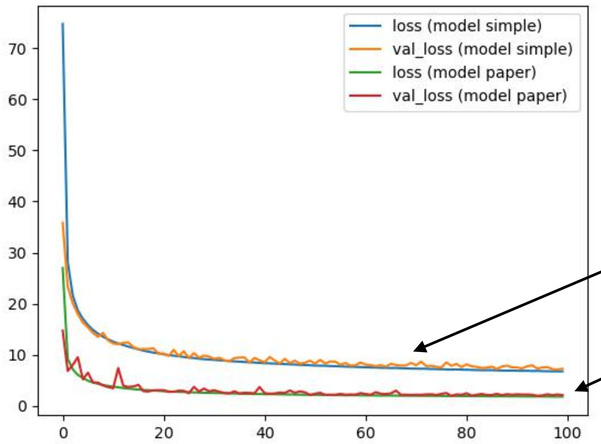
Note that the loss value after the first iteration is considerably lower that with the first (simple) CNN model that we tried... Also, training takes considerably longer due to a more complex (dense) CNN model!

It will take 1-2 hours on conventional hardware to train this model... We can't wait that long, hence you can find this saved model and training history inside the dampe package downloaded earlier, see the `saved_models` *folder*

Comparing the two CNN models


```
python

from pickle import load
import matplotlib.pyplot as plt

history1 = load(open('history_cnn_dampe.p','rb'))
history2 = load(open('history_cnn_dampe_paper.p','rb'))

plt.plot(history1['loss'])
plt.plot(history1['val_loss'])
plt.plot(history2['loss'])
plt.plot(history2['val_loss'])
plt.legend(['loss (model simple)', 'val_loss (model simple)', 'loss (model paper)', 'val_loss (model paper)'])
plt.show()
```

You can find both models and their training history inside `saved_model` folder of the DAMPE package. **Copy the content of `saved_model` folder to the folder where you run your examples.**



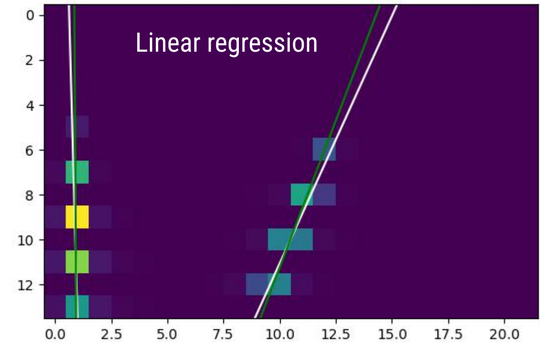
Simple CNN model
(same as for MNIST example)

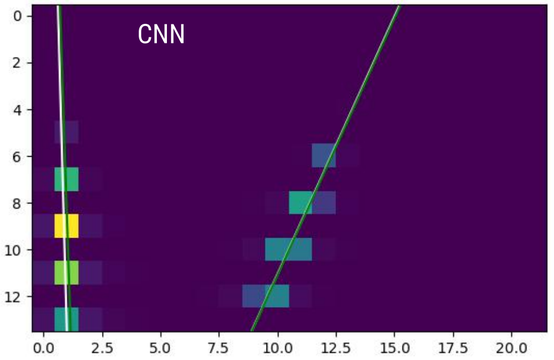
Model from the DAMPE paper
(deeper, more neurons per layer)

Fun part: inference/prediction with the DAMPE CNN

- Let's compare performance of classical DAMPE algorithm and the CNN one. Create `test_dampe_images.py` and run it:

```
# This is test_dampe_images.py file for DAMPE CNN testing

from dampe import get_dampe_data, plot_dampe_event
import tensorflow as tf
from model_cnn_dampe import model

data = get_dampe_data()

# plot the results with the standard algorithm (linear regression)
standardprediction = data['standardrecdata'][10]
plot_dampe_event(data['caloimages'][10],data['truthdata'][10], standardprediction)

# plot the results with CNN
model.compile()
model.load_weights('model_cnn_dampe_paper.weights.h5')
prediction = model(data['caloimages'][10:11])
prediction = prediction[0] # prediction is done in batches. We have a "batch" of 1 event
plot_dampe_event(data['caloimages'][10],data['truthdata'][10], prediction)

# caclcualte yourself accucay of standard approach VS CNN
import numpy as np
n = 1000
print("Standard algorithm mean absolute error:",
   np.sum(np.abs(data['truthdata'][:n]-data['standardrecdata'][:n])) / (4*n))
prediction = model(data['caloimages'][:n])
print("CNN mean absolute error:",
   np.sum(np.abs(data['truthdata'][:n]-prediction)) / (4*n))
```





Recap of the DAMPE CNN model (training script)

run_fit_dampe.py:

```
import tensorflow as tf
import matplotlib.pyplot as plt
from model_cnn_dampe_paper import model
from dampe import get_dampe_data
from pickle import dump

data = get_dampe_data()
x, y = data['caloimages'], data['truthdata']

loss_fn = tf.keras.losses.MeanAbsoluteError()
model.compile(optimizer='adam', loss=loss_fn,
metrics=['mean_squared_error'])
history = model.fit(x, y, epochs=5, validation_split=0.1)

model.save_weights('./model_cnn_dampe_paper.weights.h5')
dump(history.history, open('history_cnn_dampe_paper.p','wb'))
```

Recap of the DAMPE CNN model (model definition)

model_cnn_dampe_paper.py:

```
import tensorflow as tf

input_ = tf.keras.Input(shape=(14,22,1))

# convolutional part
layer = tf.keras.layers.Conv2D(128,(4,4), activation="relu")(input_)
layer = tf.keras.layers.Conv2D(64,(4,4), activation="relu")(layer)
layer = tf.keras.layers.Conv2D(32,(4,4), activation="relu")(layer)
layer = tf.keras.layers.Conv2D(100,(5,13), activation="relu")(layer)
layer = tf.keras.layers.Flatten()(layer)

# fully-connected part
layer = tf.keras.layers.Dense(50, activation="relu")(layer)
layer = tf.keras.layers.Dense(4, activation="linear")(layer)

model = tf.keras.Model(inputs=input_,outputs=layer)
```

Exercise (20 mins)

- **Develop your own NN (not CNN) model for DAMPE** (similar to MNIST example discussed earlier)

  ➡ Experiment with architecture (number of layers, neurons, etc...)

  ➡ The goal is to try to obtain a simple NN model comparable in performance with the CNN

  For simplicity, as a metrics of "performance" just use the value of loss (the lower the better)

  No need to run the entire model training and visualization, just run a training with 2-3 iterations at most and see if your NN model converges as fast (with the similar or lower loss values) as the CNN model (usually it is already seen from the first few iterations how good the model is...)

- **Modify the original CNN according to your intuition and try to beat the original DAMPE CNN!**

Before we move on: let's repeat the DAMPE exercise while learning a bit of **PyTorch**

(We will need PyTorch for the following part of the course)

Install PyTorch

To avoid possible conflicts between the two frameworks, we will install PyTorch in a separate conda environment

- Install **PyTorch**
  1. Create new environment that we will call “tr”:
  2. Activate the “tf” environment:

```
conda create -n tr pytorch matplotlib
conda activate tr
```

- Test **PyTorch**

```
python
>>> import torch as tr
>>> print (tr.__version__)
```

Test PyTorch NN training

- Run the following code snippet in python (let's call it `test_pytorch_minimal.py`):

```
import torch as tr
import numpy as np

x = tr.tensor(np.random.rand(1000,10),dtype=tr.float64) # random sample of 1000 sets of numbers
y = tr.tensor(np.random.rand(1000,),dtype=tr.int64)     # random sample of 1000 set 'labels'
model = tr.nn.Sequential( tr.nn.Linear(10,10,dtype=tr.float64))
criterion = tr.nn.CrossEntropyLoss()
optimizer = tr.optim.Adam(model.parameters())

# one training step
y_pred = model(x)
loss = criterion(y_pred,y)
loss.backward()
optimizer.step()
print ("Loss:",loss.item())
```

- If everything is installed correctly, you should see the log of the training:

```
Loss: 1.930114470175824
```

PyTorch DAMPE example - training script

Let's rewrite our DAMPE CNN code in PyTorch:

`cp run_fit_dampe.py run_fit_dampe_torch.py`,

edit `run_fit_dampe_torch.py`:

```
#import tensorflow as tf
import torch as tr
#from model_cnn_dampe_paper import model
from model_cnn_dampe_paper_torch import model


#... after you obtained the dampe data in the usual way, convert it to torch tensors
x, y = tr.tensor(x,dtype=tr.float32), tr.tensor(y,dtype=tr.float32)

#... we need to convert data format from [N,H,W,C] (Tensorflow) to [N,W,H,C] (Pytorch)
x = x.squeeze(3).unsqueeze(1)
#... you can check later by inserting print(x.shape) statement before and after...

#loss_fn = tf.keras.losses.MeanAbsoluteError()
#model.compile(optimizer='adam', loss=loss_fn, metrics=['mean_squared_error'])
criterion = tr.nn.L1Loss()                          # mean absolute error loss
optimizer = tr.optim.Adam(model.parameters())


# continued on the next page ...
```

PyTorch DAMPE example - training script

... edit `run_fit_dampe_torch.py` continued:

instead of this

```
...
#history = model.fit(x, y, epochs=5, validation_split=0.1)
model.train() # set model in the training mode
history = {'loss':[],'loss_val':[]}
n_total, n_split, n_batch = x.shape[0], int(x.shape[0] * 0.9), 32
for i in range(5): # loop over training epochs
    loss_train, loss_val, n_batches_train, n_batches_val = 0, 0, 0, 0
    for j in range(0, n_split, n_batch): # loop over batches in training sub-sample
        print (f"Processed: {j*100./n_split:3.1f}%",end="\r") # status bar
        optimizer.zero_grad()
        y_pred = model(x[j:j+n_batch])
        loss = criterion(y_pred,y[j:j+n_batch])
        loss.backward()
        optimizer.step()
        loss_train+=loss.item()
        n_batches_train+=1
    model.valid() # set model in validation mode
    for j in range(n_split, n_total, n_batch): # loop over batches in validation sub-sample
        print (f"Processed valid: {(j-n_split)*100./(n_total-n_split):3.1f}",end="\r") # status bar
        y_pred = model(x[j:j+n_batch])
        loss = criterion(y_pred,y[j:j+n_batch])
        loss_val+=loss.item()
        n_batches_val+=1
    loss_train, loss_val = loss_train / n_batches_train, loss_val / n_batches_val
    print ("loss:",loss_train, "loss_val:",loss_val)
    history['loss'], history['loss_val'] = history['loss']+[loss_train], history['loss_val']+[loss_val]
#model.save_weights('./model_cnn_dampe_paper.weights.h5')
#dump(history.history, open('history_cnn_dampe_smallmodel.p','wb'))
tr.save(model.state_dict(), './model_cnn_dampe_paper_torch.pth') # just to show how saving model in torch works
dump(history.history, open('history_cnn_dampe_paper_torch.p','wb')) # the same, just changed the file name
```

we have that...

PyTorch DAMPE example - model definition

Let's rewrite the **model** in PyTorch,

`cp model_cnn_dampe_paper.py model_cnn_dampe_paper_torch.py`,
edit `model_cnn_dampe_paper_torch.py`:

```
#import tensorflow as tf                                    # before
import torch as tr                                          # now
import numpy as np

#input_ = tf.keras.Input(shape=(14,22,1))

# convolutional part
#layer = tf.keras.layers.Conv2D(128,(4,4), activation="relu")(input_)
#layer = tf.keras.layers.Conv2D(64,(4,4), activation="relu")(layer)
#layer = tf.keras.layers.Conv2D(32,(4,4), activation="relu")(layer)
#layer = tf.keras.layers.Conv2D(100,(5,13), activation="relu")(layer)
#layer = tf.keras.layers.Flatten()(layer)

# usual NN part remains the same
#layer = tf.keras.layers.Dense(50, activation="relu")(layer)
#layer = tf.keras.layers.Dense(4, activation="linear")(layer)
#model = tf.keras.Model(inputs=input_,outputs=layer)

# continued on the next page...
```

Import torch instead of tensorflow

Just comment it out but do not delete yet (convenient to keep for a reference while we code the torch model ...)

PyTorch DAMPE example - model definition

... edit `model_cnn_dampe_paper_torch.py` continued:

```
class DampeCNN(tr.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_layer1   = tr.nn.Conv2d(1,   128, (4,4 ))
        self.conv_layer2   = tr.nn.Conv2d(128, 64,  (4,4 ))
        self.conv_layer3   = tr.nn.Conv2d(64,  32,  (4,4 ))
        self.conv_layer4   = tr.nn.Conv2d(32,  100, (5,13))
        self.flatten_layer = tr.nn.Flatten()
        self.dense_layer1  = tr.nn.Linear(100,50)
        self.dense_layer2  = tr.nn.Linear(50, 4)
        self.relu          = tr.nn.ReLU()
    def forward(self, x):
        x = self.conv_layer1(x)
        x = self.relu(x)
        x = self.conv_layer2(x)
        x = self.relu(x)
        x = self.conv_layer3(x)
        x = self.relu(x)
        x = self.conv_layer4(x)
        x = self.relu(x)
        #print (x.shape)
        #raise SystemExit
        x = self.flatten_layer(x)
        x = self.dense_layer1(x)
        x = self.relu(x)
        x = self.dense_layer2(x)
        return x

model = DampeCNN()
```

For simple models (including this one) we could in principle use a simplified syntax with `tr.nn.Sequential` (see our 2nd PyTorch installation test), but the low-level definition shown here offers more flexibility (for example in Transformer models that we will consider later)

PyTorch DAMPE example - test

Let's test a bit our model. Similar to PyTorch case, we can feed it some random data and print the layer shape on the way. First, let's add a print statements to the model `forward` method in `model_cnn_dampe_paper_torch.py`:

```
class DampeCNN(tr.nn.Module):
    ...
    ...
    def forward(self, x):
        x = self.conv_layer1(x)
        print ("Dimensions after layer 1:", x.shape)
        # etc
        ...
```

Enter a print statement

Now let's run it the model a random data:

```
python -i model_cnn_dampe_paper_torch.py
>>> data=tr.tensor(np.zeros((100,1,14,22)),dtype=tr.float32)
>>> tmp = model(data)

Dimensions after layer 1: torch.Size([100, 128, 11, 19])
```

Result of a print statement

PyTorch DAMPE example - test

Now finally as we have all pieces in place, we can run the training of the DAMPE CNN model in PyTorch:

```
python  run_fit_dampe_torch.py

  Processed valid: 99.9 elapsed:164.9s
  loss: 78.60382124119583 loss_val: 21.724795972978747
  Processed valid: 99.9 elapsed:165.9s
  loss: 16.86408121580472 loss_val: 12.455400391741916
  ...
```

We will not run this training, just make sure that it works in principle ... Later you can run that CNN model in PyTorch and compare the performances with Tensorflow: execution time, convergence, final accuracy (loss) of the model ...

Now let's briefly re-cap our PyTroch CNN example ...

Recap (PyTorch DAMPE example)

run_fit_dampe_torch.py

```
import torch as tr
import matplotlib.pyplot as plt
from model_cnn_dampe_torch import model
from dampe import get_dampe_data
from pickle import dump
import time

data = get_dampe_data()
x, y = data['caloimages'], data['truthdata']
x, y = tr.tensor(x,dtype=tr.float32), tr.tensor(y,dtype=tr.float32)
x = x.squeeze(dim=3).unsqueeze(dim=1)

criterion = tr.nn.L1Loss()
optimizer = tr.optim.Adam(model.parameters())

# training loop
history = {'loss':[],'loss_val':[]}
n_total, n_split, n_batch = x.shape[0], int(x.shape[0] * 0.9), 32
for i in range(5): # loop over training epochs
    loss_train, loss_val, n_batches_train, n_batches_val, timestamp = 0, 0, 0, 0, time.time()
    model.train() # set model in training mode
    for j in range(0, n_split, n_batch): # loop over batches in training sub-sample
        print (f"Processed: {j*100./n_split:3.1f}% elapsed:{time.time()-timestamp:4.1f}s",end="\r")
        optimizer.zero_grad()
        y_pred = model(x[j:j+n_batch])
        loss = criterion(y_pred,y[j:j+n_batch])
        loss.backward()
        optimizer.step()
        loss_train+=loss.item()
        n_batches_train+=1
    model.eval() # set model in validation mode
    for j in range(n_split, n_total, n_batch): # loop over batches in validation sub-sample
        print (f"Processed valid: {(j-n_split)*100./(n_total-n_split):3.1f} elapsed:{time.time()-timestamp:4.1f}s",end="\r")
        y_pred = model(x[j:j+n_batch])
        loss = criterion(y_pred,y[j:j+n_batch])
        loss_val+=loss.item()
        n_batches_val+=1
    loss_train, loss_val = loss_train / n_batches_train, loss_val / n_batches_val
    print ("\nloss:",loss_train, "loss_val:",loss_val)
    history['loss'], history['loss_val'] = history['loss'] + [loss_train], history['loss_val'] + [loss_val]

# save model and training history
tr.save(model.state_dict(), './model_cnn_dampe_paper_torch.pth')
dump(history, open('history_cnn_dampe_paper_torch.p','wb')) # remains the same - we just changed the name of the file
```

Recap (PyTorch DAMPE example)

model_cnn_dampe_paper_torch.py

```
import torch as tr
class DampeCNN(tr.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_layer1   = tr.nn.Conv2d(1,   128, (4,4 ))
        self.conv_layer2   = tr.nn.Conv2d(128, 64,  (4,4 ))
        self.conv_layer3   = tr.nn.Conv2d(64,  32,  (4,4 ))
        self.conv_layer4   = tr.nn.Conv2d(32,  100, (5,13))
        self.flatten_layer = tr.nn.Flatten()
        self.dense_layer1  = tr.nn.Linear(100,50)
        self.dense_layer2  = tr.nn.Linear(50, 4)
        self.relu          = tr.nn.ReLU()
   def forward(self, x):
        x = self.conv_layer1(x)
        x = self.relu(x)
        x = self.conv_layer2(x)
        x = self.relu(x)
        x = self.conv_layer3(x)
        x = self.relu(x)
        x = self.conv_layer4(x)
        x = self.relu(x)
        x = self.flatten_layer(x)
        x = self.dense_layer1(x)
        x = self.relu(x)
        x = self.dense_layer2(x)
        return x
model = DampeCNN()
```

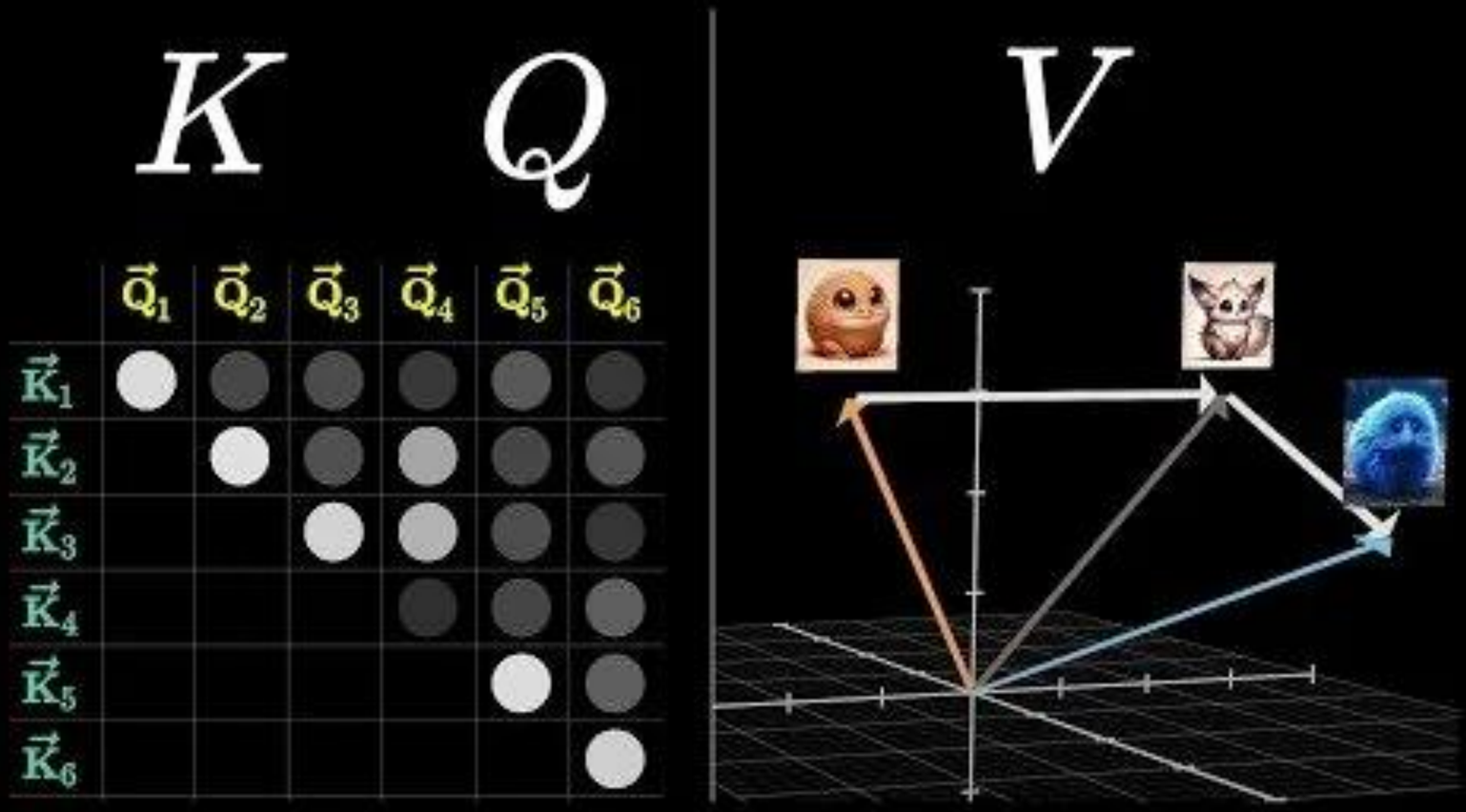
Exercise (10 mins)

- **Implement your previously developed DAMPE NN (not CNN) in PyTorch**
  - ➡ Goal: make sure you understand the layer implementation in PyTorch (where and how to look for documentation)
  - ➡ You will create your own PyTorch model class similar to the CNN example
  - ➡ No need to run the full training, just make sure that the training works and you are able to run ~ 1 training epoch

From the previous example, you should already know the name of layers in PyTorch. For more information/references, see PyTorch documentation

- For example, the Flatten layer in PyTorch: https://docs.pytorch.org/docs/stable/generated/torch.nn.Flatten.html

Part III: Transformers



Transformers

Before starting preparing this tutorial, I promised myself not to make too obvious flat jokes connected to the comic books characters ...

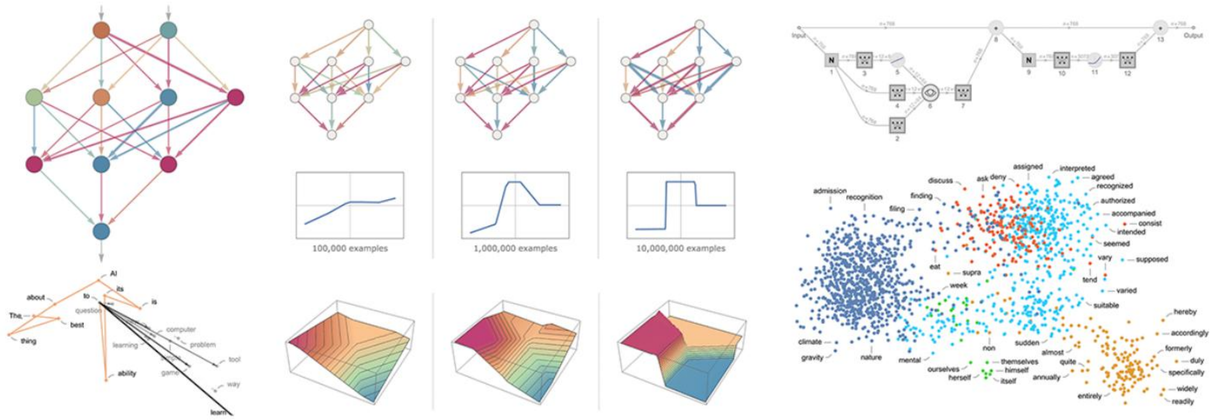
Transformers

Before starting preparing this tutorial, I promised myself not to make too obvious flat jokes connected to the comic books characters ...

Now seriously... According to Stephen Wolfram:

What Is ChatGPT Doing ... and Why Does It Work?

