Machine Learning at CoDaS-HEP 2024, Lesson 4: Survey of Architectures

In lesson 1, I introduced neural networks and the universal function approximation theorem. A single hidden layer implements *adaptive* basis functions, more flexible than classic Taylor and Fourier series.

In lesson 2, we talked about issues involed in any fitting procedure, whether multilayered or not (i.e. a pure linear fit).

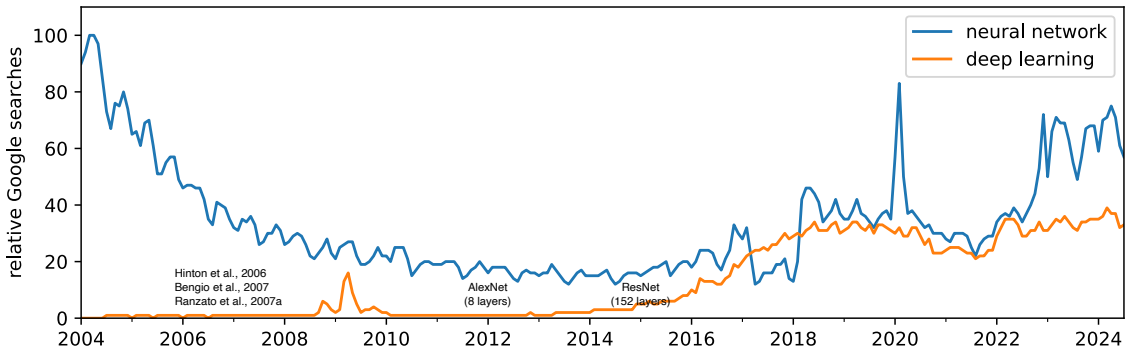
Lesson 3 was an open-ended project to build your own neural network.

In lesson 4, we will consider a variety of neural network "architectures": ways of building networks to improve learning for different types of problems.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import h5py
import awkward as ak