February 14, 2023



It's Just Adding One Word at a Time

https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/

On Large Language Models (LLMs)

From Wolfram's article:

The best thing about AI is its ability to

learn	4.5%
predict	3.5%
make	3.2%
understand	3.1%
do	2.9%

And the remarkable thing is that when ChatGPT does something like write an essay what it's essentially doing is just asking over and over again "given the text so far, what should the next word be?"—and each time adding a word. (More precisely, as I'll explain, it's adding a "token", which could be just a part of a word, which is why it can sometimes "make up new words".)

Adding from myself: in some sense, LLMs can be considered ~ autocompletion on steroids

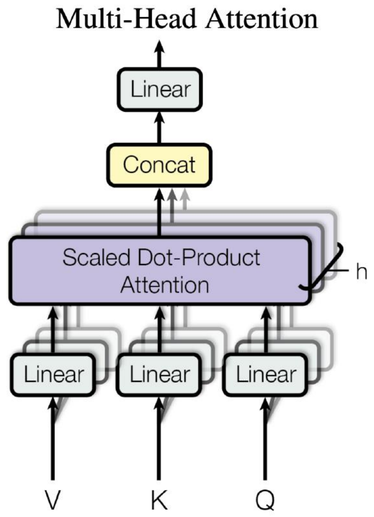
On Large Language Models (LLMs) & Transformers

Computer Science > Computation and Language https://arxiv.org/abs/1706.03762

[Submitted on 12 Jun 2017 (v1), last revised 2 Aug 2023 (this version, v7)]

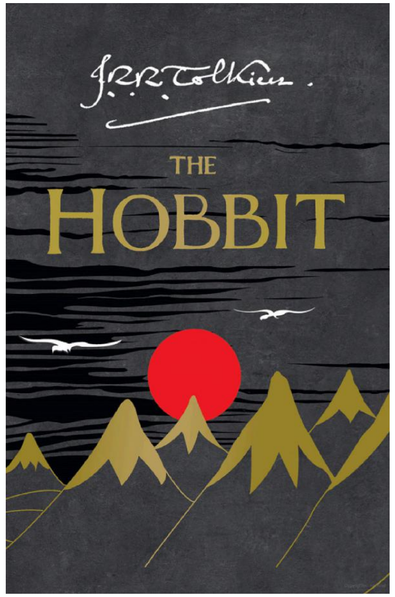
Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin



It turned to be that the so-called "Transformer"-like architectures, originally suggested in the "Attention is All You Need" paper, are currently the best ones for LLM implementation (including GPT, Gemini, DeepSeek)

While the first GPT models were predicting tokens based on the preceding sequence of text of ~1-2 pages at most, GPT-4 turbo has a maximum sequence length (based on which it does "autocompletion") of the size of ~ Orwell's "1984" (or "The Hobbit", if you like)



What is a **token**?

LLMs are prediction tokens one-by-one

- **A word is token ?** - in most cases **YES**
- **Is token a word?** - **NOT** necessarily
  - ➡A token can be a part of a word
  - ➡It can be a single digit, i.e. "1", "2", etc, combination of digits "12", "567", etc.
  - ➡One single letter is a token, frequent combination of letters/syllables are tokens (that's how LLMs can invent new words)
  - ➡Event frequent combination of words, or entire sentences can be tokens, depending on the implementation
  - ➡End of sequence (e.g. what tells GPT to stop further generation of reply in the chat) is a token (in fact an important one!)

What is a **token**?

Neural Networks deal with numbers. How do we represent tokens as numbers? Say, GPT-4 uses ~100'000 token dictionary

- Obviously, we assign every token a number from 0 to 100'000
- We can also represent those in vectors of 100'000 digits, all of which are 0 except for one:

100'000 vectors

```
[0,0,0, ..., 1, ..., 0, 0,0]   <-- Token 1 ("Hello")
[0,1,0, ..., 0, ..., 0, 0,0]   <-- Token 2 ("world")
...                                etc.
[0,0,0, ..., 0, ..., 0, 0,1]   <-- Token 100'000 ("bye")
```

100'000 numbers

Now what LLM does when prediction a next token, it essentially yields a so-called **logit** vector of 100'000 numbers each representing a probability for a certain token to be the next one in a sequence:

```
[0.01, 0., 0.8, ..., 0.02, ..., 0, 0,1]
```

 # Numbers sum up to 1

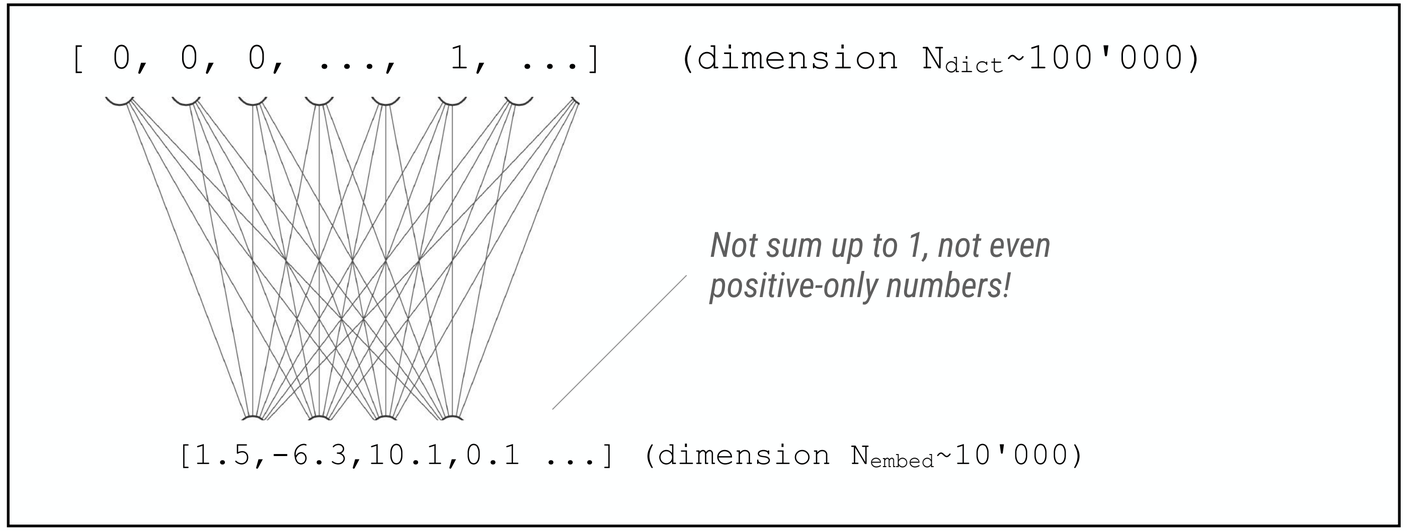
100'000 numbers

Tokens & **Embedding** vectors

Embedding vectors (or Embeddings) - key concept in LLMs!

- In previous illustration of token vectors - how do we encode similarity of tokens in that kind of vectors in that n-dimensional space (n is the dictionary size - 100'000 in previous example) - The answer is - **we don't do that!**
- Instead, we **convert token into embedding vector** in lower dimension space through linear transformation:

Embedding transformation is a look-up table of $N_{dict} \cdot N_{embed}$ *parameters (**trainable**):*



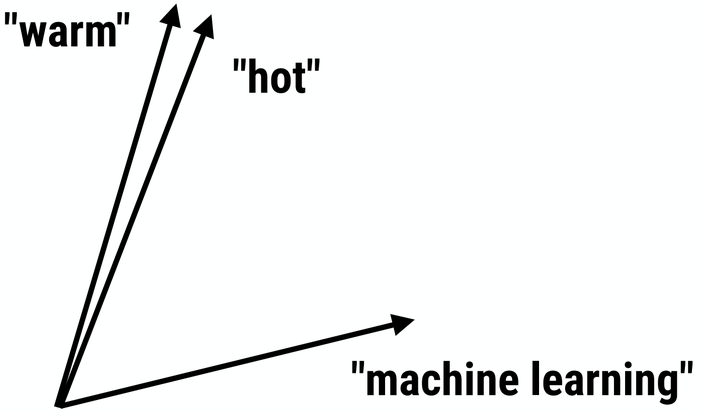
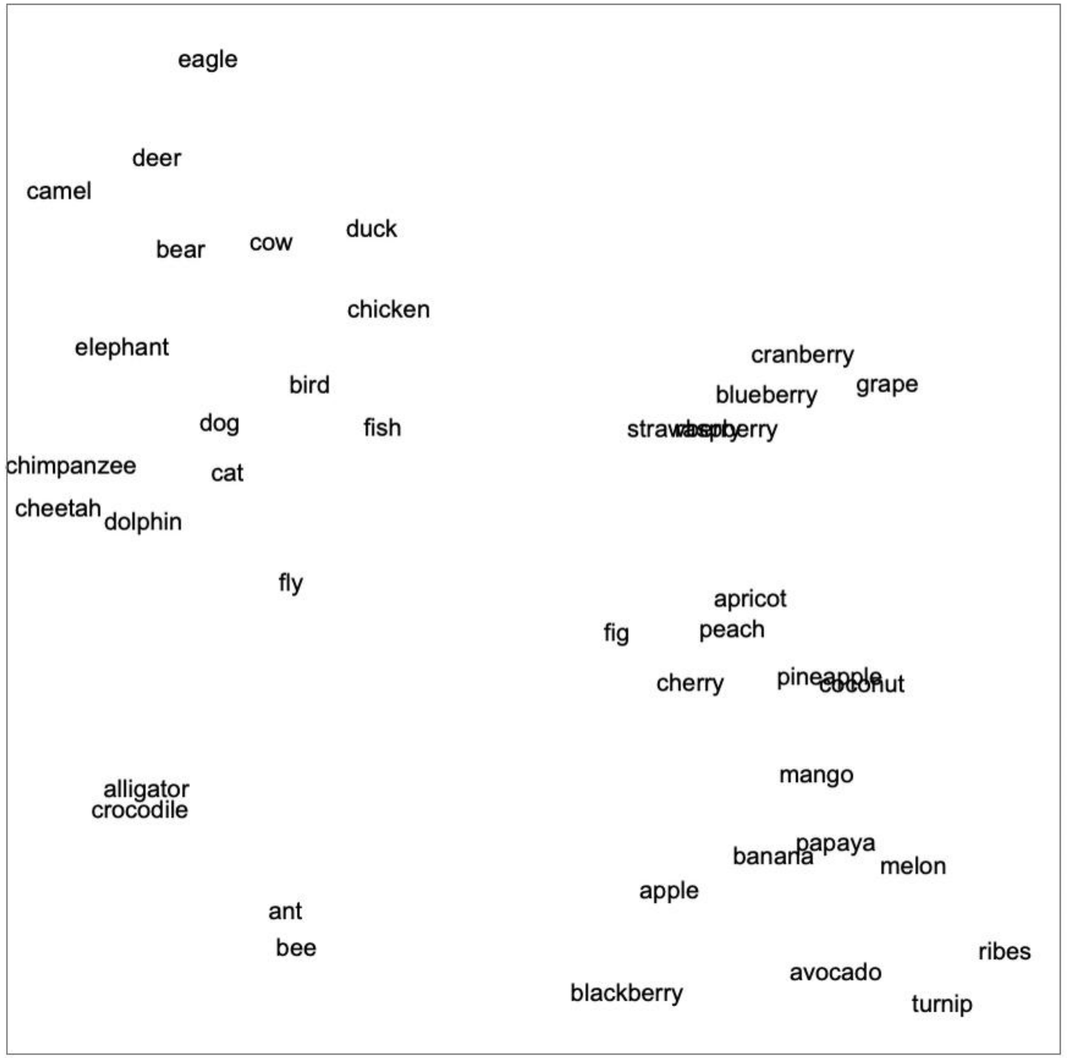


Illustration of embedding vectors corresponding to the tokens "warm", "hot", and "machine learning". The "warm" and "hot" embeddings will likely occur very aligned in the embedding space

Tokens & **Embedding** vectors



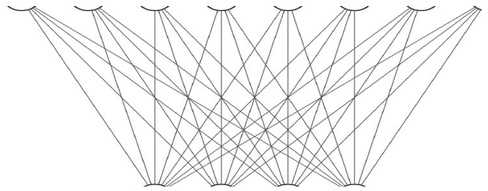
Projection of typical Embedding space in 2D (credit: Stephen Wolfram)

*You can think of embedding space as a **way of mapping language into a multi-dimensional cartesian space**, where close-by-semantical-meaning words/phrases/ syllables/etc. will normally appear as nearby vectors*

Data flow in Langauge Models

Tokens-to-embeddings

`[ 0, 0, 0, ..., 1, ...]` (input tokens)

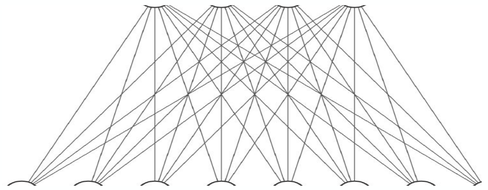


`[1.5,-6.3,10.1,0.1 ...]` (input embedding vectors)

Neural Network (Transformer)

`[1.5,-6.3,10.1,0.1 ...]` (output/predicted embedding vector)

Embeddings-to-token probabilities



Sum up to 1

`[ 0.1, 0.01, 0.8, ...]` (output token probabilities logit vector N_{dict}~100'000)

Tokens & Embedding vectors: try yourself

Before we go to Transformer implementation, let's first experiment with tokens & embeddings

Why this is important: if we want to *transform* the use "Transformers" into physics applications we shall first get a clear idea of the "Transformers" original intended use (which is in fact language processing)

```
conda activate tr
conda install transformers
python

>>>
>>> from transformers import  AutoTokenizer
>>> tokenizer = AutoTokenizer.from_pretrained("gpt2")
>>> input_tokens = tokenizer("I am GPT", return_tensors="pt").input_ids
>>> output_tokens = tokenizer("Sono GPT", return_tensors="pt").input_ids
>>> print(input_tokens)
  tensor([[   40,   716,   402, 11571]])
>>> print(output_tokens)
  tensor([[   50, 29941,   402, 11571]])
>>> print (tokenizer.vocab_size)
  50257
```

Don't worry - we will implement tranformers ourself, we just need this package for token generation (we can/will do it ourself, but better to use the pre-cooked one)

- *We will consider a **vastly oversimplified** example of language translation when one token encoding an English word is sought/trained to connect to exactly one other token that encodes the corresponding Italian word*
- *In this example, both English and Italian phrases are represented by 4 tokens, each token as a number is from 1 to 50256, 0 is reserved for empty token (masked in model predictions - as you will see further...)*
- *As can be seen the word "GPT" is represented presented by 2 tokens (402 and 11571)*

Don't close the python session yet, we continue on the next slide ...

Tokens & Embedding vectors: try yourself

... continued from the previous slide

```
>>>
>>> import torch as tr
>>> embedding_layer = tr.nn.Embedding(tokenizer.vocab_size, 512)
>>> input_embeddings = embedding_layer(input_tokens)
>>>
>>> print (input_embeddings)
      tensor([[[ 1.1413, -0.0440, -0.0919,  ..., -0.5301, -0.1368, -1.3453],
               [-0.6645, -1.8470,  2.2914,  ...,  1.6478,  2.7773,  1.2885],
               [ 0.2227,  2.6014, -0.1832,  ...,  1.9270, -0.4772, -0.9424],
               [ 0.0736, -1.0592, -0.2594,  ...,  1.0837, -0.9624,  0.1389]]],
             grad_fn=<EmbeddingBackward0>)
>>>
>>> print (input_embeddings.shape)
      torch.Size([1, 4, 512])
```

We choose embedding dimension to be 512 - as in the original "Attention is all you need" paper (GPT-1 had 768)

Batch size (number of token sequence in a batch)*

Token sequence length

Embedding vector dimension

***Question**: why do we need a dedicated PyTorch `Embedding` layer and why `Linear` is not OK for this job?*

***Answer**: this is all about data format, `Linear` would fit the purpose if each toked would be presented in the format of `[0,0,0, ...1, 0,0]` sparse vectors (of dimension `vocab_size`) which is clearly not practical ...*

* Tensor operations are supposed to be done in batches (e.g. batches of token sequences) for efficient parallel execution on the hardware (e.g. GPU). In our case we have only one sequence of four tokens, which explains the first dimension

A trivial Language Model: training

Create a new file `fit_languate_model_eng_it.py` and run it:

```
import torch as tr
from transformers import  AutoTokenizer

# tokenize data
tokenizer = AutoTokenizer.from_pretrained("gpt2")
x = tokenizer("I am GPT", return_tensors="pt").input_ids
y = tokenizer("Sono GPT", return_tensors="pt").input_ids

# define the model, loss and optimizer for training
model = tr.nn.Sequential(
   tr.nn.Embedding(tokenizer.vocab_size, 512),
   tr.nn.Linear(512,tokenizer.vocab_size))
criterion = tr.nn.CrossEntropyLoss(ignore_index=0)
optimizer = tr.optim.Adam(model.parameters())

# fit the model
model.train() # training mode on
for i in range(30):
    y_pred = model(x)
    loss = criterion(y_pred.view(-1, tokenizer.vocab_size), y.view(-1))
    # to understand the above view(...) transformation - uncomment the below:
    # print (y_pred.shape, y.shape)
    # print (y_pred.view(-1, tokenizer.vocab_size).shape, y.view(-1).shape)
    loss.backward()
    optimizer.step()
    print ("Loss:",loss.item())
tr.save(model.state_dict(), 'model_eng_it.pth')
```

A trivial Language Model: training

Create a new file `fit_languate_model_eng_it.py` and run it:

```
import torch as tr
from transformers import  AutoTokenizer

# tokenize data
tokenizer = AutoTokenizer.from_pretrained("gpt2")
x = tokenizer("I am GPT", return_tensors="pt").input_ids
y = tokenizer("Sono GPT", return_tensors="pt").input_ids

# define the model, loss and optimizer for training
model = tr.nn.Sequential(
   tr.nn.Embedding(tokenizer.vocab_size, 512),
   tr.nn.Linear(512,tokenizer.vocab_size))
criterion = tr.nn.CrossEntropyLoss(ignore_index=0)
optimizer = tr.optim.Adam(model.parameters())

# fit the model
model.train() # training mode on
for i in range(30):
    y_pred = model(x)
    loss = criterion(y_pred.view(-1, tokenizer.vocab_size), y.view(-1))
    # to understand the above view(...) transformation - uncomment the below:
    # print (y_pred.shape, y.shape)
    # print (y_pred.view(-1, tokenizer.vocab_size).shape, y.view(-1).shape)
    loss.backward()
    optimizer.step()
    print ("Loss:",loss.item())
tr.save(model.state_dict(), 'model_eng_it.pth')
```

The `view(...)` operation is needed to reduce batch and sequence dimensions into a single one for the `CrossEntropyLoss` to "understand" the data(try uncommenting the corresponding print statements)*:

```
torch.Size([1, 4, 50257]) torch.Size([1, 4])
torch.Size([4, 50257]) torch.Size([4])
```

**Note that* `CrossEntropyLoss` *of PyTorch automatically acts as* `SparseCategoricalCrossentropy` *of Tensorflow*

A trivial Language Model: inference

Create a new file `eval_languate_model_eng_it.py`

```
import torch as tr
from transformers import  AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2") # tokenizer instance
model = tr.nn.Sequential(                         # define the model
   tr.nn.Embedding(tokenizer.vocab_size, 512),
   tr.nn.Linear(512,tokenizer.vocab_size)
)
model.load_state_dict(tr.load('model_eng_it.pth', weights_only=True)) # load trained model weights
model.eval() # set model in the evaluation mode
```

Now let's run it: `python -i eval_languate_model_eng_it.py`

```
>>> in_data = tokenizer("I am", return_tensors="pt").input_ids
>>> out_data = model(in_data)
>>> print (out_data.shape)
>>> print (out_data)
   torch.Size([1, 2, 50257])
   tensor([[[-6.3526, -5.8213, -5.9804,  ..., -6.5088, -5.9832, -6.1094],[-6.8054, -6.9886, -5.6512,  ...,
   -6.5140, -6.8379, -6.2876]]],grad_fn=<ViewBackward0>)
```

In principle we could run the `Softmax` transformation to convert the vector of dimension `50257` into the actual probabilities, however it is not needed since we are basically interested in the order only (`Softmax` is monotonic function) - at which position is the highest value?

A trivial Language Model: inference

Create a new file `eval_languate_model_eng_it.py`

```
import torch as tr
from transformers import  AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2") # tokenize data
model = tr.nn.Sequential(                         # define the model
   tr.nn.Embedding(tokenizer.vocab_size, 512),
   tr.nn.Linear(512,tokenizer.vocab_size)
)
model.load_state_dict(tr.load('model_eng_it.pth', weights_only=True)) # load trained model weights
model.eval() # set model in the evaluation mode
```

Now let's run it: `python -i eval_languate_model_eng_it.py`

```
>>> in_data = tokenizer("I am", return_tensors="pt").input_ids
>>> out_data = model(in_data)
>>> print (out_data.shape)
>>> print (out_data)
   torch.Size([1, 2, 50257])
   tensor([[[-6.3526, -5.8213, -5.9804,  ..., -6.5088, -5.9832, -6.1094],[-6.8054, -6.9886, -5.6512,  ...,
   -6.5140, -6.8379, -6.2876]]],grad_fn=<ViewBackward0>)

>>> out_indexmax = tr.argmax(out_data,dim=-1)
>>> print (out_indexmax.shape)
   torch.Size([1, 2])
>>> print (out_indexmax)
   tensor([[   50, 29941]])
>>> print (tokenizer.batch_decode(out_indexmax))
   ['Sono']
```

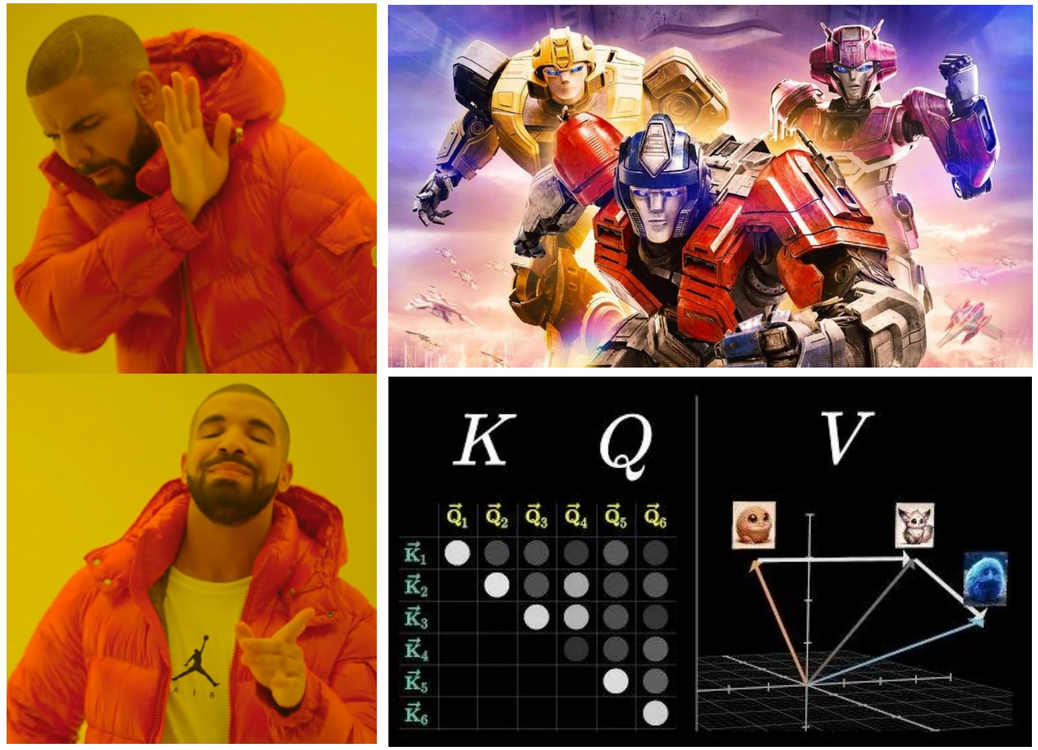
Et voilà!
We see our Eng-It model in action!

A trivial Language Model: inference (recap)

Update `eval_languate_model_eng_it.py` to have everything in one place (will need it for later...)

```
import torch as tr
from transformers import  AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2") # tokenize data
model = tr.nn.Sequential(                         # define the model
  tr.nn.Embedding(tokenizer.vocab_size, 512),
  tr.nn.Linear(512,tokenizer.vocab_size)
)
model.load_state_dict(tr.load('model_eng_it.pth', weights_only=True)) # load trained model
weights
model.eval() # set model in the evaluation mode

# example of model inference
in_data = tokenizer("I am", return_tensors="pt").input_ids
out_data = model(in_data)
out_indexmax = tr.argmax(out_data,dim=-1)
human_readable_output = tokenizer.batch_decode(out_indexmax)
```

Coming back to Transformers...

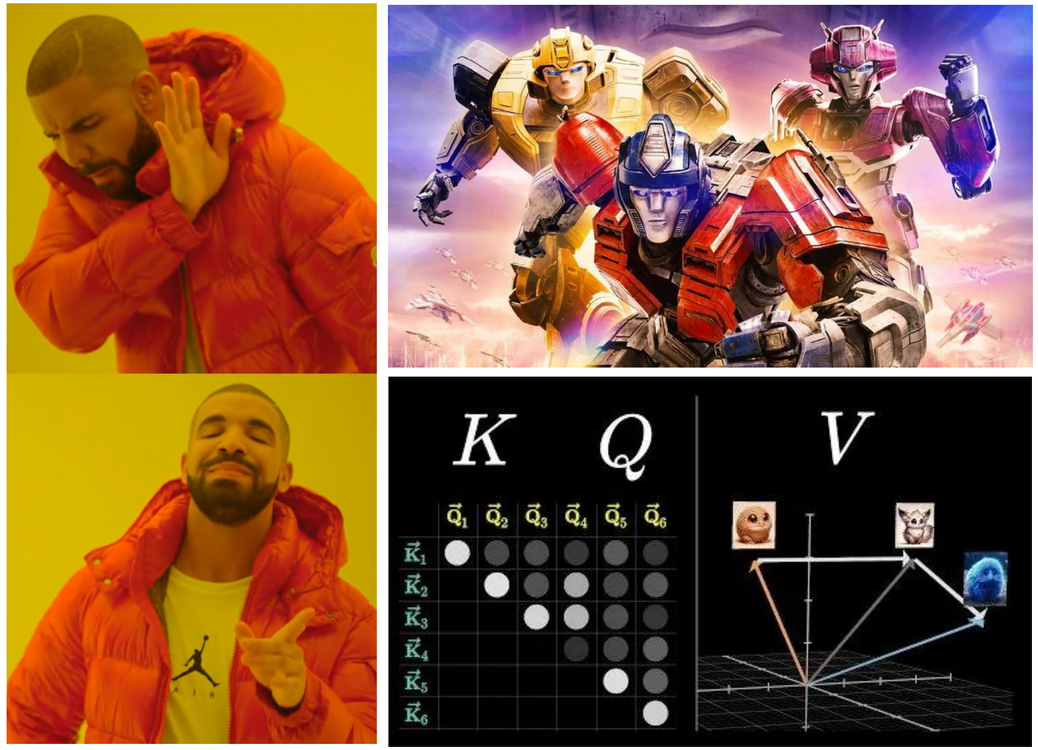


Now as we (hopefully) have some intuition regarding the Language models, we can get back to the Transformers. **Transformer is a model architecture.** In the previously considered example we used a trivial model architecture with 2 layers only:

- Embedding (tokens --> embedding vectors)
- Linear (embedding vectors --> token probabilities)

```
model = tr.nn.Sequential(
   tr.nn.Embedding(tokenizer.vocab_size, 512),

   tr.nn.Linear(512,tokenizer.vocab_size)
)
```

Coming back to Transformers...



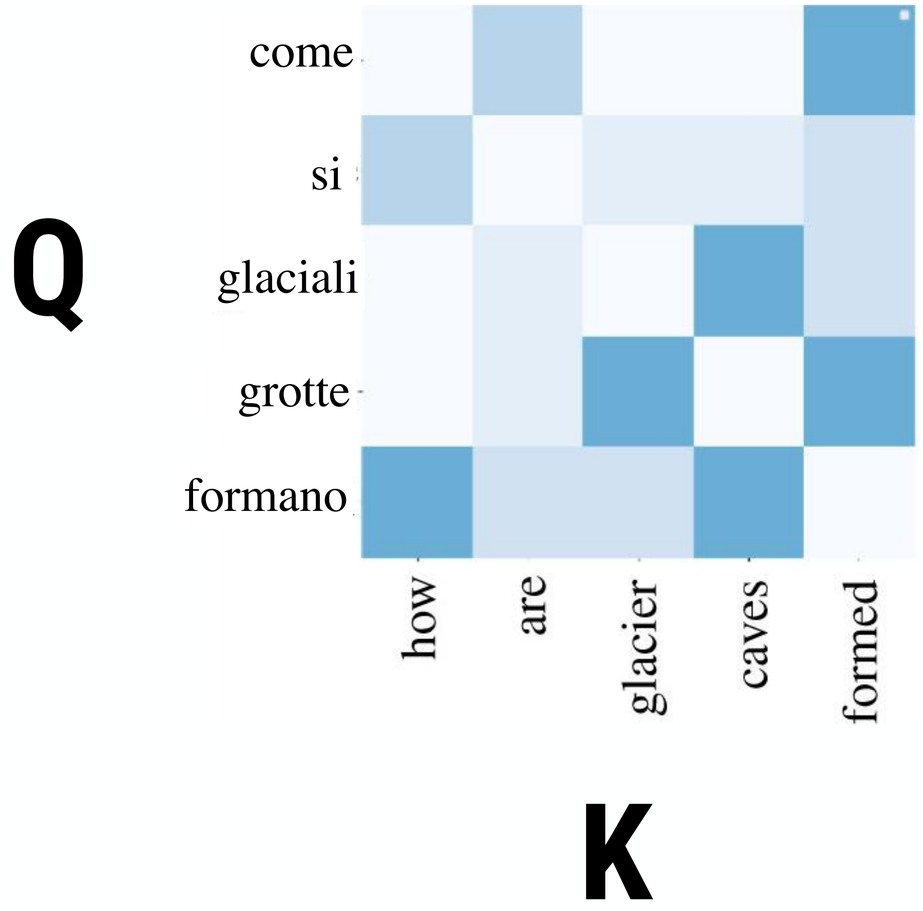
Now as we (hopefully) have some intuition regarding the Language models, we can get back to the Transformers. **Transformer is a model architecture.** In the previously considered example we used a trivial model architecture with 2 layers only:

- Embedding (tokens --> embedding vectors)
- **Transformer**
- Linear (embedding vectors --> token probabilities)

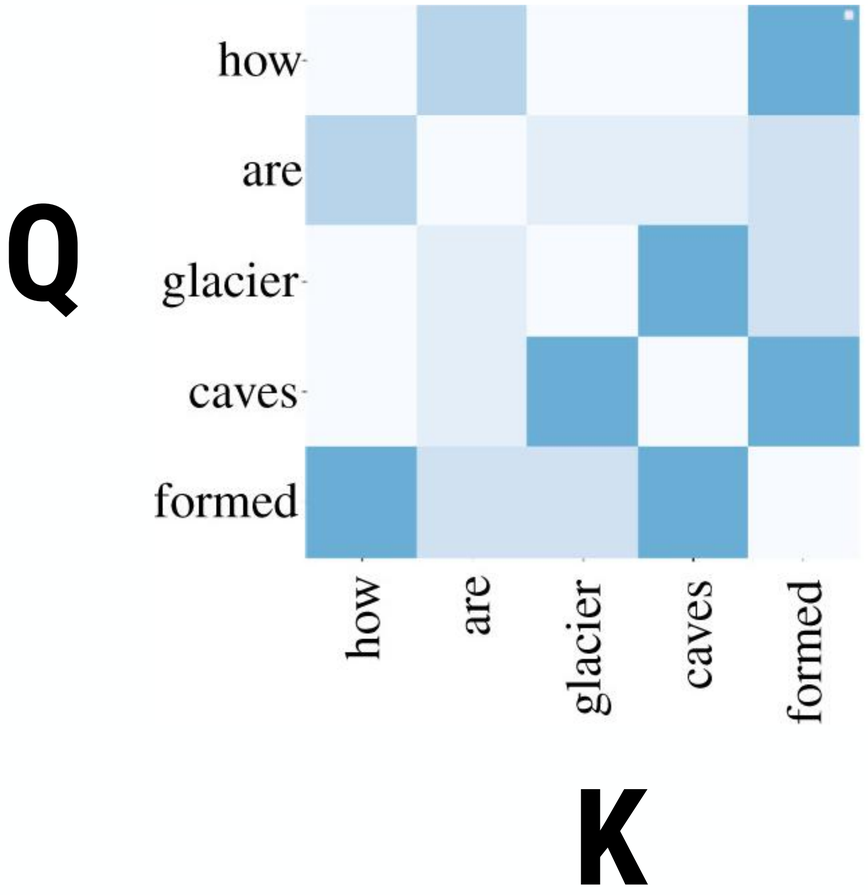
```
model = tr.nn.Sequential(
   tr.nn.Embedding(tokenizer.vocab_size, 512),
   Transofmer
   tr.nn.Linear(512,tokenizer.vocab_size)
)
```

Transformers are about the attention matrix

Cross-attention



Self-attention

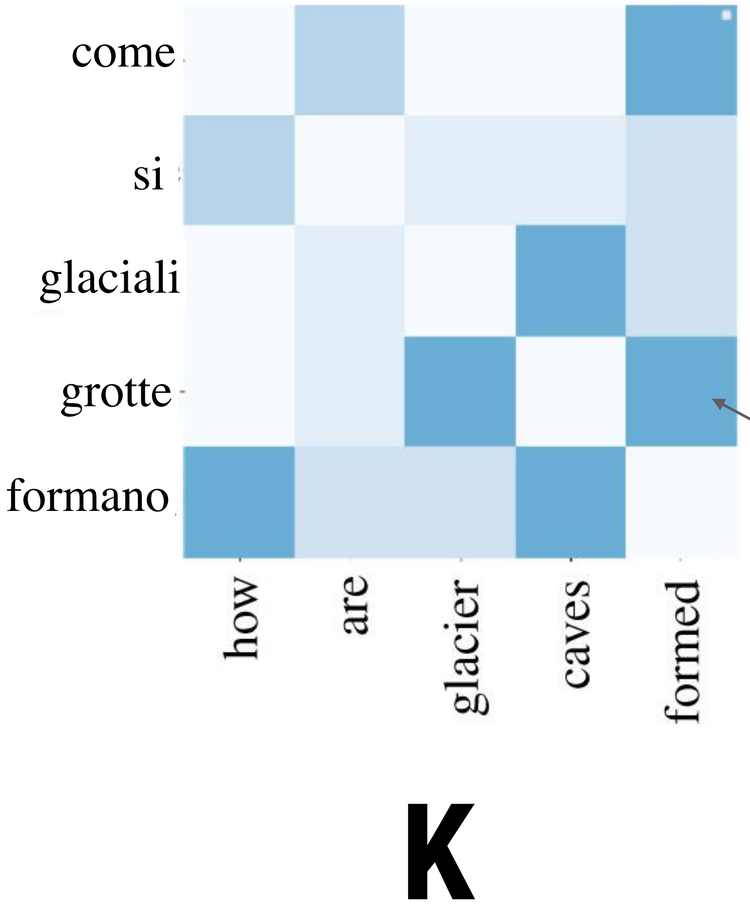


Q = query, K = key

See also: https://fall-2023-python-programming-for-data-science.readthedocs.io/en/latest/Lectures/Theme_3-Model_Engineering/Lecture_20-Transformer_Networks/Lecture_20-Transformer_Networks.html

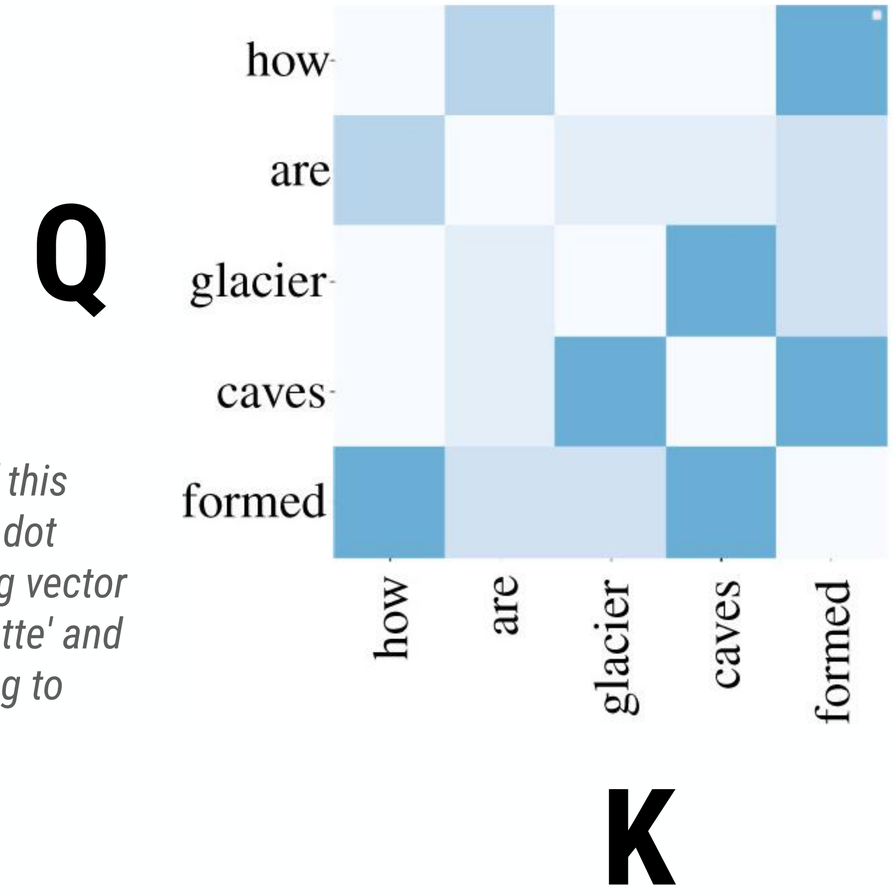
Transformers are about the attention matrix

Cross-attention



For example, think of this specific number as a dot product of embedding vector corresponding to 'grotte' and the one corresponding to 'formed'

Self-attention

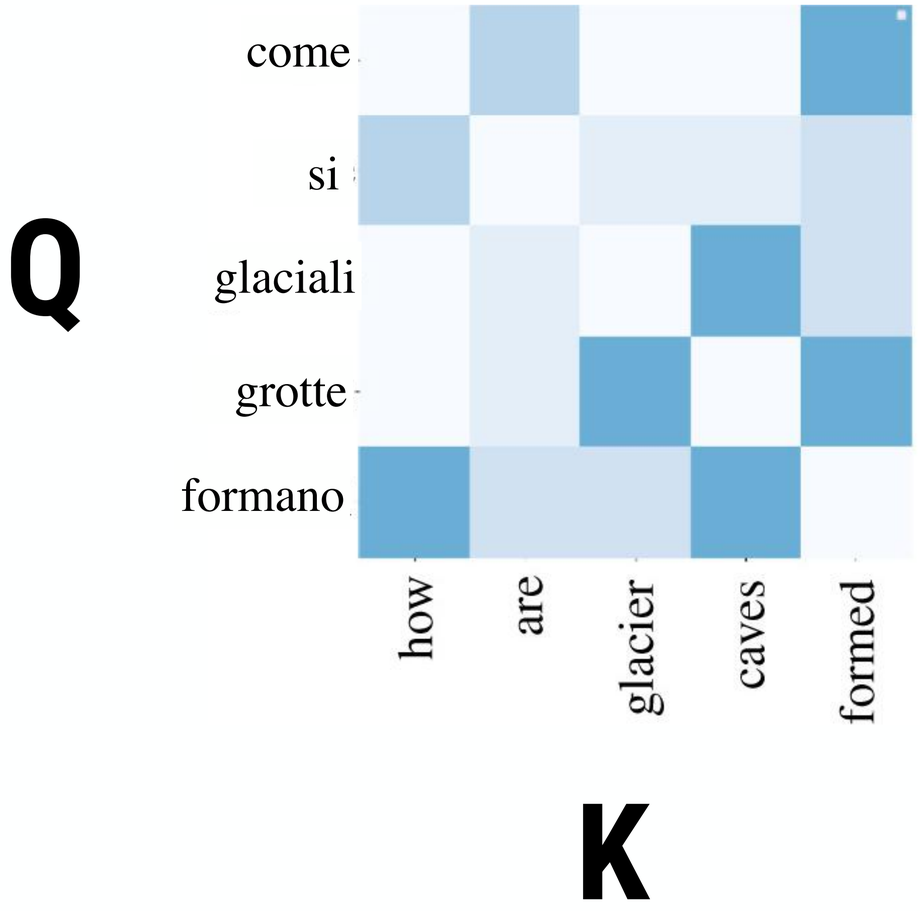


Q = query, K = key

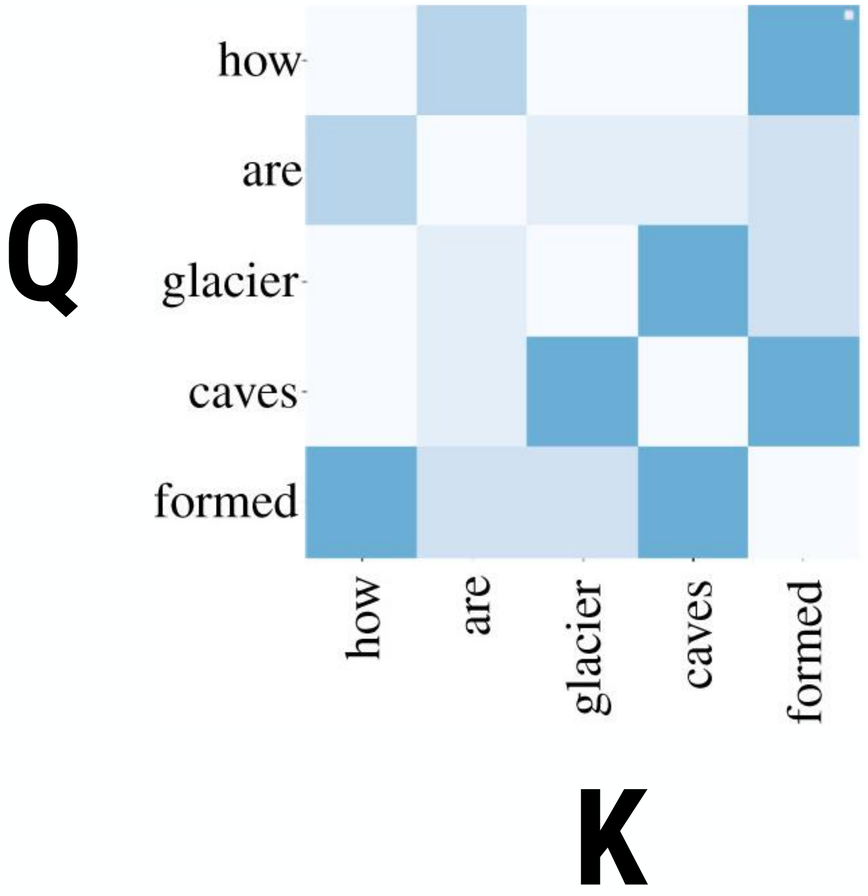
See also: https://fall-2023-python-programming-for-data-science.readthedocs.io/en/latest/Lectures/Theme_3-Model_Engineering/Lecture_20-Transformer_Networks/Lecture_20-Transformer_Networks.html

Transformers are about the attention matrix

Cross-attention

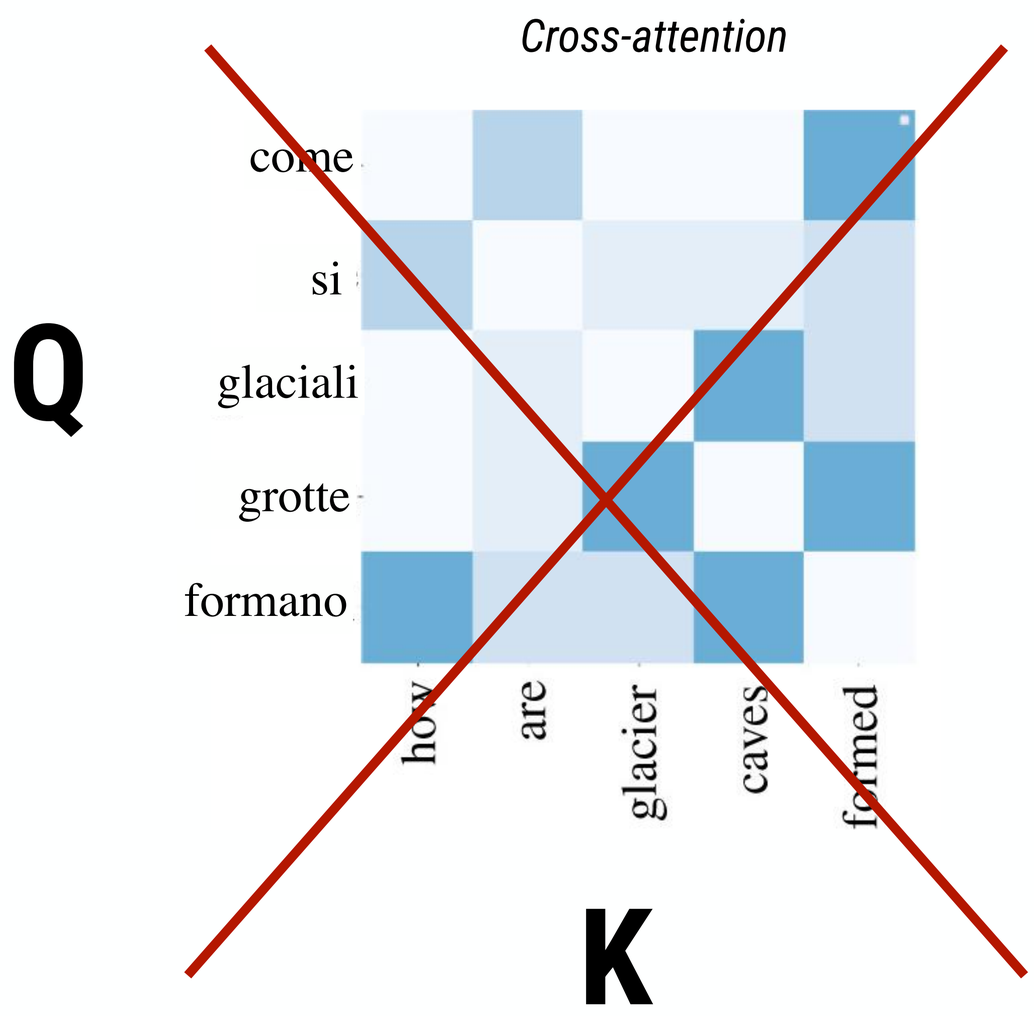


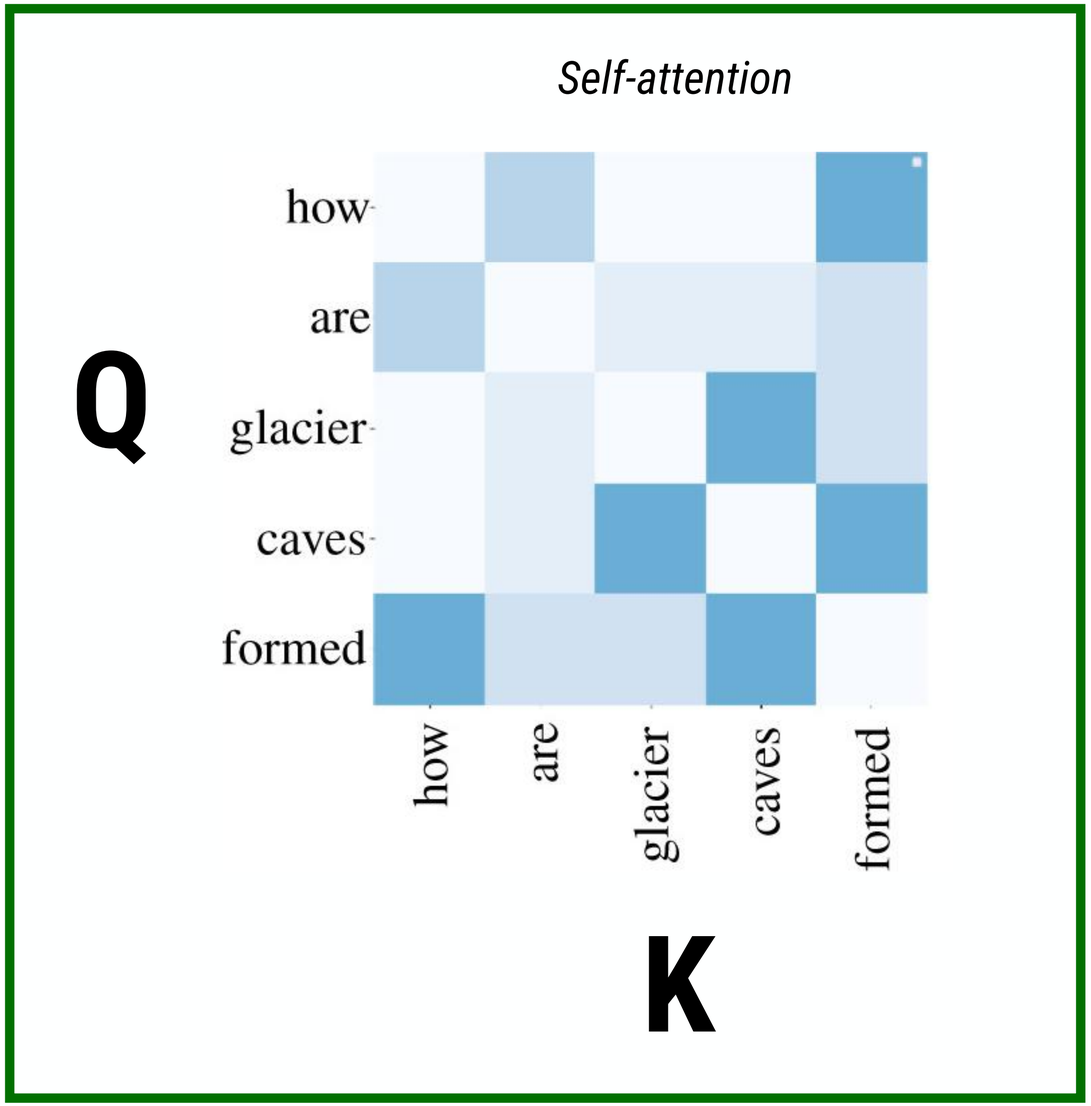
Self-attention



*Both cross-attention and self-attention is used in **Language translation problems** (original model in "Attention is all you need" paper)*