import sklearn.datasets
import torch
from torch import nn
from torch import optim
```

Why should learning be "deep"?

Deep learning: a neural network with 3 or more layers (which is common nowadays).



- 2006–2007: problems that *prevented* the training of deep learning were solved.
- 2012: AlexNet, a GPU-enabled 8 layer network (with ReLU), won the ImageNet competition.
- 2015: ResNet, a GPU-enabled 152+ layer network (with skip-connections), won the ImageNet competition.

By 2015, it was clear that networks with many layers have more potential than one big hidden layer.

Why does it work?

One big hidden layer can approximate any shape, by optimizing adaptive basis functions, but according to conventional wisdom,

> Shallow networks are very good at memorization, but not so good at generalization.

That is, they have a tendency to overfit.

Why are multiple layers better?

Reminder of adaptive basis function:

$$\psi(x; a, b) = \begin{cases} a + bx & \text{if } x > -a/b \\ 0 & \text{otherwise} \end{cases}$$

Function approximation with one hidden layer:

$$f_j(x) = \sum_i^{N_1} \psi(x; a_{ij}, b_{ij}) c_{ij}$$

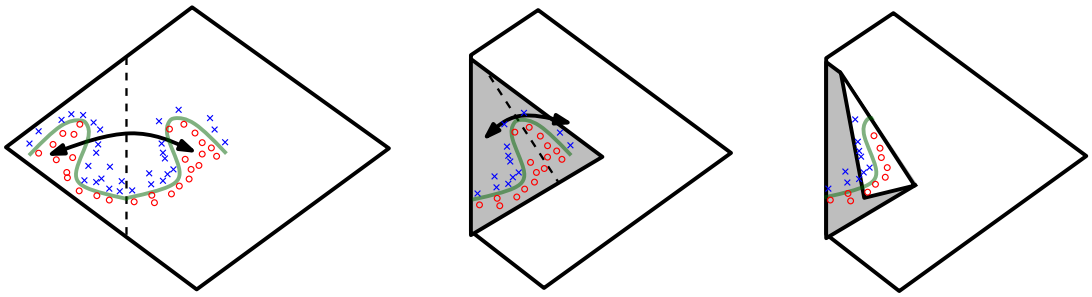
Function approximation with two hidden layers:

$$f_k(x) = \sum_j^{N_2} \psi\left(x; \left[\sum_i^{N_1} \psi(x; a_{i1}, b_{i1}) c_{i1}\right], \left[\sum_i^{N_1} \psi(x; a_{i2}, b_{i2}) c_{i2}\right]\right) \left[\sum_i^{N_1} \psi(x; a_{i3}, b_{i3}) c_{i3}\right.$$

And so on: adaptively adaptive basis functions, then adaptively adaptively adaptive basis functions...

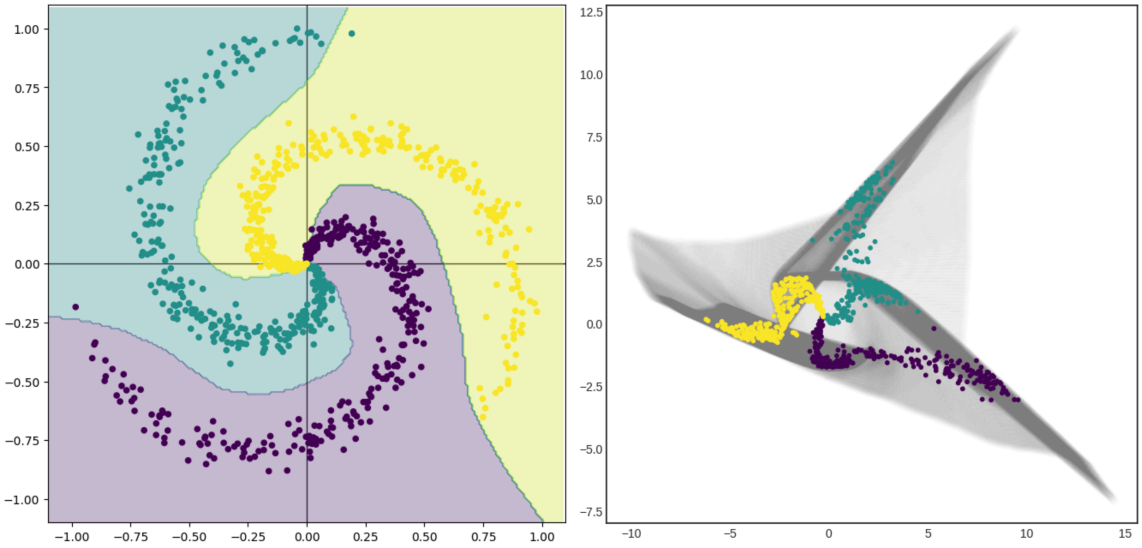
- Adding one more neuron in a single layer adds a wiggle to the fit function.
- Adding one more layer effectively folds the space under the next set of wiggly functions. Instead of fitting individual wiggles, they find symmetries in the data that (probably) correspond to an underlying relationship, rather than noise.