Transformers are about the attention matrix

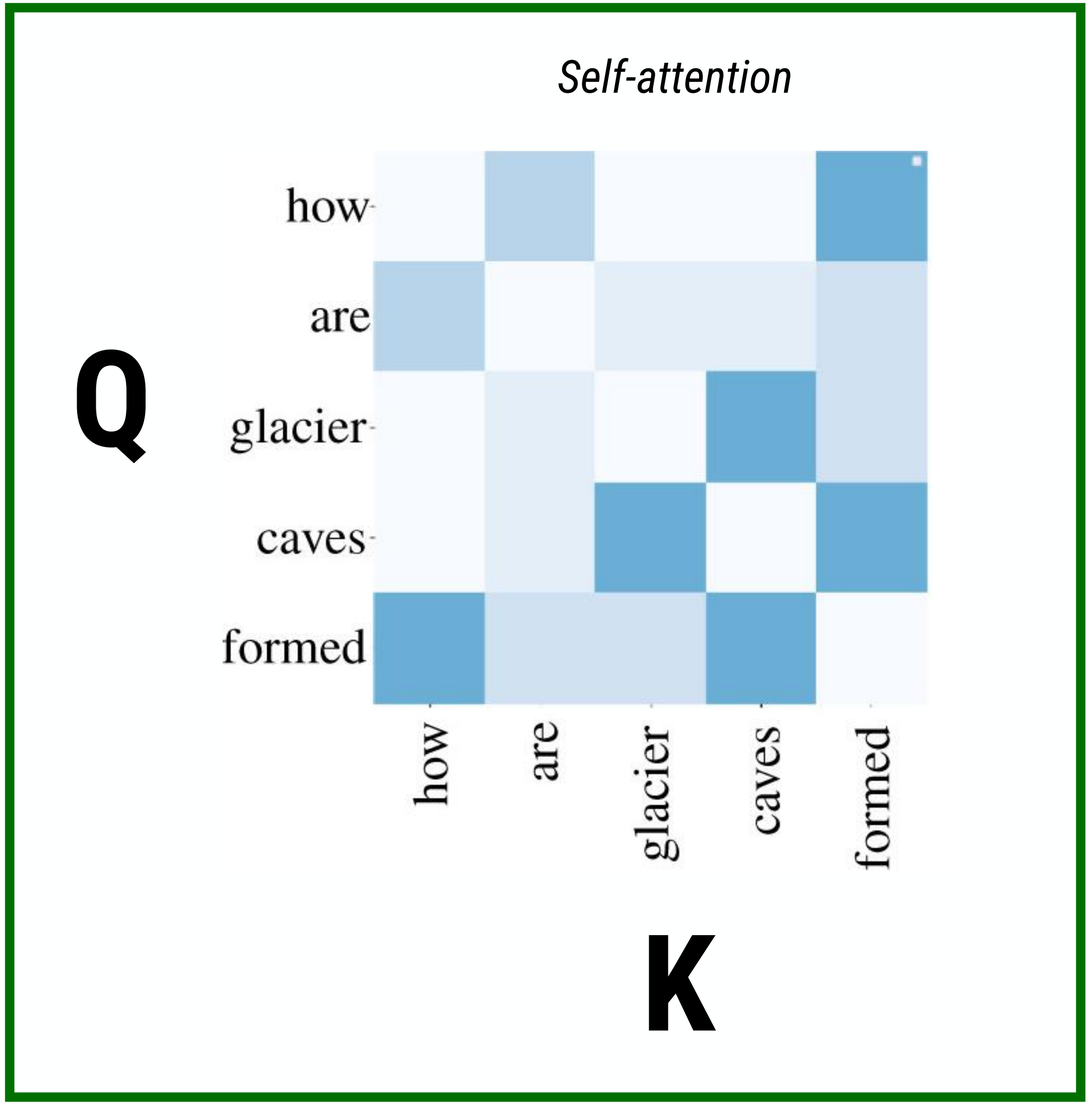




In chatbot models like GPT only self-attention is used (unless it deals with translation)

Transformers are about the attention matrix

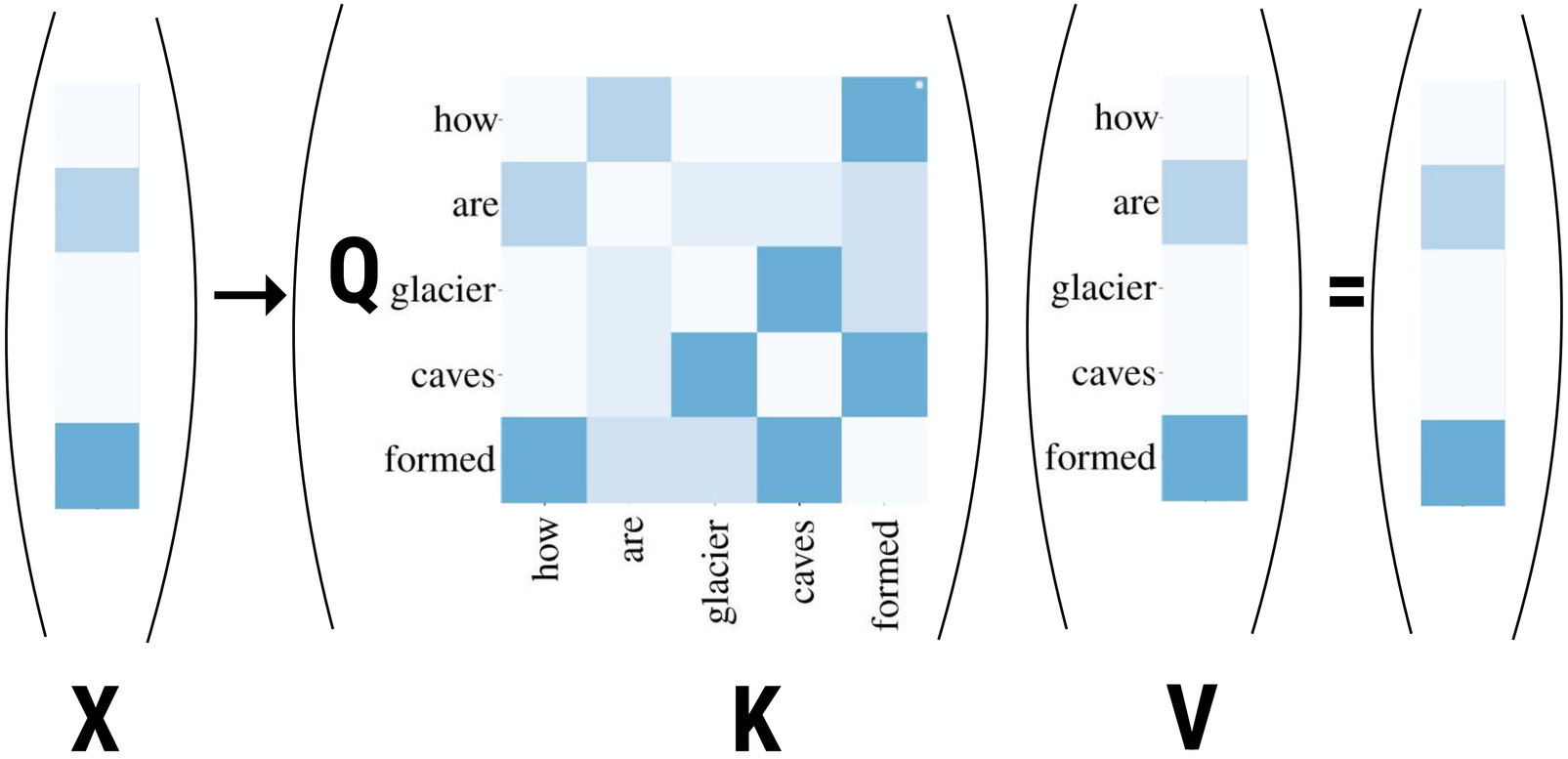
For clarity, let's not consider cross-attention and focus on self-attention only (all basic concepts are valid ~ identical for both cases)



In chatbot models like GPT only self-attention is used (unless you ask it for translation of something)

Transformers are about the attention matrix

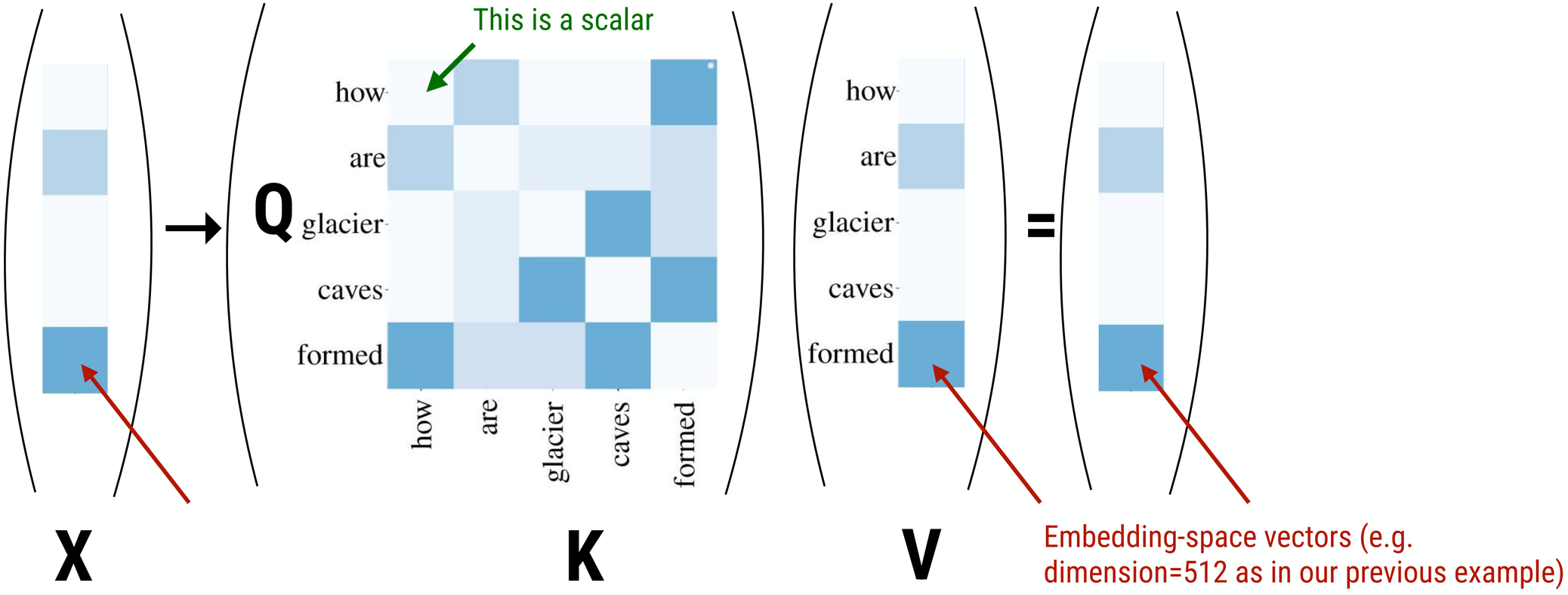
Input sequence of embeddings (e.g. "how", "are", "glacier", "caves", "formed") is transformed through attention matrix (**attention block**):



X - input; for now, think of **Q = K = V = X** (sequence of input embedding vectors)

Transformers are about the attention matrix

Input sequence of embeddings (e.g. "how", "are", "glacier", "caves", "formed") is transformed through attention matrix (**attention block**):

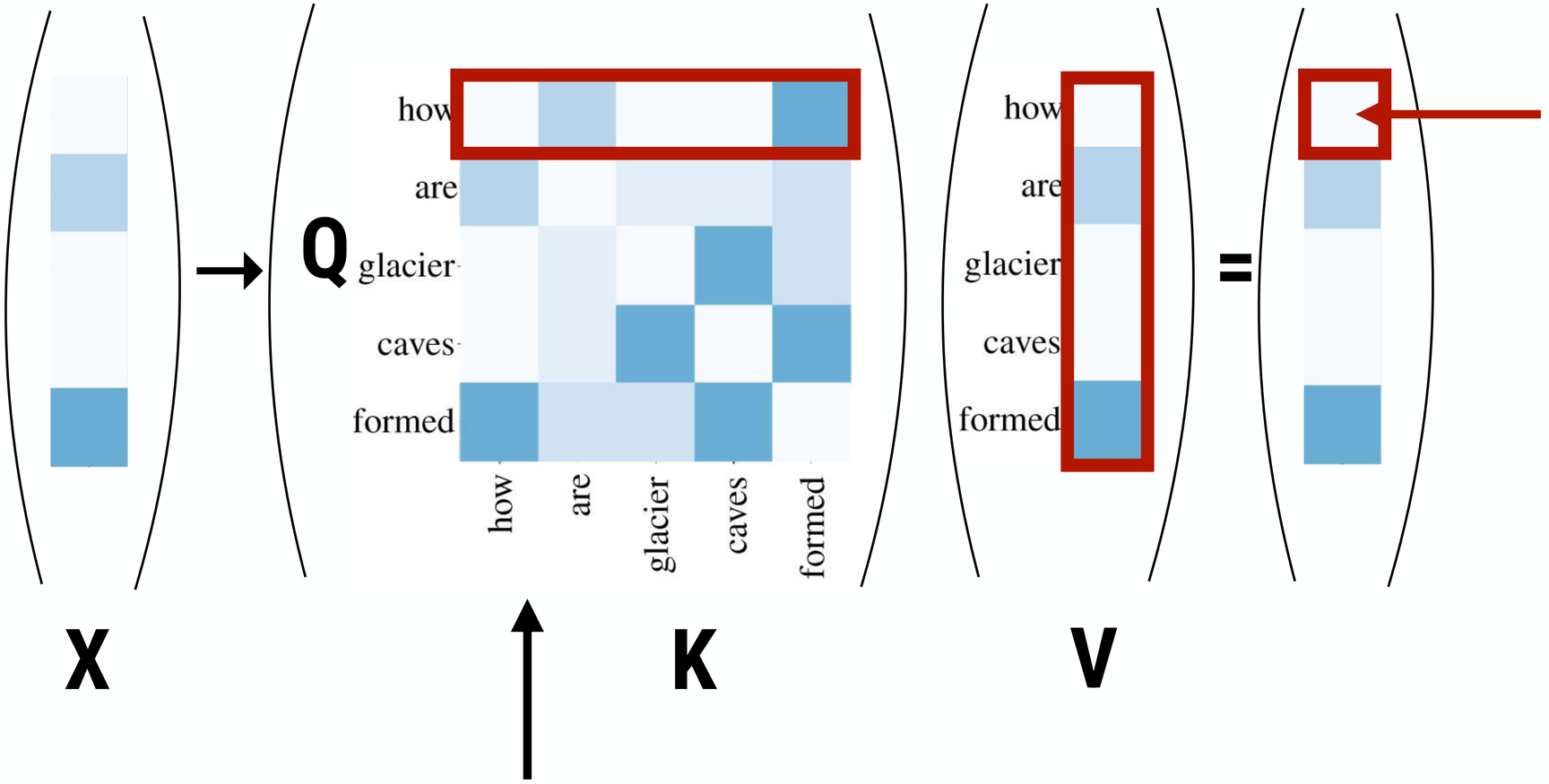


X - input; for now, think of **Q = K = V = X** (sequence of input embedding vectors)

For clarity, imagine that the Embedding space has only 1 dimension (not 512 or whatever...)

Transformers are about the attention matrix

Input sequence of embeddings (e.g. "how", "are", "glacier", "caves", "formed") is transformed through attention matrix (**attention block**):



For example, let's consider the value of first element, it equals:

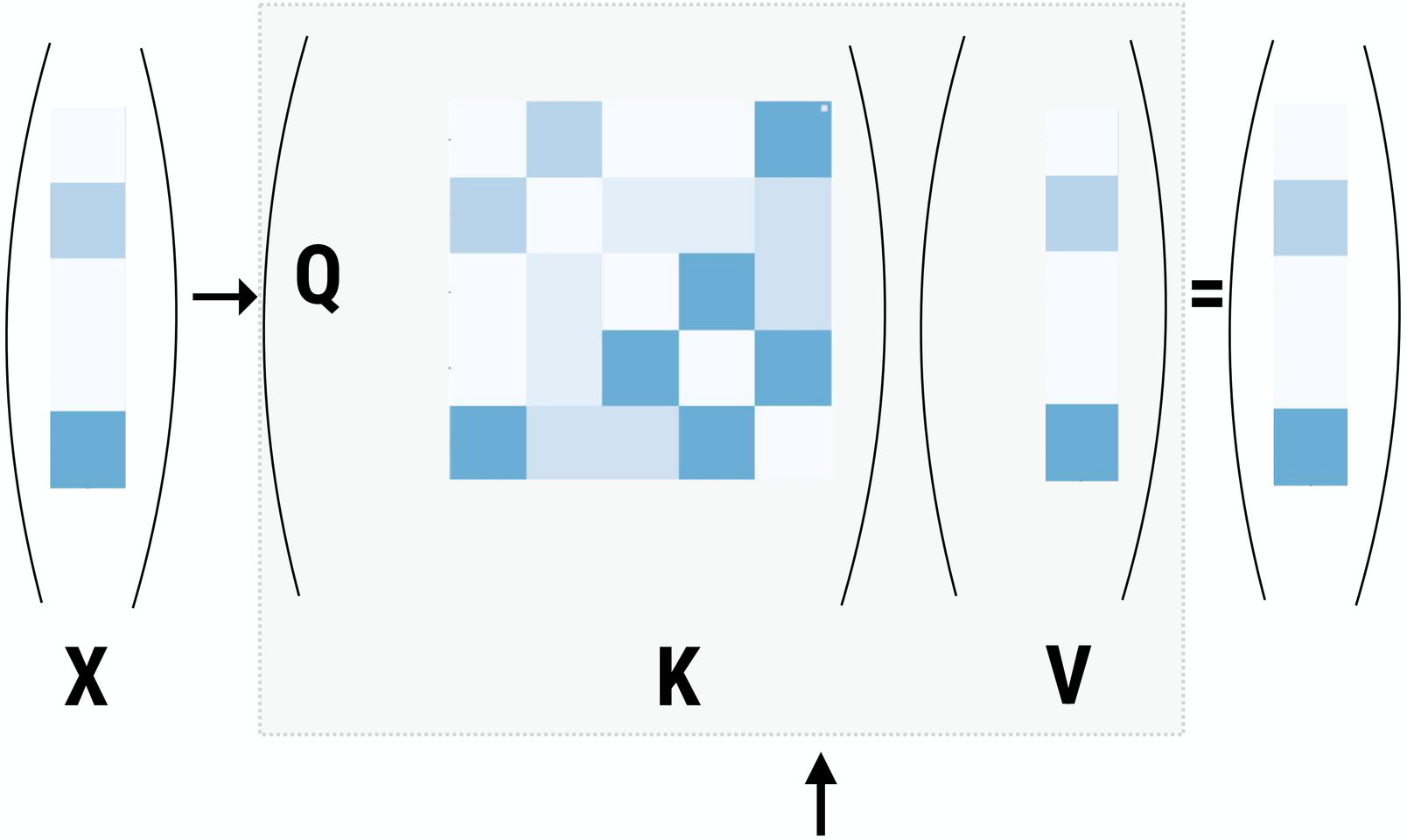
```
QK(how-how)       * Embed(how)   +
QK(how-are)       * Embed(are)   +
QK(how-glacier)   * Embed(glacier)   +
QK(how-caves)     * Embed(caves)   +
QK(how-formed)    * Embed(formed)   +
```

Scalar (element of QK matrix)

Embedding vector

Transformers are about the attention matrix

Input sequence of embeddings (e.g. "how", "are", "glacier", "caves", "formed") is transformed through attention matrix (**attention block**):



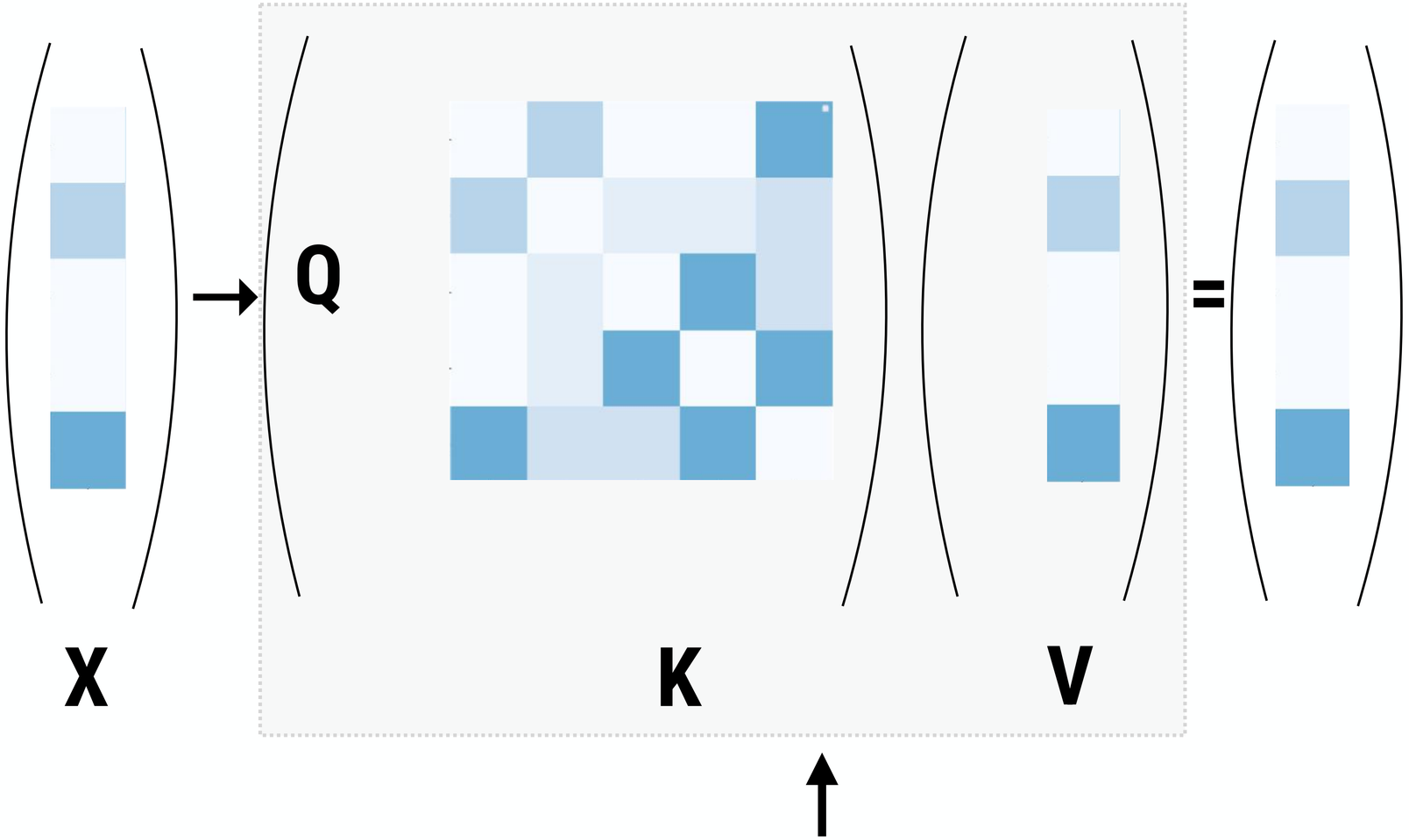
Output of the Transformer - same structure as the input: a sequence of embedding-space vectors (5 vectors in our example) but...

Each "embedding" is not merely the one corresponding to the word/token itself, but is rather a sum of all original embeddings weighted according to their correlation/attention

Attention Transformation block (in analogy with convolutional filter of CNN)

Transformers are about the **Attention** matrix

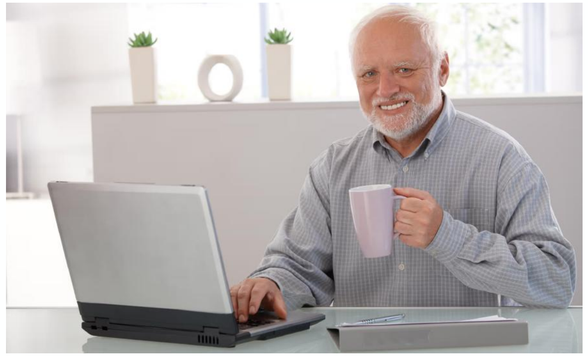
Input sequence of embeddings (e.g. "how", "are", "glacier", "caves", "formed") is transformed through attention matrix (**attention block**):



Attention Transformation block (in analogy with convolutional filter of CNN)

Output of the Transformer - same structure as the input: a sequence of embedding-space vectors (5 vectors in our example) but...

Each "embedding" is not merely the one corresponding to the word/token itself, but is rather a sum of all original embeddings weighted according to their correlation/attention



Attention: **queries (Q), keys (K), values (V)**

Now where is the catch?

- Before we considered **Q=K=V** = (sequence of embedding vectors each one corresponding to an input token)
  - ➡If that would be the case, the only way were model could learn something is in the Embedding layer (the one that translates tokens into embedding vectors) - which would barely be enough for any real-life application

- **In reality is that Q, K, V are not merely the input embeddings, but linearly-transformed input embeddings!**

$$Q = W_Q \cdot X \quad K = W_K \cdot X \quad V = W_V \cdot X$$

W_q, W_k, W_v - **n x n** matrices where n is the number of dimensions in embedding space (512 x 512 in our example)

X - input embeddings (5 x 512 matrix in our example of "how", "are", "glacier", "caves", "formed")
Q - query-transformed "embeddings" (5 x 512 matrix ...)
K - key-transformed "embeddings" (5 x 512 matrix ...)
V - value-transformed "embeddings" (5 x 512 matrix ...)

Attention: **queries (Q), keys (K), values (V)**

Now where is the catch?

- Before we considered **Q=K=V** = (sequence of embedding vectors each one corresponding to an input token)
  - ➡ If that would be the case, the only way were model could learn something is in the Embedding layer (the one that translates tokens into embedding vectors) - which would barely be enough for any real-life application

- **In reality is that Q, K, V are not merely the input embeddings, but linearly-transformed input embeddings!**

$$Q = W_Q \cdot X \quad K = W_K \cdot X \quad V = W_V \cdot X$$

Important!

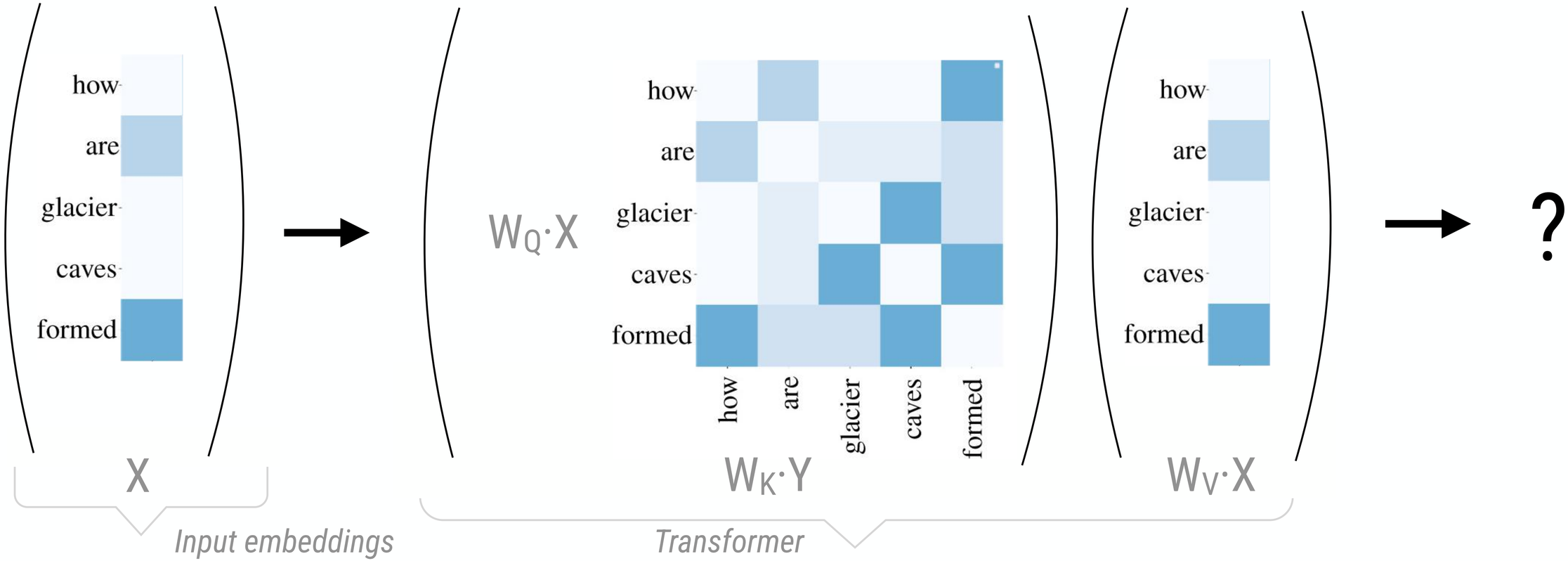
*The **W_Q, W_K, W_V** (**query, key, value**) matrices are trainable parameters of a Transformer model (**in analogy with convolutional filters of CNN**) - that is where the model encodes much of its knowledge!*

In PyTorch (given our example) the three matrices are implemented with the `Linear` layers:

```
W_Q = tr.nn.Linear(512, 512)
W_K = tr.nn.Linear(512, 512)
W_V = tr.nn.Linear(512, 512)
```

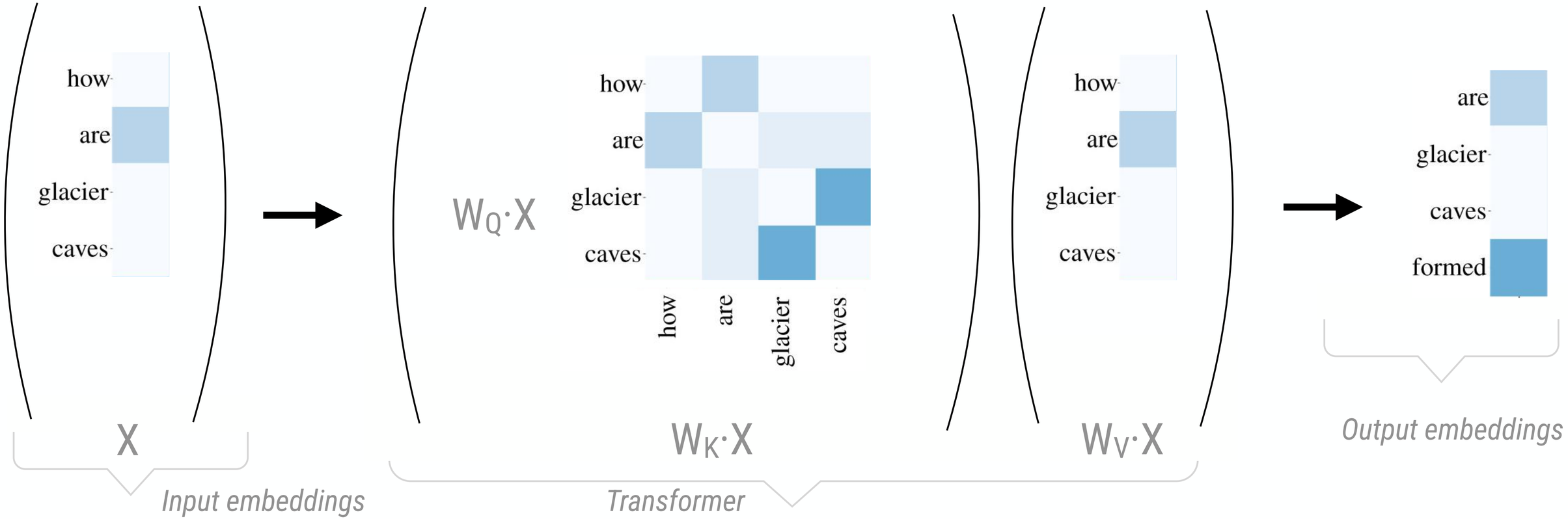
Attention: **queries (Q), keys (K), values (V)**

- Okay, the transformer takes as an input a sequence of embedding corresponding to tokens ("how", "are", "glacier", "caves", "formed")
- What do we want to predict? The answer is: **for every input token we train the model to predict next one in the sequence...**



Transformer training: input & output

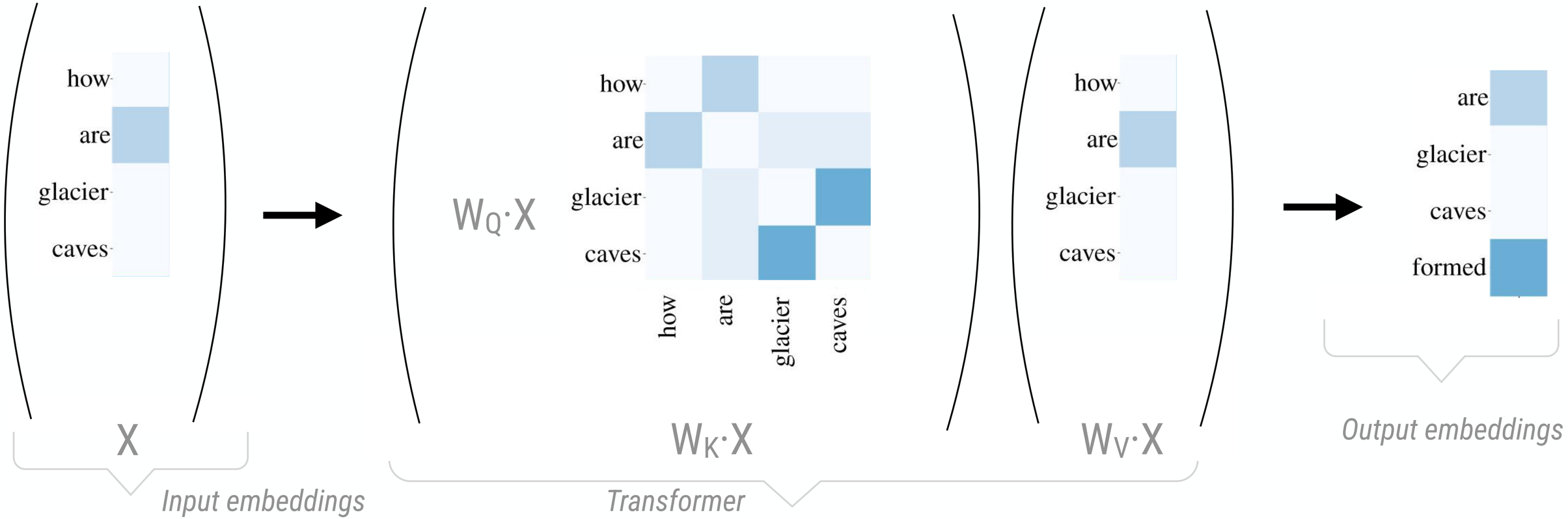
- Okay, the transformer takes as an input a sequence of embedding corresponding to tokens ("how", "are", "glacier", "caves", "formed")
- What do we want to predict? The answer is: **for every input token we train the model to predict next one in the sequence...**



We removed the last token/embedding from the input ("formed"), while the target output (used to trained the model) is the same as input but shifted by one position ("how" removed in the beginning).
Remember: number if input and output tokens in self-attention models is always the same!

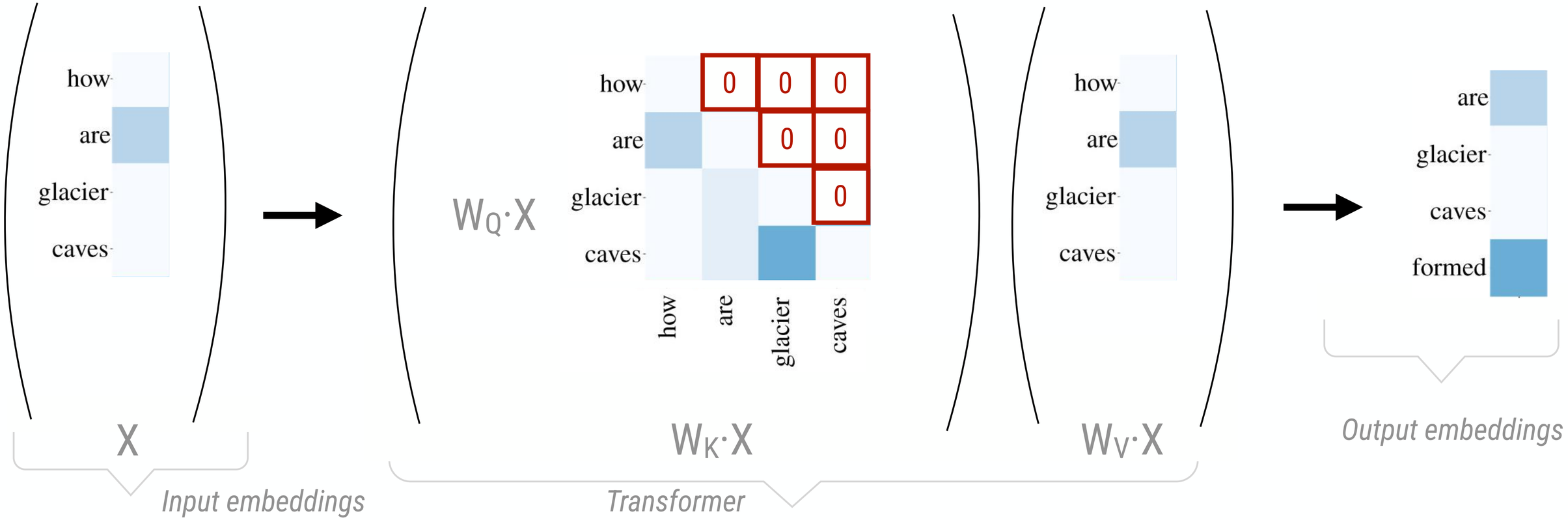
Transformer training: input & output

- To avoid the model "looking into the future" at training, we mask elements below the main diagonal in the attention matrix



Transformer training: input & output

- To avoid the model "looking into the future" at training, we mask elements below the main diagonal in the attention matrix



For example, the output of transformer for the word "glacier" will be a weighted sum of embeddings of word "glacier" itself and the words preceding in the sequence ("are","how") but not those after ("caves"). Otherwise, the model will just pick up (learn) to use the features of a word/token that comes right after - it will trivially work to predict n-1 tokens out of n, but will yield nonsense for the last one (as it has no information about the word that come after)

Transformer inference: input & output

During the inference, we ask model to iteratively predict one toke at at time:

- First call of the model:
  - ➡ Inputs: **["how"]**
  - ➡ Outputs: ["are"]
- Second call of the model:
  - ➡ Inputs: **["how", "are"]**
  - ➡ Outputs: ["are", "glacier"]
- Third call of the model:
  - ➡ Inputs: **["how", "are", "glacier"]**
  - ➡ Outputs: ["are", "glacier", "caves"]
- Fourth call of the model:
  - ➡ Inputs: **["how", "are", "glacier", "caves"]**
  - ➡ Outputs: ["are", "glacier", "caves", "formed"]

etc. (we can, in principle, continue forever ...)

Transformer inference: input & output (**cross-attention**)

During the inference, we ask model to iteratively predict one toke at at time:

Special start-of-sequence token -- it is required since, by construction, one Italian input token will correspond to exactly one Italian output token (number of English inputs/tokens can be whatever, in the end of translation the number of Italian tokens might be different from the English ones)

- First call of the model:
  - ➡ Inputs: **["how", "are", "glacier", "caves", "formed"], [<s>]**
  - ➡ Outputs: **["come"]**
- Second call of the model:
  - ➡ Inputs: **["how", "are", "glacier", "caves", "formed"], [<s>, "come"]**
  - ➡ Outputs: **["come","si"]**
- Third call of the model:
  - ➡ Inputs: **["how", "are", "glacier", "caves", "formed"], [<s>, "come", "si"]**
  - ➡ Outputs: **["come","si", "glaciali"]**
- Fourth call of the model:
  - ➡ Inputs: **["how", "are", "glacier", "caves", "formed"], [<s>, "come","si","glaciali"]**
  - ➡ Outputs: **["come","si", "glaciali", "grotte"]**
- Fifth call of the model (we don't predict, we translate the sentence):
  - ➡ Inputs: **["how", "are", "glacier", "caves", "formed"], [<s>, "come","si","glaciali", "grotte"]**
  - ➡ Outputs: **["come","si", "glaciali", "grotte","formatto"]**
- Fifth call of the model (we don't predict, we translate the sentence):
  - ➡ Inputs: **["how", "are", "glacier", "caves", "formed"], [<s>, "come","si","glaciali", "grotte", "fomratto"]**
  - ➡ Outputs: **["come","si", "glaciali", "grotte","formatto", <e>]**

If trained well, the model will likely yield a stop-sequence token telling us that it no more tokens are needed and the English sentence translation is completed


Enough (almost) theory, let's get
back to coding...

Attention block code

Code adapted from: https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch

Create `transformer_torch.py` :

```
import torch as tr
import math
class Attention(tr.nn.Module):
    def __init__(self, d_model):
        super().__init__()
        # Linear layers for transforming inputs
        self.W_q = tr.nn.Linear(d_model, d_model) # Query transformation
        self.W_k = tr.nn.Linear(d_model, d_model) # Key transformation
        self.W_v = tr.nn.Linear(d_model, d_model) # Value transformation

        self.d_model = d_model  # embedding dimension

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # Calculate attention scores
        attn_scores = tr.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_model)

        # Apply mask (for example the under-diagonal elements)
        if mask is not None: attn_scores = attn_scores.masked_fill(mask==0,-1e9)

        # Softmax is applied to obtain attention probabilities
        attn_probs = tr.softmax(attn_scores, dim=-1)

        # Multiply by values to obtain the final output
        return tr.matmul(attn_probs, V)

    def forward(self, x, mask=None):
        # Apply linear transformations and split head
        Q = self.W_q(x)
        K = self.W_k(x)
        V = self.W_v(x)
        # Perform scaled dot-product attention and return
        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        return attn_output
```

We already know W_k, W_q, W_v matrices

- Calculate attention (QK) matrix
- Apply mask (for example QK elements below the main diagoanal)
- Obtained QK matrix embedding vector (divided by `d_model` to normalize typical value and avoid growth with large `d_model`)
- Softamax is applied to normalize to QK matrix element to probabilities

Propagate our input sequence through the attention block

Attention block code: test

Create `test_attention_block_torch.py` :

```
import torch as tr
from transformer_torch import Attention
from transformers import  AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
attentionblock = Attention(512)

x = tr.randint(1, tokenizer.vocab_size-1, (32, 100))
print (x.shape)
x_embed = tr.nn.Embedding(tokenizer.vocab_size, 512)(x)
print (x_embed.shape)

x_attention = attentionblock(x_embed)
print (x_embed.shape)
```

We generate a batch of random data 32 sequences of 100 tokens each. Then we convert it to embedding vectors

We process the embedding vectors of our input data through the attention block, in the end we should get the output sequence if the same dimension

`python test_attention_block_torch.py` :

```
torch.Size([32, 100])
torch.Size([32, 100, 512])
torch.Size([32, 100, 512])
```

?

Exercise (5 mins): **understanding of the Attention**

Look inside the function `scaled_dot_product_attention` **of the** `Attention` **class**

➡ Insert print statements to show the dimension (shape) of `Q, K, V, attn_scores, attn_probs`

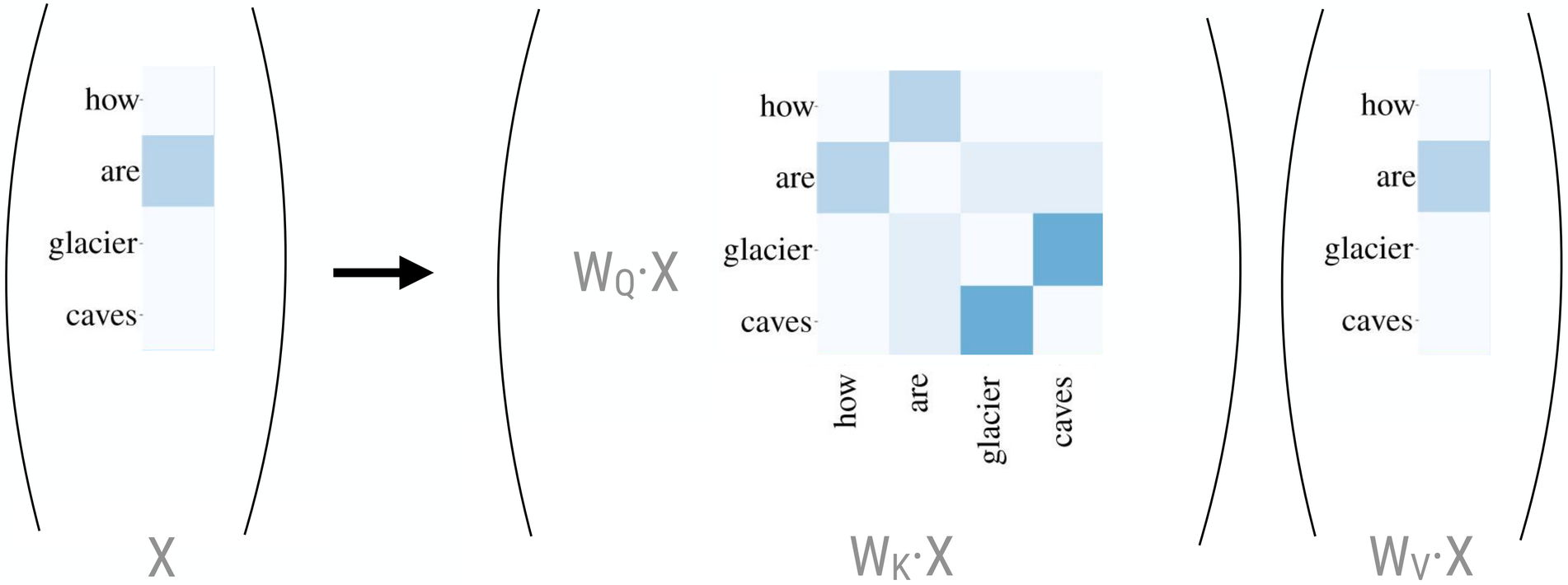
... Do you understand those?

➡ Print the values of `attn_scores` and `attn_probs` matrices

... Does the second one look like probabilities?