Consider this horseshoe-shaped decision boundary: with two well-chosen folds along the symmetries, it reduces to a simpler curve to fit. Instead of 4 ad-hoc wiggles, it's 2 folds and 1 wiggle.



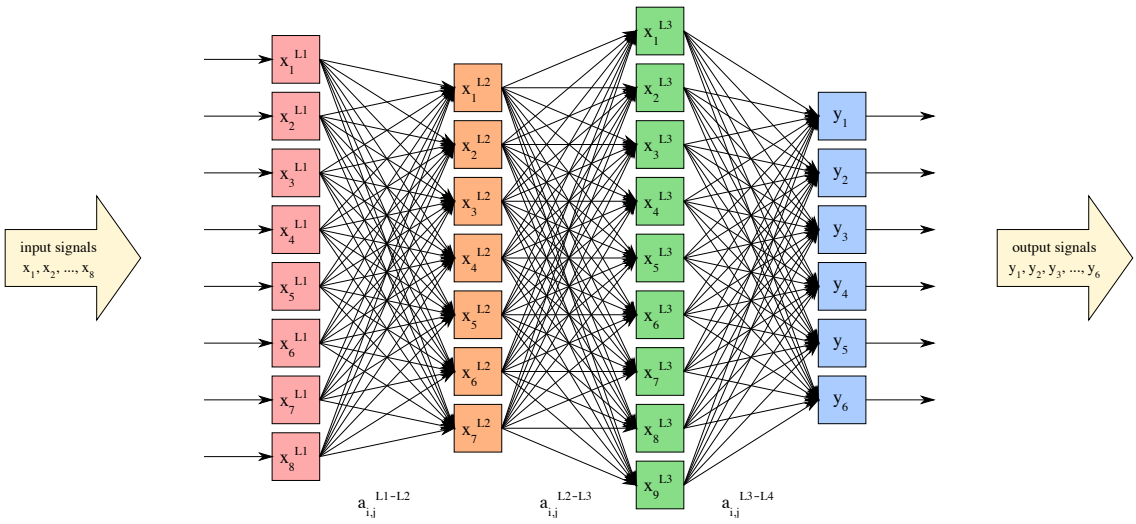
Montúfar, Pascanu, Cho, & Bengio, *On the Number of Linear Regions of Deep Neural Networks* (2014).

Roy Keyes's fantastic demo (with code):



A uniform grid on the feature space (left; grid not shown) projected through the first layer's transformation shows what the underlying space looks like (right; grid is gray) before the second layer makes a linear decision boundary.

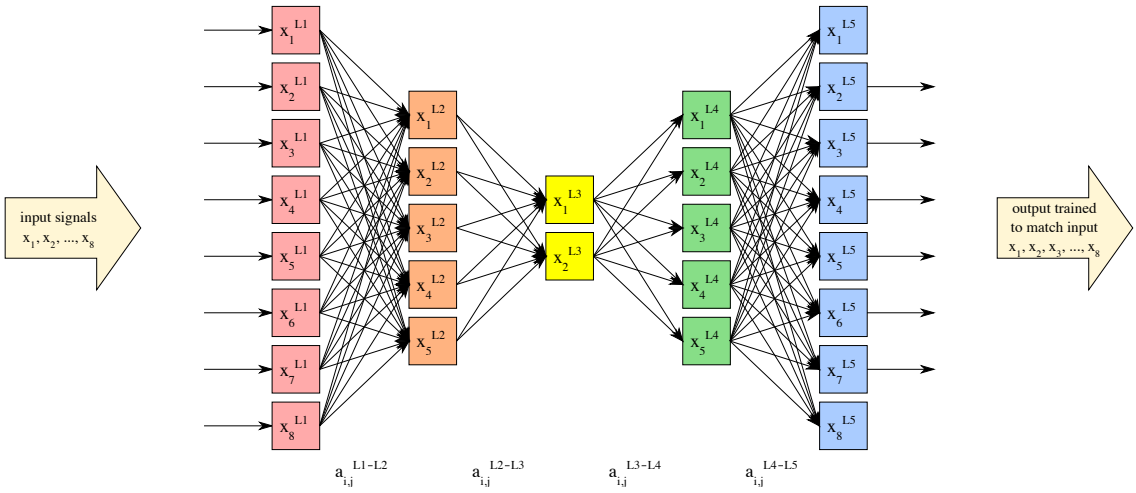
This is our first architecture, just a feed-forward "Multi Layer Perceptron" (MLP):



Neurons in a layer add wiggles to the fitted function; layers add reflections and symmetries that are (probably) real structure.

Autoencoders

Here is our second architecture, an **autoencoder**:



The network structure is qualitatively like the first; the only difference is that the number of neurons shrinks to a "pinch point" and then returns to the original size.

The training is different: instead of trying to fit to known targets, we train the model to produce output that matches the input. When fully trained, it approximates the identity function.

This is **unsupervised learning**. Unlike **supervised learning**, in which we want the model to produce an expected answer, we let this model examine the data and come up with something on its own.

Before talking about it in detail, let's run an autoencoder on the same jet data that you classified with supervised learning.

In [2]:
```
hls4ml_lhc_jets_hlf = sklearn.datasets.fetch_openml("hls4ml_lhc_jets_hlf")