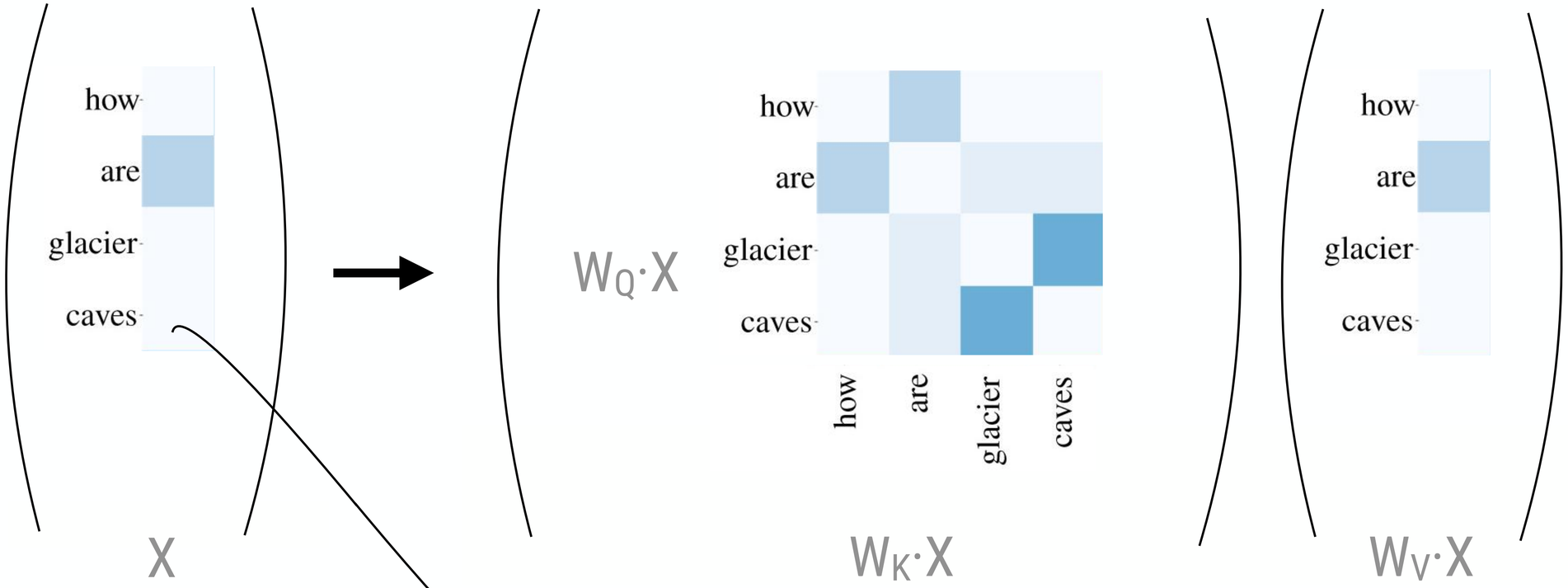
Multi-head attention

- In reality, each transformer block has multiple attention heads. What does that mean? Consider our example:



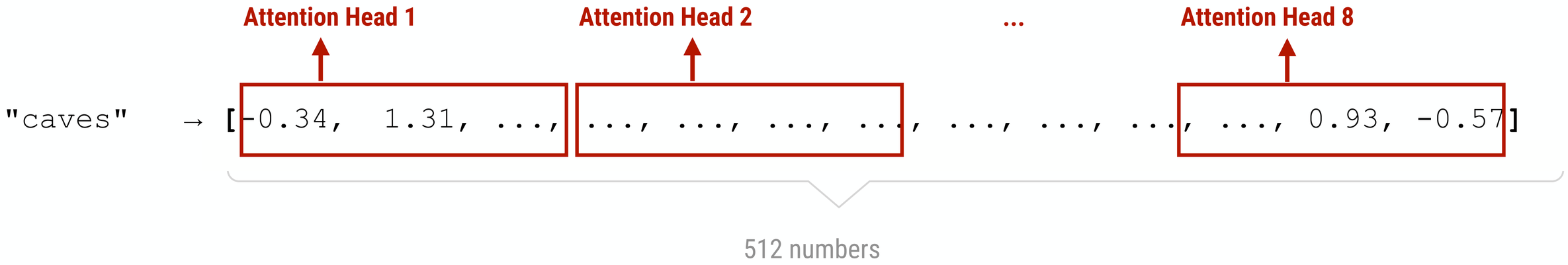
Multi-head attention

- In reality, each transformer block has multiple attention heads. What does that mean? Consider our example:



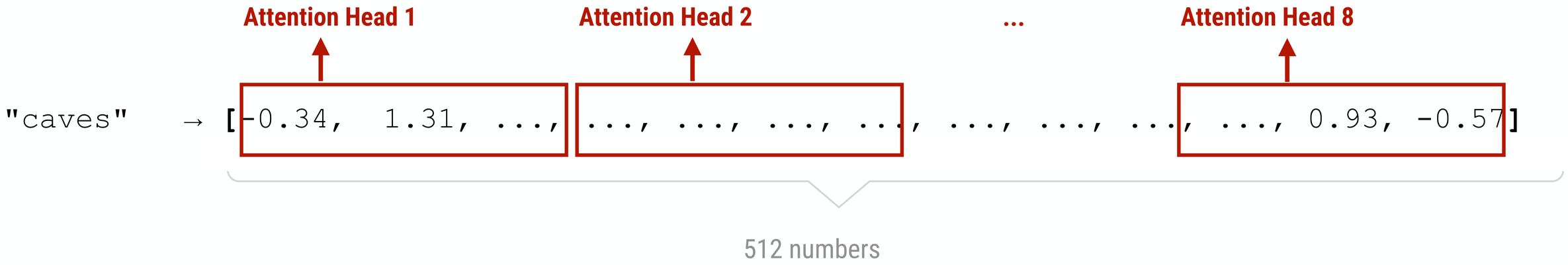
Each element here is an embedding vector (dimension 512 in our example)

Multi-head attention



Vectors in embedding space are split into equal size parts processed separately each by its own Transformer (Key-Query-Value transformation), which is called a "head". In a canonical "Attention is all you need" paper, there are 8 heads per attention block, which makes the dimension of sub-embedding for each head 512/8 = 64.

Multi-head attention



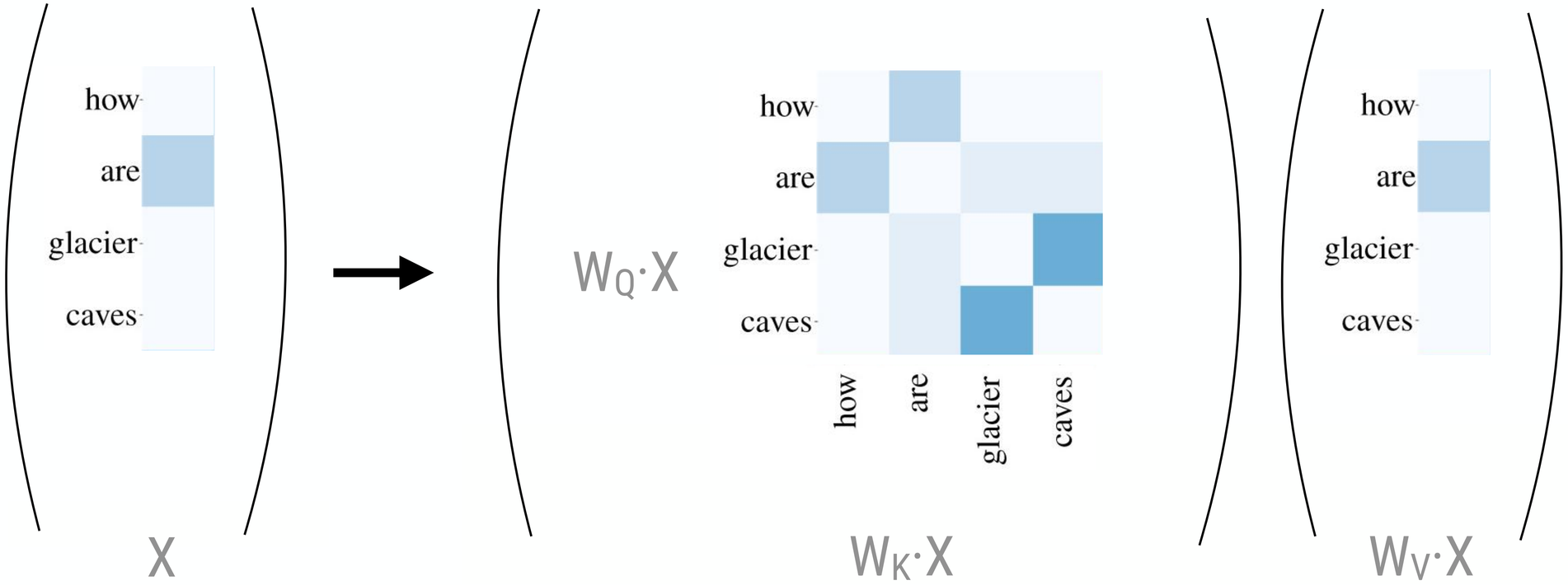
Vectors in embedding space are split into equal size parts processed separately each by its own Transformer (Key-Query-Value transformation), which is called a "head". In a canonical "Attention is all you need" paper, there are 8 heads per attention block, which makes the dimension of sub-embedding for each head 512/8 = 64.

In practice, the algorithm is as follows:

1. Apply key/value/query transformation to each embedding
2. Split embeddings into multiple equal-length parts (8 pars, 64 each - in the original paper)
3. Apply attention transformation for each sub-embedding (head) (8 attention blocks)
4. Combine the outputs of 8 transformations back to the original-size embedding vector (dimension 512 in our example)

Multi-head attention

Main advantage if multi-head attention is that at the ~same computation complexity, the model can encode different attention patterns in through different heads in one attention block



For example, the first head may converge towards focusing on correlation in the beginning and the end of the phrase, putting more weight to those ("how" and "caves" in our example), while the other head may focus more on attention patters between subsequent words ("are", "glacier") etc. Remember that these attention patterns are encoded in the W_Q, W_K, W_V, matrices...

Multi-head attention block: code

Now let's update the `transformer_torch.py` code (make a back-up copy of it before!):

```
#class Attention(tr.nn.Module):                 # before
class MultiHeadAttention(tr.nn.Module):         # now
    #def __init__(self, d_model):              # before
    def __init__(self, d_model, num_heads):    # now
        super().__init__()
        ... everything before remains the same, add the following in the end of __init__:
        self.num_heads = num_heads                  # Number of attention heads
        self.d_k = d_model // num_heads             # Dimension of each head's key, query, value
        self.W_o = tr.nn.Linear(d_model, d_model) # Mixing matrix (we'll understand it later)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        ... everything the same except for replacing "d_model" with "d_k":
        #attn_scores = tr.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_model)
        attn_scores = tr.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

    def forward(self, x, mask=None):
        ...
        # add the following code to split heads after the Q,V,K definition
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        ...
        # add the following after the "attn_output" definition and before "return":
        attn_output = self.combine_heads(attn_output)    # combine heads
        attn_output = self.W_o(attn_output)              # mix heads


    ... contiunued on the next pate (we need to implement "split_heads" and "combine_heads" methods )
```

We need to define number of heads and also to create a "mixing" matrix to mix the heads in the end

Replace "d_model" with "d_k"

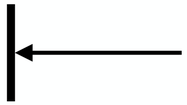
We first split the Q, K, V vectors into parts (8 parts 64 length each in our example) to be processed by 8 independent Attention transformations

Then we combine back the output

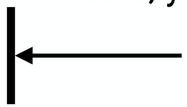
Multi-head attention block: code

edit `transformer_torch.py` (continued from the previous slide):

```
# ... continued from the previous page
# add these methods :
def split_heads(self, x):
    # Reshape the input to have num_heads for multi-head attention
    batch_size, seq_length, d_model = x.shape
    output = x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)
    print ("Tensor shape before splitting heads", x.shape)
    print ("Tensor shape after splitting heads", output.shape)
    return output

def combine_heads(self, x):
    # Combine the multiple heads back to original shape
    batch_size, _, seq_length, d_k = x.shape
    output = x.transpose(1, 2).contiguous()
    output = output.view(batch_size, seq_length, self.d_model)
    print ("Tensor shape before combining heads", x.shape)
    print ("Tensor shape after combining heads", output.shape)
    return output
```



We need these print stamtants in order to understand what does the `split_heads` and `combine_heads` code do. After we do all the tests, you can/should remove or comment those



Multi-head attention block: code (recap and test)

transformer_torch.py:

```
import torch as tr
import math
class MultiHeadAttention(tr.nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        # Linear layers for transforming inputs
        self.W_q = tr.nn.Linear(d_model, d_model) # Query transformation
        self.W_k = tr.nn.Linear(d_model, d_model) # Key transformation
        self.W_v = tr.nn.Linear(d_model, d_model) # Value transformation

        self.d_model = d_model                  # embedding dimension
        self.num_heads = num_heads              # Number of attention heads
        self.d_k = d_model // num_heads         # Dimension of each head's key, query, and value
        self.W_o = tr.nn.Linear(d_model, d_model)# Mixing matrix (we'll understand it later)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # Calculate attention scores
        attn_scores = tr.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        # Apply mask (for example the under-diagonal elements)
        if mask is not None: attn_scores = attn_scores.masked_fill(mask==0,-1e9)

        # Softmax is applied to obtain attention probabilities
        attn_probs = tr.softmax(attn_scores, dim=-1)

        # Multiply by values to obtain the final output
        return tr.matmul(attn_probs, V)

    def forward(self, x, mask=None):
        # Apply linear transformations and split head
        Q = self.W_q(x)
        K = self.W_k(x)
        V = self.W_v(x)
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))


        # Perform scaled dot-product attention and return
        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        attn_output = self.combine_heads(attn_output)    # combine heads
        attn_output = self.W_o(attn_output)              # mix heads
        return attn_output

    def split_heads(self, x):
        # Reshape the input to have num_heads for multi-head attention
        batch_size, seq_length, d_model = x.shape
        output = x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)
        print ("Tensor shape before splitting heads", x.shape)          # remove me later!
        print ("Tensor shape after splitting heads", output.shape)     # remove me later!
        return output

    def combine_heads(self, x):
        # Combine the multiple heads back to original shape
        batch_size, _, seq_length, d_k = x.shape
        output = x.transpose(1, 2).contiguous()
        output = output.view(batch_size, seq_length, self.d_model)
        print ("Tensor shape before combining heads", x.shape)         # remove me later!
        print ("Tensor shape after combining heads", output.shape)     # remove me later!
        return output
```

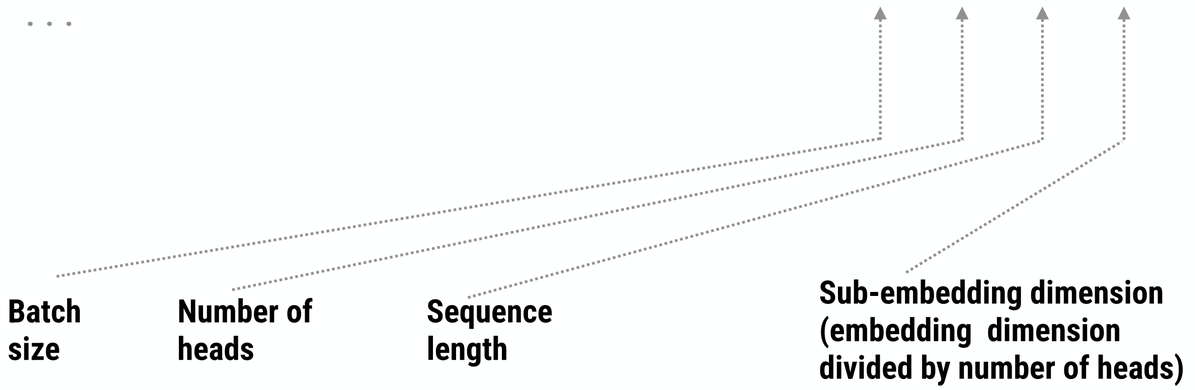
In `test_attention_block_torch.py` replace:

```
...
#from transformer_torch import Attention
from transformer_torch import MultiHeadAttention
...
#attentionblock = Attention(512)
attentionblock = MultiHeadAttention(512, 8)
...
```

Run the test:

```
python test_attention_block_torch.py :
```

```
...
Tensor shape before splitting heads torch.Size([32, 100, 512])
Tensor shape after splitting heads torch.Size([32, 8, 100, 64])
Tensor shape before combining heads torch.Size([32, 8, 100, 64])
...
```



Multi-head attention block: code recap and test

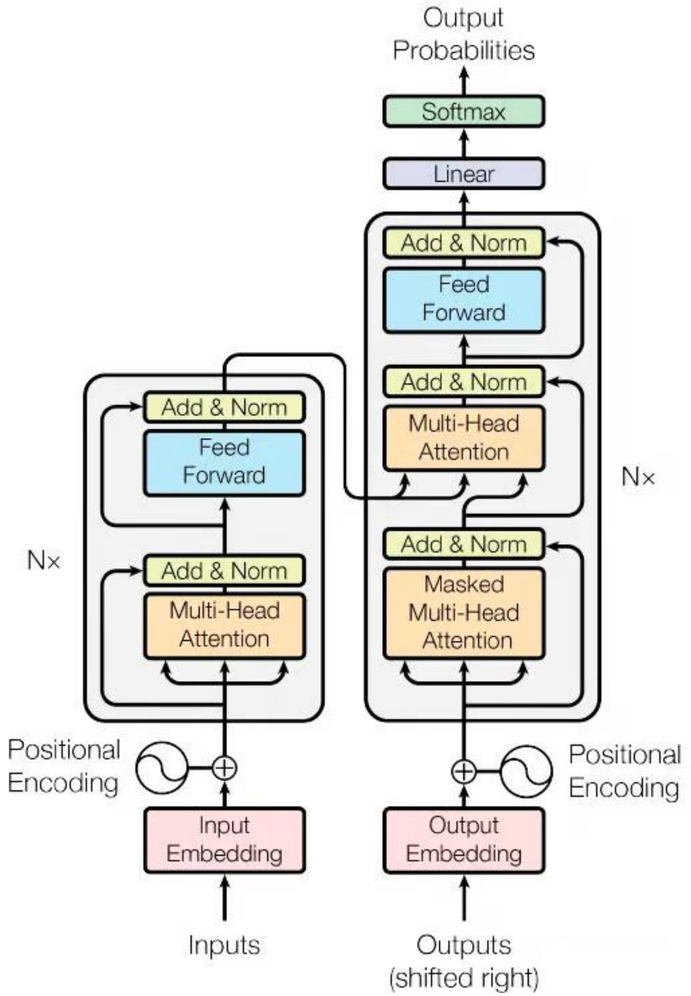
transformer_torch.py:

```
import torch as tr
import math
class MultiHeadAttention(tr.nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        # Linear layers for transforming inputs
        self.W_q = tr.nn.Linear(d_model, d_model) # Query transformation
        self.W_k = tr.nn.Linear(d_model, d_model) # Key transformation
        self.W_v = tr.nn.Linear(d_model, d_model) # Value transformation

        self.d_model = d_model                  # embedding dimension
        self.num_heads = num_heads              # Number of attention heads
        self.d_k = d_model // num_heads          # Dimension of each head's key, query, and value
        self.W_o = tr.nn.Linear(d_model, d_model)# Mixing matrix (we'll understand it later)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # Calculate attention scores
        attn_scores = tr.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        # Apply mask (for example the under-diagonal elements)
        if mask is not None: attn_scores = attn_scores.masked_fill(mask==0,-1e9)

        # Softmax is applied to obtain attention probabilities
        attn_probs = tr.softmax(attn_scores, dim=-1)

        # Multiply by values to obtain the final output
        return tr.matmul(attn_probs, V)

    def forward(self, x, mask=None):
        # Apply linear transformations and split head
        Q = self.W_q(x)
        K = self.W_k(x)
        V = self.W_v(x)
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))


        # Perform scaled dot-product attention and return
        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        attn_output = self.combine_heads(attn_output)    # combine heads
        attn_output = self.W_o(attn_output)              # mix heads
        return attn_output

    def split_heads(self, x):
        # Reshape the input to have num_heads for multi-head attention
        batch_size, seq_length, d_model = x.shape
        output = x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)
        print ("Tensor shape before splitting heads", x.shape)          # remove me later!
        print ("Tensor shape after splitting heads", output.shape)      # remove me later!
        return output

    def combine_heads(self, x):
        # Combine the multiple heads back to original shape
        batch_size, _, seq_length, d_k = x.shape
        output = x.transpose(1, 2).contiguous()
        output = output.view(batch_size, seq_length, self.d_model)
        print ("Tensor shape before combining heads", x.shape)         # remove me later!
        print ("Tensor shape after combining heads", output.shape)     # remove me later!
        return output
```

MultiHeadAttention is a building block of a Transformer model. Now let's build the model itself - which is, roughly speaking, just a sequence of multiple instances of **MultiHeadAttention** stacked one after another (sort of convolutional filters in a CNN)...

Transformer model: architecture

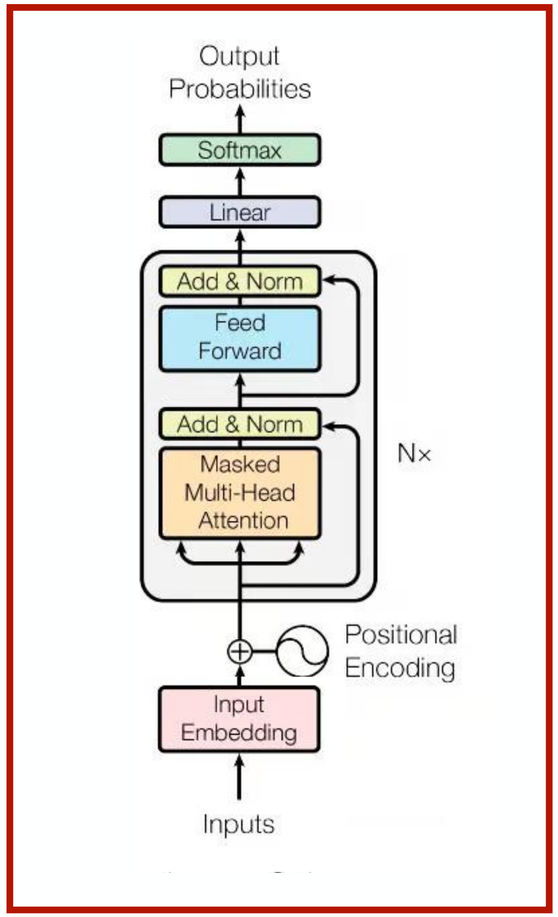
Original cross-attention Transformer from the "Attention is all you need " paper https://arxiv.org/abs/1706.03762



See the full code of cross-attention tutorial in: https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch

Transformer model: architecture

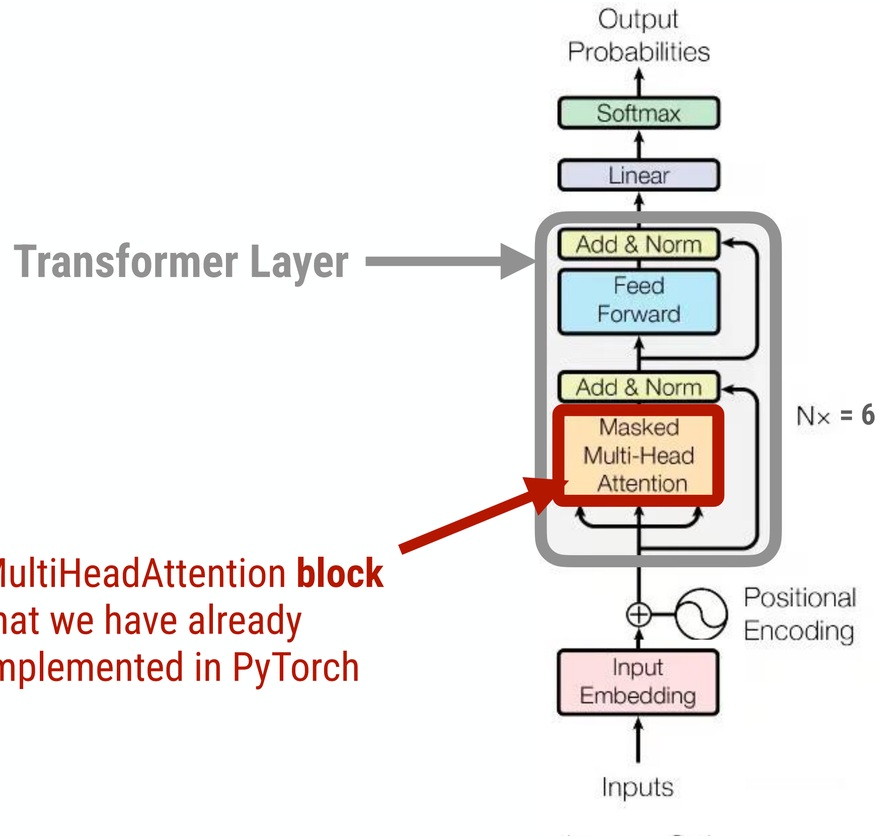
Self-attention-only (decoder-only)
Transformer used in GPT-like models



So-called "Decoder-only" architecture is
Considered in this tutorial

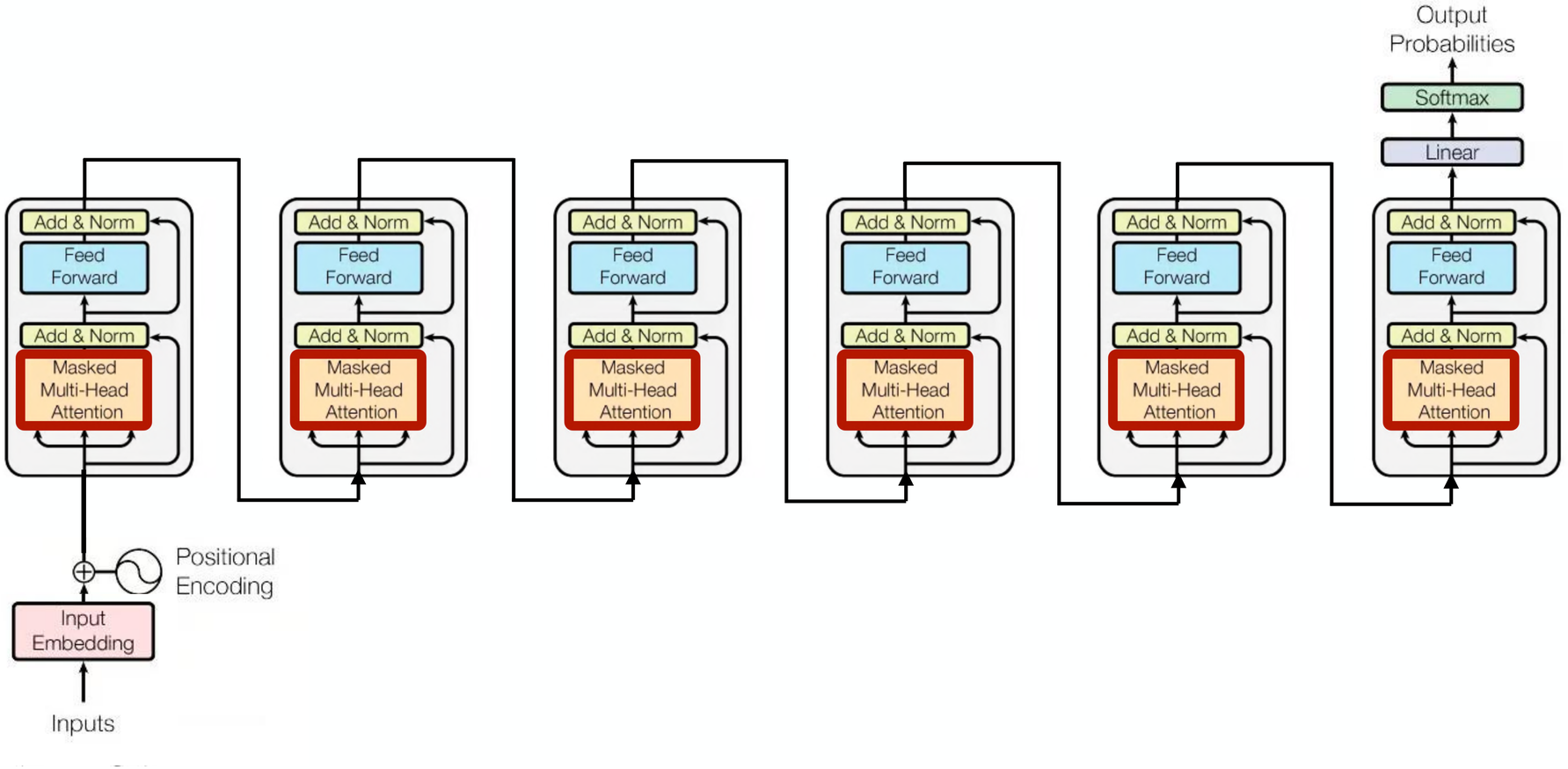
Transformer model: architecture

Self-attention-only (decoder-only)
Transformer used in GPT-like models



Transformer model: architecture

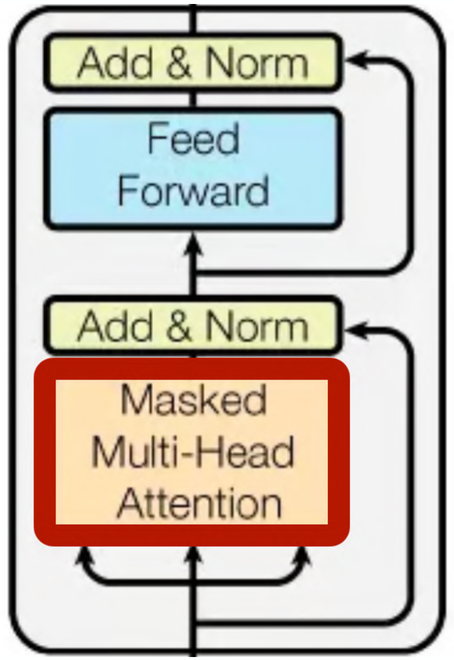
Self-attention-only (decoder-only)
Transformer used in GPT-like models



Transformer layer: code

Edit `transformer_torch.py` (add the following in the end):

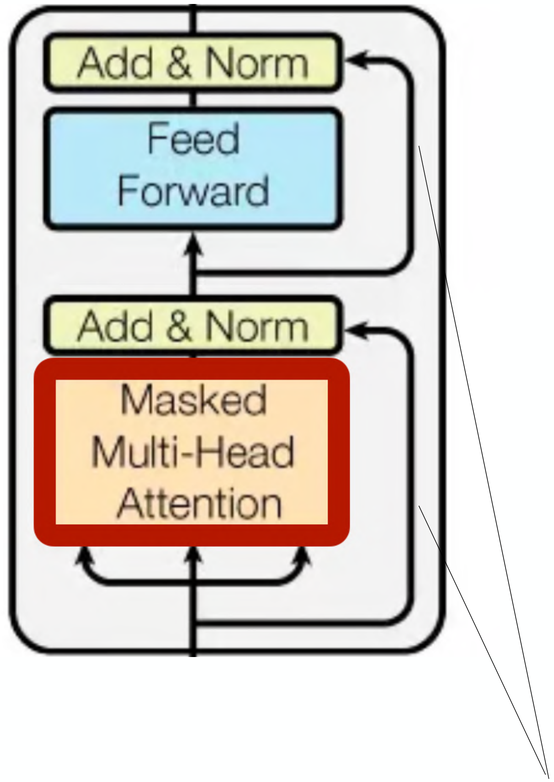
```
class TransformerLayer(tr.nn.Module):  # a.k.a. "Decoder" Layer
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super().__init__()
        # instance of attnetion head
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        # instance of layer normalisation
        self.norm1 = tr.nn.LayerNorm(d_model)
        # layers of feed-forward part
        self.fc1 = tr.nn.Linear(d_model, d_ff)
        self.fc2 = tr.nn.Linear(d_ff, d_model)
        self.relu = tr.nn.ReLU()
        # another layer normalisation and dropout
        self.norm3 = tr.nn.LayerNorm(d_model)
        self.dropout = tr.nn.Dropout(dropout)

    def forward(self, x, mask):
        # attention block
        attn_output = self.self_attn(x, mask)
        # sum attention output and the original input, normalise
        x = self.norm1(x + self.dropout(attn_output))
        # non-linear feed-forward oparation
        ff_output = self.fc2(self.relu(self.fc1(x)))
        # sum the non-linear output and the with its input, normalise
        x = self.norm3(x + self.dropout(ff_output))
        return x
```



Transformer layer: code

Edit `transformer_torch.py` (add the following in the end):

```
class TransformerLayer(tr.nn.Module):  # a.k.a. "Decoder" Layer
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super().__init__()
        # instance of attnetion head
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        # instance of layer normalisation
        self.norm1 = tr.nn.LayerNorm(d_model)
        # layers of feed-forward part
        self.fc1 = tr.nn.Linear(d_model, d_ff)
        self.fc2 = tr.nn.Linear(d_ff, d_model)
        self.relu = tr.nn.ReLU()
        # another layer normalisation and dropout
        self.norm3 = tr.nn.LayerNorm(d_model)
        self.dropout = tr.nn.Dropout(dropout)

    def forward(self, x, mask):
        # attention block
        attn_output = self.self_attn(x, mask)
        # sum attention output and the original input, normalise
        x = self.norm1(x + self.dropout(attn_output))
        # non-linear feed-forward oparation
        ff_output = self.fc2(self.relu(self.fc1(x)))
        # sum the non-linear output and the with its input, normalise
        x = self.norm3(x + self.dropout(ff_output))
        return x
```



Pay attention to flow of non-changed input that is always added to the output (inspired by ResNET convolutional nets...)

Positional Encoding

So far we never mentioned positional encoding... How does the Attention transformation know about relative positions of tokens/ embedding in the sequence? A neat trick to facilitate positional encoding is to **add to the embedding vector another vector that encodes the position (Positional Encoding a.k.a. Positional Embedding)**. This is done before the data is fed to transformer layers.

```
      "caves" embedding →  [-0.34,  1.31, ..., ..., ..., ..., ..., ..., ..., ..., ..., 0.93, -0.57]
encoding for position 3 →  [ 0.14, -0.99, ..., ..., ..., ...,...,  ..., ..., ..., ..., .. 0.0003,  1.00]
                                                              +
```

Position encoding vector:

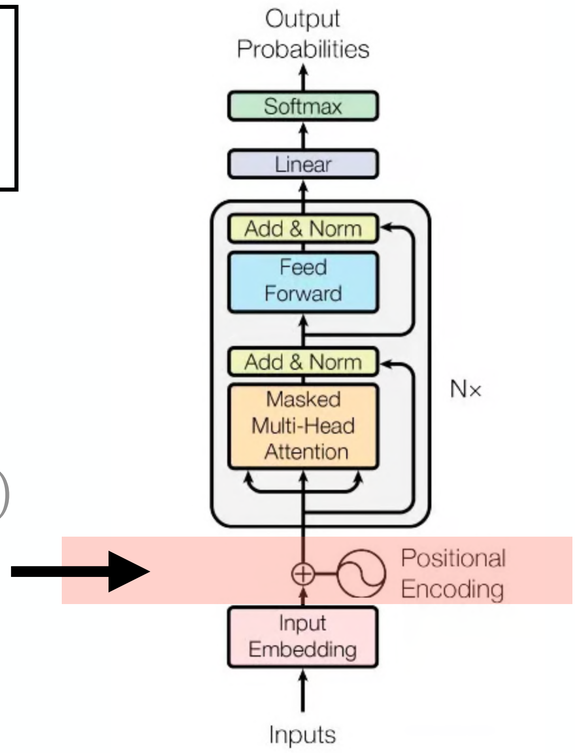
$$\mathrm{PE}(pos, i) = \begin{cases} \mathbf{sin}(pos/1000^{i/d}), \text{ for } i=0,2,... \\ \mathbf{cos}(pos/1000^{(i-1)/d}), \text{ for } i=1,3,.. \end{cases}$$

where

$i=0, .., d\text{-}1$

d - embedding dimension (512 in our example)

$pos = 0, ...,$ max_seqence_length

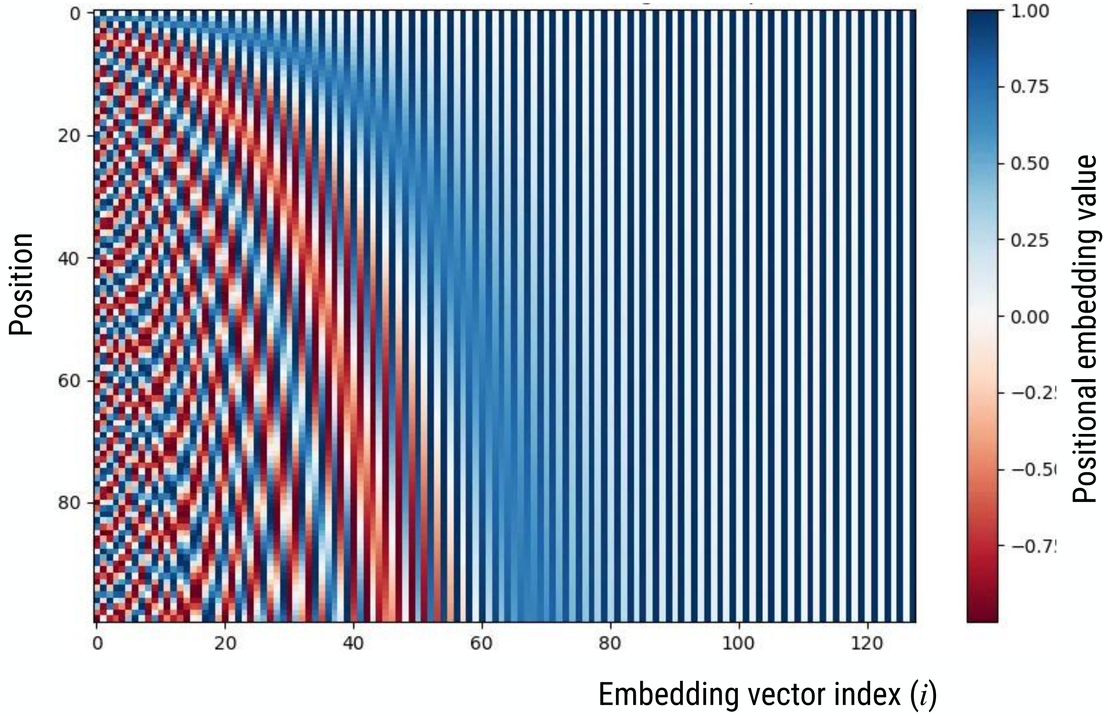


Why 10000? → Wide range of wavelets covering patterns from short to long distances.
For example, for small i the period is close to ~1 - short dependancies, for the largest i period gets very high - close to 10000 (long dependencies)

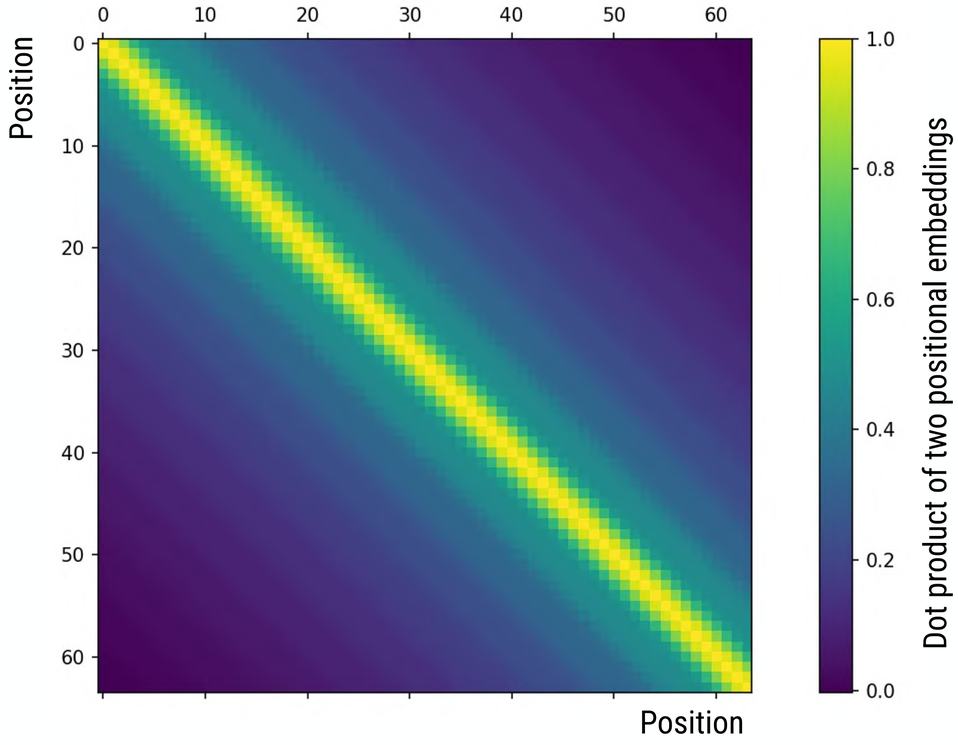
Positional Encoding

Why this specific definition is used?

1. **Unique representation of each position**
2. **Values limited in +-1 range → good for NN**



3. **Encodes relative distance*:**



* Dot product between between two positional vectors is only defined by their relative position difference

See also:
https://medium.com/@hirok4/understanding-transformer-sinusoidal-position-embedding-7cbaaf3b9f6a

Positional Encoding: code implementation

Edit `transformer_torch.py` (add the following in the end):

```
class PositionalEncoding(tr.nn.Module):
    def __init__(self, d_model, max_seq_length):
        super().__init__()

        pe = tr.zeros(max_seq_length, d_model)
        position = tr.arange(0, max_seq_length, dtype=tr.float).unsqueeze(1)
        div_term = tr.exp(tr.arange(0, d_model, 2).float() / d_model * -(math.log(10000.0)))

        pe[:, 0::2] = tr.sin(position * div_term)
        pe[:, 1::2] = tr.cos(position * div_term)

        # register pe as part of the model but do not trat as (trainable) parameters
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:x.size(1)]
```

Let's test it, run **`python`** `-i transformer_torch.py`:

```
>>> p = PositionalEncoding(512, 100) # embedding dimension 512, maximum 100 tokens in a sequence
>>> x = tr.zeros(100, 512)           # 100 embedding-size (512) vectors with with initiated with zeros
>>> position_embeddings = p(x)       # this is a sum of embedding itself + position, however our embeddings are just zeros...
>>> print (position_embeddings[0])
   tensor([0., 1., 0., 1., 0., ... 1., 0., 1.])
>>> print (position_embeddings[1])
   tensor([8.4147e-01, 5.4030e-01,..., 1.0366e-04, 1.0000e+00])
```

Putting Transformer pieces together

Incorporate everything into the final model → edit `transformer_torch.py` (add the following in the end):

```
class Transformer(tr.nn.Module):
    def __init__(self, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout):
        super().__init__()
        self.embedding = tr.nn.Embedding(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length)
        self.layers = tr.nn.ModuleList([TransformerLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)])
        self.fc = tr.nn.Linear(d_model, tgt_vocab_size) # conversion from embedding space to token probabilities
        self.dropout = tr.nn.Dropout(dropout)

    def generate_mask(self, x): # mask 0 (empty) tokens and above-diagonal elements during the attention matrix generation
        mask_nonzero = (x != 0).unsqueeze(1).unsqueeze(3)
        seq_length = x.shape[1]
        mask = (1 - tr.triu(tr.ones(1, seq_length, seq_length), diagonal=1)).bool()
        mask = mask_nonzero & mask
        return mask

    def forward(self, x):
        mask = self.generate_mask(x)
        x_embedded = self.dropout(self.positional_encoding(self.embedding(x)))

        output = x_embedded
        for layer in self.layers:
            output = layer(output, mask)
        return self.fc(output)
```

Training Transformer with mockup data

Now let's test the code with a random data, create `train_transformer_torch.py`:

```
import torch as tr
from transformer_torch import Transformer

tgt_vocab_size = 5000
d_model = 512
num_heads = 8
num_layers = 6
d_ff = 2048
max_seq_length = 100
dropout = 0.1

transformer = Transformer(tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout)

data = tr.randint(1, tgt_vocab_size, (64, max_seq_length))  # radom integers in the range from 1 to tgt_vocab_size (batch_size, seq_length)
x = data[:, :-1]    # remember, we remove last token in the sequence
y = data[:, 1:]     # the target for training is the same as the input but shifted by +1

criterion = tr.nn.CrossEntropyLoss(ignore_index=0)
optimizer = tr.optim.Adam(transformer.parameters())    #, lr=0.0001, betas=(0.9, 0.98), eps=1e-9)

transformer.train()
for epoch in range(10):
    optimizer.zero_grad()
    y_pred = transformer(x)
    loss = criterion(y_pred.contiguous().view(-1, tgt_vocab_size), y.contiguous().view(-1))
    loss.backward()
    optimizer.step()
    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

Run the test: **`python`** `train_transformer_torch.py`:

```
Epoch: 1, Loss: 8.696113586425781
Epoch: 2, Loss: 8.363970756530762
...
```

Training Transformer with real data

- Let's run it with some real text, create `train_transformer_torch_real.py`:

```
import torch as tr
from transformer_torch import Transformer
from transformers import  AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("gpt2")

tgt_vocab_size = tokenizer.vocab_size
d_model = 128 #512  # we use simpler model compared to the original "Attention is all you need" paper
num_heads = 8
num_layers = 2 #6 # same here .. simple model
d_ff = 2048
max_seq_length = 10 #100  # and here...
dropout = 0.1

transformer = Transformer(tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout)
with open('little_prince.txt','r') as f: text = f.read()
data = tokenizer(text, return_tensors="pt").input_ids[0]
n_seq = 32
data = data[:n_seq*max_seq_length] # use a small portion of text, 32*100 tokens
data = data.view(n_seq,max_seq_length)
x = data[:,:-1]
y = data[:,1:]

criterion = tr.nn.CrossEntropyLoss(ignore_index=0)
optimizer = tr.optim.Adam(transformer.parameters())

transformer.train()
for epoch in range(50):
    optimizer.zero_grad()
    y_pred = transformer(x)
    loss = criterion(y_pred.contiguous().view(-1, tgt_vocab_size), y.contiguous().view(-1))
    loss.backward()
    optimizer.step()
    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
    tr.save(transformer.state_dict(), './model.pth')
```

- Create `little_prince.txt` manually by copying the first fragment of text (few pages) from https://archive.org/stream/TheLittlePrince-English/littleprince_djvu.txt
- Run the training `python train_transformer_torch_real.py`:

```
...
Epoch: 50, Loss: 0.7570830583572388
```

Transformer prediction/inference: word-by-word

- Create `test_transformer_torch_real.py`:

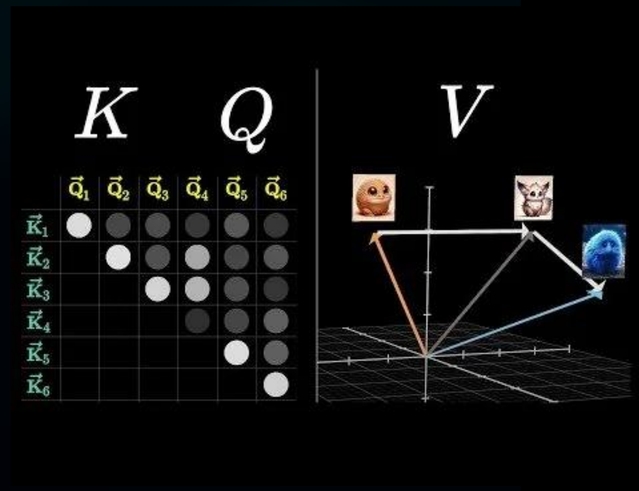
```
# ============== copypasted from the training file before ============
import torch as tr
from transformer_model import Transformer
from transformers import  AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("gpt2")

tgt_vocab_size = tokenizer.vocab_size
d_model = 128
num_heads = 8
num_layers = 2
d_ff = 2048
max_seq_length = 10
dropout = 0.1

transformer = Transformer(tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout)
# ============== copypasted from the training file before ============

transformer.load_state_dict(tr.load('./model.pth'))
transformer.eval()

def predict_next_word(x):
    print ("Input string: ", x)
    data = tokenizer(x, return_tensors="pt").input_ids
    output = transformer(data) # apply our transformer model
    outtokens = tr.argmax(output,dim=-1)
    return tokenizer.batch_decode(outtokens[:,-1:])[0] # decode only last token
```

- Run the code: `python test_transformer_torch_real.py`:

```
>>> x="The Little Prince"
>>>
>>> x+=predict_next_word(x)
Input string:  The Little Prince
>>> x+=predict_next_word(x)
Input string:  The Little Prince appears
>>> x+=predict_next_word(x)
Input string:  The Little Prince appears to
>>> x+=predict_next_word(x)
Input string:  The Little Prince appears to be
>>> x+=predict_next_word(x)
Input string:  The Little Prince appears to be a
...
```

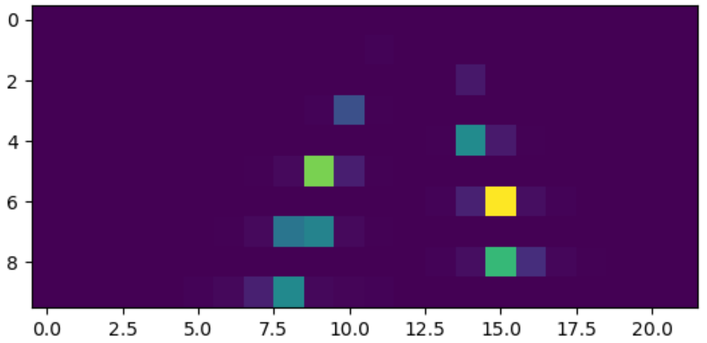
Transformers & Language Models: summary

- **Language Model** ~ a neural network that predicts word/token one at a time given the tokens before
- **Embedding** - key instrument in Language Models:
  ➡ representation of words/tokens with continues vectors in multidimensional space
- **Transformers**- the most powerful/efficient architecture for language models so far
- **Attention transformation**: central component of Transformers
  ➡ alike convolutional filter in CNNs
- Can we use Transformers outside Language Models? - **Of course!**



Step 4: Visualizing the results

prediction of layer 9

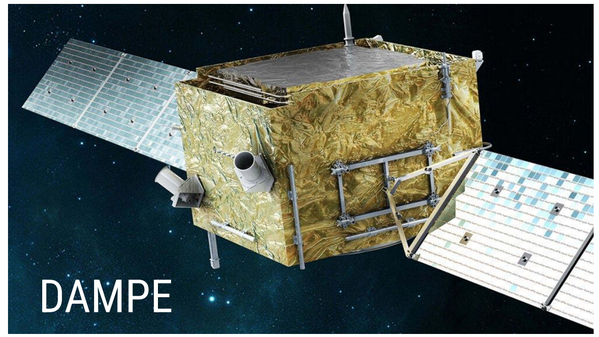


input

AI generated

Final Exercise: Transformer in physics (DAMPE case)

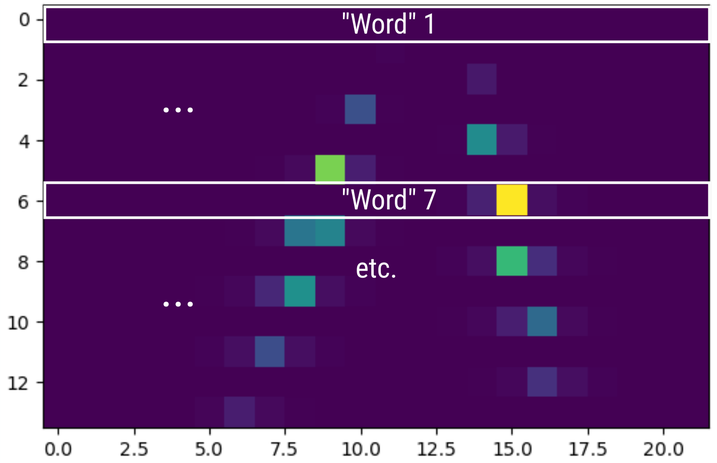
Consider applying our Transformer model to the analysis of DAMPE data. The goal is to train the model to predict the development of particle interaction (shower) in the detector.



➡ Consider DAMPE calorimeter image (14 x 22) as a sequence of (14) "words"

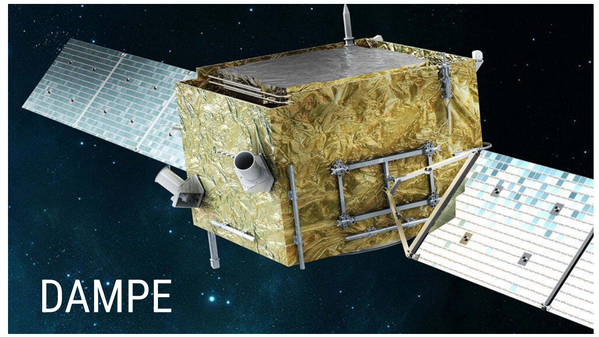
➡ Can we predict next word based on the current and previous ones?