features_unnormalized = torch.tensor(hls4ml_lhc_jets_hlf["data"].values).flo

features = (features_unnormalized - features_unnormalized.mean(axis=0)) / fe
```

In [24]:
```
class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.shrinking = nn.Sequential(
            nn.Linear(16, 12),
            nn.Sigmoid(),
```

```
            nn.Linear(12, 8),
            nn.Sigmoid(),
            nn.Linear(8, 4),
            nn.Sigmoid(),
            nn.Linear(4, 2),
            nn.Sigmoid(),
        )
        self.growing = nn.Sequential(
            nn.Linear(2, 4),
            nn.Sigmoid(),
            nn.Linear(4, 8),
            nn.Sigmoid(),
            nn.Linear(8, 12),
            nn.Sigmoid(),
            nn.Linear(12, 16),
        )

    def forward(self, features):
        return self.growing(self.shrinking(features))

model = Autoencoder()
```

In [25]:
```
NUM_EPOCHS = 100
BATCH_SIZE = 1000

loss_function = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=0.03)

loss_vs_epoch = []
for epoch in range(NUM_EPOCHS):
    total_loss = 0

    for start_batch in range(0, len(features), BATCH_SIZE):
        stop_batch = start_batch + BATCH_SIZE

        optimizer.zero_grad()

        predictions = model(features[start_batch:stop_batch])
        loss = loss_function(predictions, features[start_batch:stop_batch])
        total_loss += loss.item()

        loss.backward()
        optimizer.step()

    loss_vs_epoch.append(total_loss)
    print(f"{epoch = } {total_loss = }")
```

```
epoch = 0 total_loss = 340.67363464832306
epoch = 1 total_loss = 195.9505846053362
epoch = 2 total_loss = 181.89459784328938
epoch = 3 total_loss = 175.5614755153656
epoch = 4 total_loss = 169.6572003364563
epoch = 5 total_loss = 165.32354862987995
epoch = 6 total_loss = 163.7754179239273
epoch = 7 total_loss = 160.22878435254097
epoch = 8 total_loss = 177.49626170098782
epoch = 9 total_loss = 172.1844368427992
epoch = 10 total_loss = 165.95759485661983
epoch = 11 total_loss = 164.50746181607246
epoch = 12 total_loss = 163.3015213906765
epoch = 13 total_loss = 160.0451771467924
epoch = 14 total_loss = 159.1603980064392
epoch = 15 total_loss = 155.93816044926643
epoch = 16 total_loss = 150.65988148748875
epoch = 17 total_loss = 140.77408468723297
epoch = 18 total_loss = 138.31560385227203
epoch = 19 total_loss = 124.40992127358913
epoch = 20 total_loss = 115.33388155698776
epoch = 21 total_loss = 105.11989720910788
epoch = 22 total_loss = 95.68772979825735
epoch = 23 total_loss = 91.86825350672007
epoch = 24 total_loss = 89.12317411601543
epoch = 25 total_loss = 87.27332054078579
epoch = 26 total_loss = 86.05342517793179
epoch = 27 total_loss = 85.01312731951475
epoch = 28 total_loss = 84.32397194206715
epoch = 29 total_loss = 83.27330777049065
epoch = 30 total_loss = 82.57011391967535
epoch = 31 total_loss = 80.9153439104557
epoch = 32 total_loss = 79.79540333151817
epoch = 33 total_loss = 78.1874712780118
epoch = 34 total_loss = 77.59924752265215
epoch = 35 total_loss = 76.7325478643179
epoch = 36 total_loss = 75.75507731735706
epoch = 37 total_loss = 74.79725742340088
epoch = 38 total_loss = 73.85095381736755
epoch = 39 total_loss = 73.08706083893776
epoch = 40 total_loss = 72.65483015030622
epoch = 41 total_loss = 72.29192493855953
epoch = 42 total_loss = 71.62889560312033
epoch = 43 total_loss = 71.24272540956736
epoch = 44 total_loss = 70.82846567034721
epoch = 45 total_loss = 70.57312244176865
epoch = 46 total_loss = 70.85279311984777
epoch = 47 total_loss = 70.49931671470404
epoch = 48 total_loss = 70.27760070562363
epoch = 49 total_loss = 70.09992001205683
epoch = 50 total_loss = 69.7387509867549
epoch = 51 total_loss = 69.41295458376408
epoch = 52 total_loss = 70.03160474449396
epoch = 53 total_loss = 70.36876714229584
epoch = 54 total_loss = 69.99319179356098
epoch = 55 total_loss = 69.35819844901562
```