```
>>> from dampe import get_dampe_data, plot_dampe_event
>>> data = get_dampe_data()
>>> plot_dampe_event(data['caloimages'][12])
```



NOTE: for simplicity, we have specifically chosen the task that somewhat resembles the Language Model text sequence. However, Transformers can be equally applied to any other task in physics, such as classification (identifying particle type from the image), regression (predicting particle direction) etc.

Final Exercise: Transformer in physics (DAMPE case)

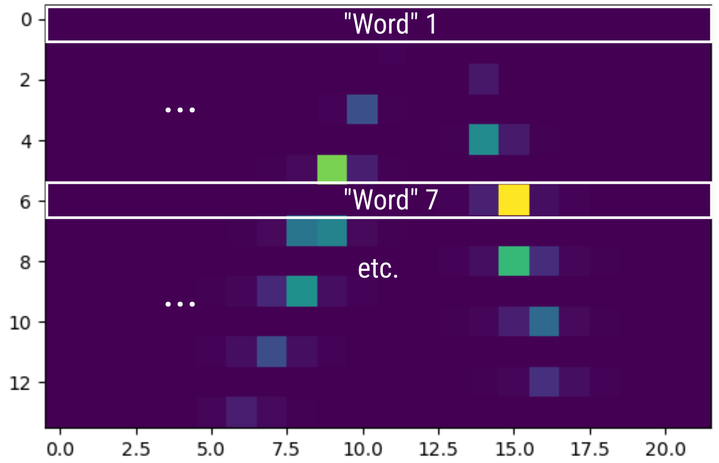


Consider applying our Transformer model to the analysis of DAMPE data. The goal is to train the model to predict the development of particle interaction (shower) in the detector.

➡ Consider DAMPE calorimeter image (14 x 22) as a sequence of (14) "words"

➡ Can we predict next word based on the current and previous ones?

```
>>> from dampe import get_dampe_data, plot_dampe_event
>>> data = get_dampe_data()
>>> plot_dampe_event(data['caloimages'][12])
```



Implementation hints:

- No need for tokenizer
- In the Transformer model use `Linear` layer instead of `Embedding` to convert 22-pixel array ("word") into 512 vector
- Output of the model should be a set of 22 numbers, each representing a signal in a pixel of subsequent layer (instead of probabilities)
- Use `L1Loss` instead of `CrossEntropyLoss`

Step 1: Modify Transformer to DAMPE case

Create a DAMPE transformer file: `cp transformer_torch.py` **`transformer_torch_dampe.py`**, edit it:

```
class Transformer(tr.nn.Module):
    def __init__(self, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout):
```

```
        ....
        #self.embedding = tr.nn.Embedding(tgt_vocab_size, d_model)
        self.embedding = tr.nn.Linear(tgt_vocab_size, d_model)
        ....
```

Replace embedding layer with with a linear transform, `tgt_vocab_size` will 22 in our case (one row of DAMPE image). We still call it "embedding" since the concept is the same -- we map our "word" (array of 22 numbers) into a vector in a higher-dimensional space

```
    def generate_mask(self, x): #
        #mask_nonzero = (x != 0).unsqueeze(1).unsqueeze(3)
        seq_length = x.shape[1]
        mask = (1 - tr.triu(tr.ones(1, seq_length, seq_length), diagonal=1)).bool()
        #mask = mask_nonzero & mask
        return mask
```

Comment our the `mask_nonzero` part of the mask, since we will not have such thing as empty tokens. Also the input data format changed - before we had one number that is a token ID, now we have an array of 22 numbers which represents a "word" (row of DAMPE pixels)

Step 2: Modify torch training code to DAMPE case

Create training code: `cp run_fit_dampe_torch.py run_fit_dampe_torch_transformer.py`, edit it:

```
...
#from model_cnn_dampe_torch import model
from transformer_torch_dampe import Transformer
model = Transformer(tgt_vocab_size=22, d_model=64, num_heads=2, num_layers=2, d_ff=256, max_seq_length=13, dropout=0.)
#model = Transformer(tgt_vocab_size=22, d_model=512, num_heads=8, num_layers=6, d_ff=2048, max_seq_length=13, dropout=0.)
...
```

Initiate the transformer model. We will use simpler version of the model (compared to the original one), having 2 heads instead of 8, 2 layers instead of 6, d_model = 64 instead of 512, and d_ff=256 instead of 2048

```
...
#x, y = data['caloimages'], data['truthdata']
x = data['caloimages'][:,:-1,:]
y = data['caloimages'][:,1:, :]
```

Unlike before, our images are now both input (x) and target (y). The difference is that in the input we remove the last word (last row of image, so now it has 13 rows, not 14). For the output, we shift the rows by +1

```
...
#x = x.squeeze(dim=3).unsqueeze(dim=1)
x = tr.squeeze(x,dim=3)
y = tr.squeeze(y,dim=3)
```

We change the shape of the data to remove the last dimension/mode, so that instead of `[n_samples, n_rows, n_columns, 1]` it becomes `[n_samples, n_rows, n_columns]`
Remember that the last axis was required for CNNs and represented the number of image channels (1 in our DAMPE case, 3 in case of typical RGB color images)

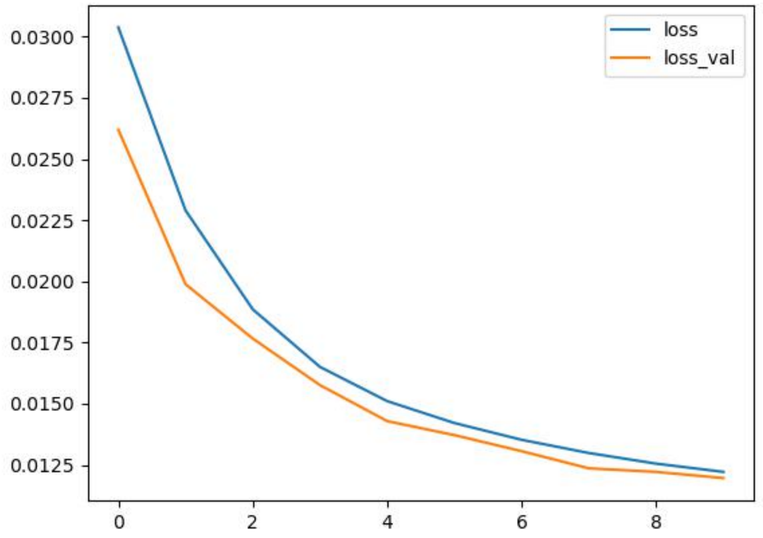
```
...
tr.save(model.state_dict(), './model_transformer_dampe.pth') #'./model_cnn_dampe_paper_torch.pth')
dump(history, open('history_transformer_dampe.p','wb'))      #'history_cnn_dampe_paper_torch.p'
```

Don't forget to rename the model and history to save ...

Step 3: Running torch training code

Set **number of epochs to 10** in `run_fit_dampe_torch_transformer.py` and run the code:

```
pyhon run_fit_dampe_torch_transformer.py
   Processed valid: 99.9 elapsed:28.5s
   loss: 0.03037726884624309 loss_val: 0.026196253926468058
   Processed valid: 99.9 elapsed:28.8s
   loss: 0.022899649906208453 loss_val: 0.019884693373397395
   Processed valid: 99.9 elapsed:28.6s
   loss: 0.01885740954644424 loss_val: 0.017663721486019926
   Processed valid: 99.9 elapsed:28.5s
   loss: 0.016504979629425022 loss_val: 0.015762685200300167
   Processed valid: 99.9 elapsed:28.6s
   loss: 0.015107720140568331 loss_val: 0.014291047355629973
   Processed valid: 99.9 elapsed:28.6s
   loss: 0.01421325063580615 loss_val: 0.013719815335043514
   Processed valid: 99.9 elapsed:28.6s
   loss: 0.013527586701636979 loss_val: 0.013062344417227683
   Processed valid: 99.9 elapsed:28.3s
   loss: 0.012988961886007207 loss_val: 0.01236261014600058
   Processed valid: 99.9 elapsed:28.5s
   loss: 0.012554175169350597 loss_val: 0.012217357914122913
   Processed valid: 99.9 elapsed:28.2s
   loss: 0.012218388652761583 loss_val: 0.011963540029519045
```



Depending on the hardware, it may take 5 to 10 minutes to finish. We have this time to discuss, ask question etc. If it takes significantly longer on your hardware, either use small number of iterations (~2) - it will be enough for illustrative purposes, or just copy the already pre-trained model from the DAMPE package you dowloaded earlier (see `saved_models` folders there)

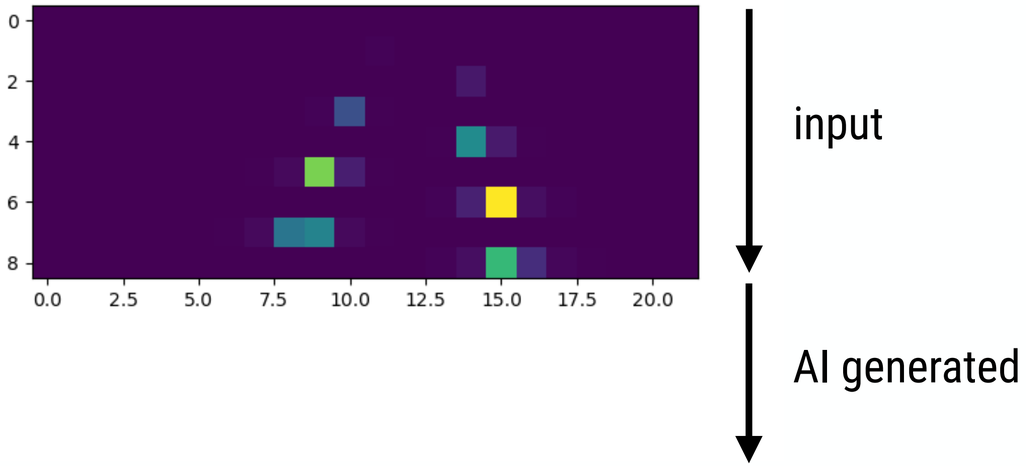
Step 4: Visualizing the results

`cp run_fit_dampe_torch_transformer.py` **`test_dampe_torch_transformer.py`**,
edit : `test_dampe_torch_transformer.py`:

```
#criterion = tr.nn.L1Loss() # <--- remove everything below this line (training loop etc.)
# ... add the following code instead:

from dampe import plot_dampe_event # we will need it for plotting
model.load_state_dict(tr.load("model_transformer_dampe.pth"))
model.eval() # set the model into evaluation mode

def predict_dampe_layers(x, event_id, predict_from_layer):
    in_image = x[event_id:event_id+1,:predict_from_layer] # DAMPE image truncated after [predict_from_layer]
    plot_dampe_event(in_image[0],                         # show image before any prediction
    title=f"input truncated image (no predictions yet")   #
    for i in range(predict_from_layer,14):                # loop over remaining layers
        out_image = model(in_image)                       # predicted image shifted by +1
        out_last_predicted_layer = out_image[:,-1:,:]     # take the last layer of the predicted image and ...
        in_image = tr.cat(                                # ... add it to the input image
            (in_image, out_last_predicted_layer),dim=1)   #
        plot_dampe_event(in_image.detach().numpy()[0],    # show image after one prediction step
        title=f"prediction of layer {i}")
    return in_image.detach().numpy()[0]

# plotting code
event_id = 12                   # let's pick some event from the data
predict_from_layer = 9          # let's feed the model an image of 9 layers and see if it can predict the remaining 5
ai_generated_image = predict_dampe_layers(x, event_id, predict_from_layer)
plot_dampe_event(data['caloimages'][event_id],title="that is the original image")
plot_dampe_event(ai_generated_image,title=f"that image is generated starting from layer {predict_from_layer}")
```

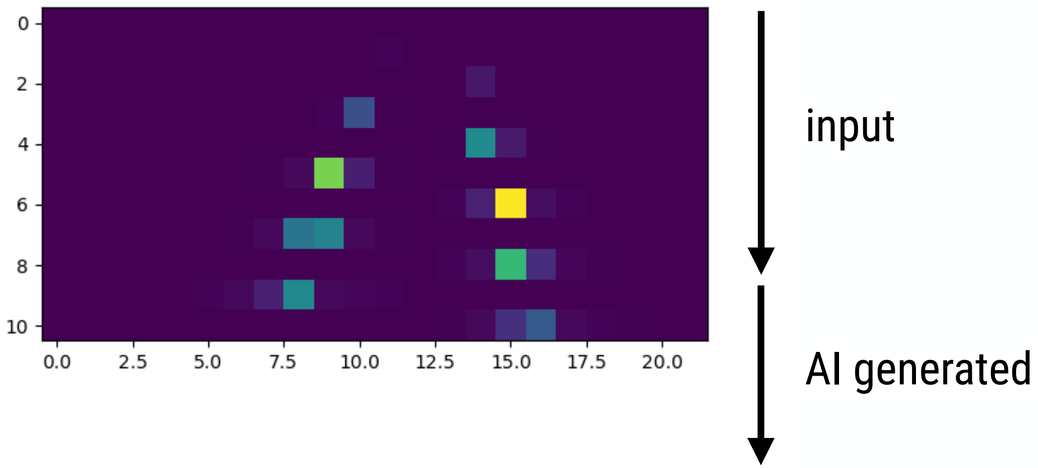
Step 4: Visualizing the results

input truncated image (no predictions so far)



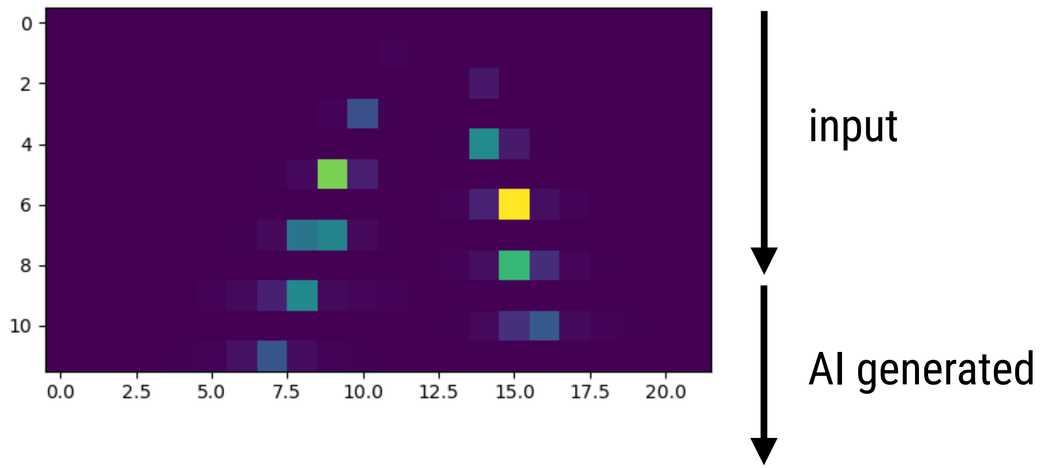
Step 4: Visualizing the results

prediction of layer 10



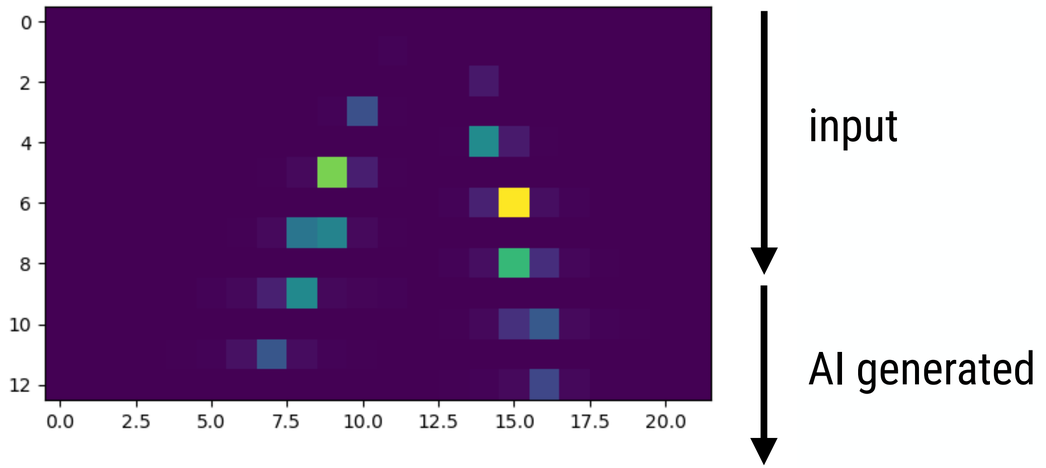
Step 4: Visualizing the results

prediction of layer 11



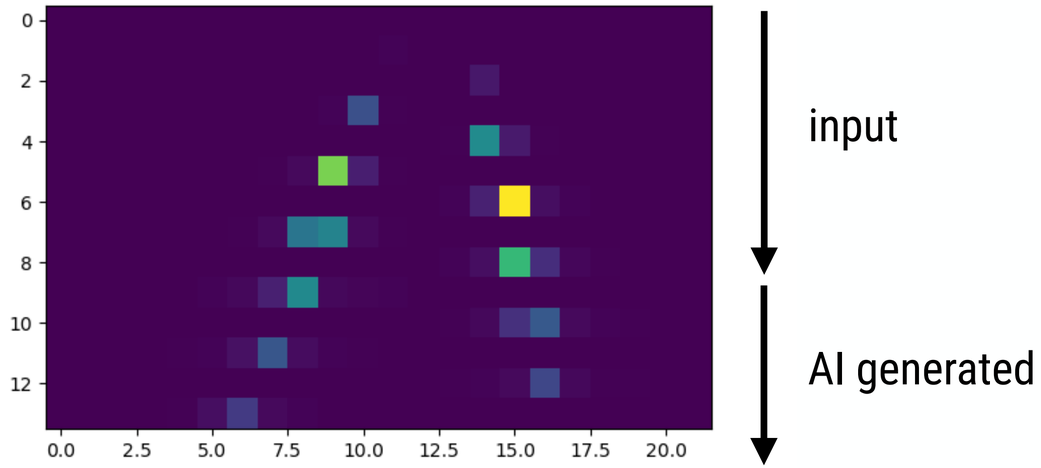
Step 4: Visualizing the results

prediction of layer 12

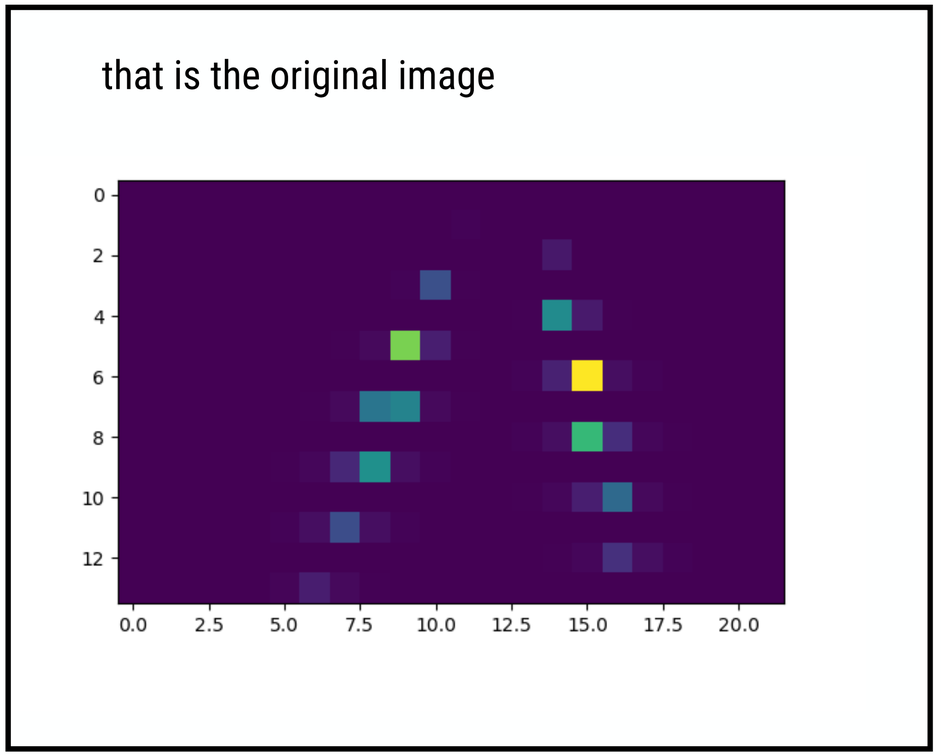


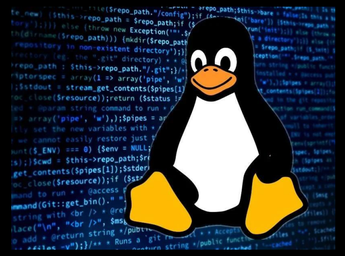
Step 4: Visualizing the results

prediction of layer 13



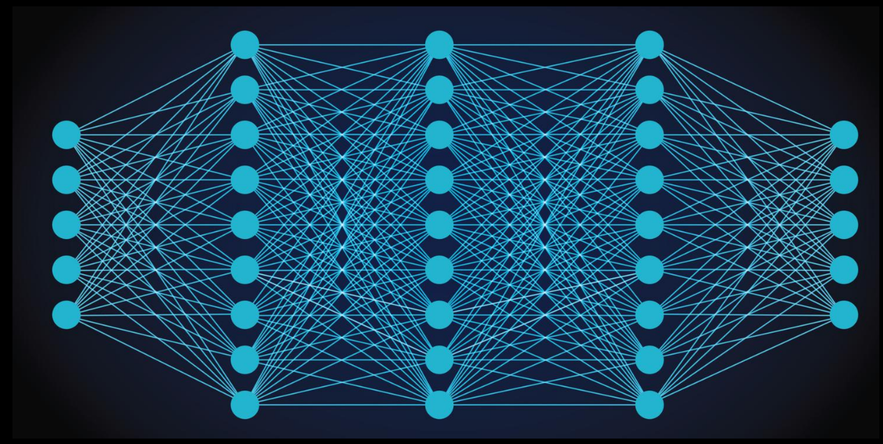
Step 4: Visualizing the results





python™

ANACONDA®



TensorFlow

PyTorch

matplotlib