```
epoch = 56 total_loss = 69.3108624741435
epoch = 57 total_loss = 68.47860571742058
epoch = 58 total_loss = 69.18251391500235
epoch = 59 total_loss = 69.51537895202637
epoch = 60 total_loss = 69.61244164407253
epoch = 61 total_loss = 69.95644647628069
epoch = 62 total_loss = 68.99934091418982
epoch = 63 total_loss = 68.94473230838776
epoch = 64 total_loss = 68.27670296281576
epoch = 65 total_loss = 68.41960521787405
epoch = 66 total_loss = 67.92770597338676
epoch = 67 total_loss = 67.67766281217337
epoch = 68 total_loss = 67.46218267828226
epoch = 69 total_loss = 67.73006649315357
epoch = 70 total_loss = 67.61123272776604
epoch = 71 total_loss = 67.448685772717
epoch = 72 total_loss = 69.17987044900656
epoch = 73 total_loss = 69.90692564845085
epoch = 74 total_loss = 70.24353022128344
epoch = 75 total_loss = 68.00135891884565
epoch = 76 total_loss = 67.27065566927195
epoch = 77 total_loss = 67.2871702387929
epoch = 78 total_loss = 67.41475253552198
epoch = 79 total_loss = 68.34375190734863
epoch = 80 total_loss = 67.4028872475028
epoch = 81 total_loss = 69.12362632155418
epoch = 82 total_loss = 69.39766921103
epoch = 83 total_loss = 68.42363093793392
epoch = 84 total_loss = 67.88294392079115
epoch = 85 total_loss = 67.80852922052145
epoch = 86 total_loss = 66.63873319327831
epoch = 87 total_loss = 66.99424140155315
epoch = 88 total_loss = 66.52692829817533
epoch = 89 total_loss = 66.73177321255207
epoch = 90 total_loss = 65.98841262608767
epoch = 91 total_loss = 66.46983990073204
epoch = 92 total_loss = 66.13528480380774
epoch = 93 total_loss = 66.06313636153936
epoch = 94 total_loss = 66.54619736224413
epoch = 95 total_loss = 67.05613427609205
epoch = 96 total_loss = 67.02046509087086
epoch = 97 total_loss = 66.46214816719294
epoch = 98 total_loss = 66.2453470826149
epoch = 99 total_loss = 66.66824232786894
```

To reproduce the 16-dimensional input data using only 2 dimensions in the middle, the model has to encode it with as little redundancy as possible.

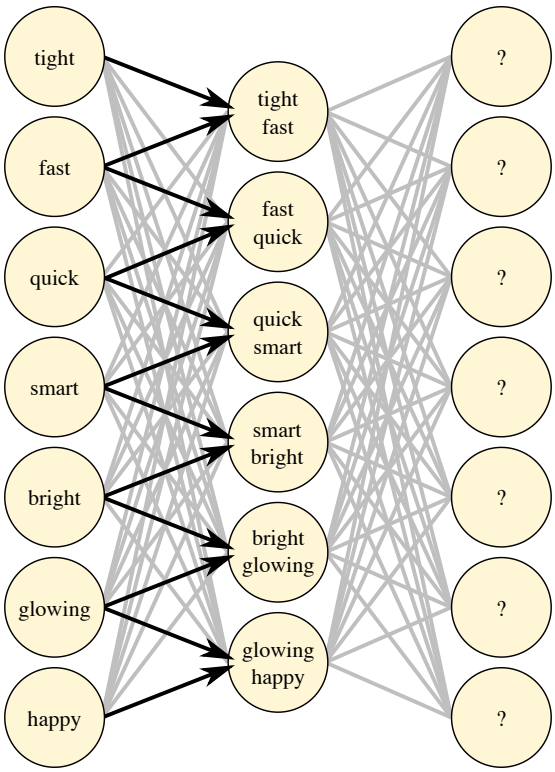
We are asking the model to perform lossy compression—to *approximate* the 16-dimensional data in 2 dimensions.

The data in 2 dimensional space looks very different from how it looks in the original 16 dimensions, but the biggest distinctions in one space are big distinctions in the other. This is an **embedding space** for the data.

Aside: in text processing networks, the physical space consists of exact words and the embedding space consists of *meanings*, which might not be one-to-one with words. (In this embedding space, relationships like

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

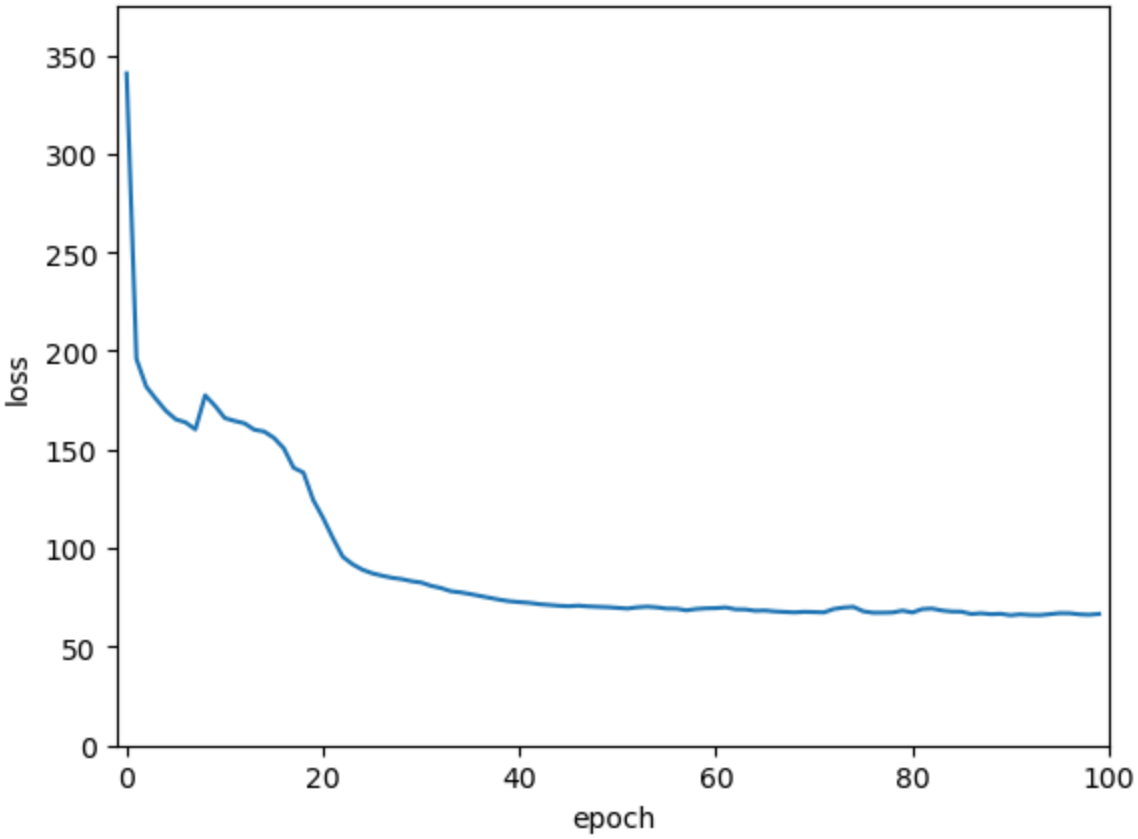
hold.)



In [26]:
```
fig, ax = plt.subplots()

ax.plot(range(len(loss_vs_epoch)), loss_vs_epoch)
ax.set_xlim(-1, len(loss_vs_epoch))
ax.set_ylim(0, 1.1*max(loss_vs_epoch))
ax.set_xlabel("epoch")
```

```
ax.set_ylabel("loss")
```

```
None
```

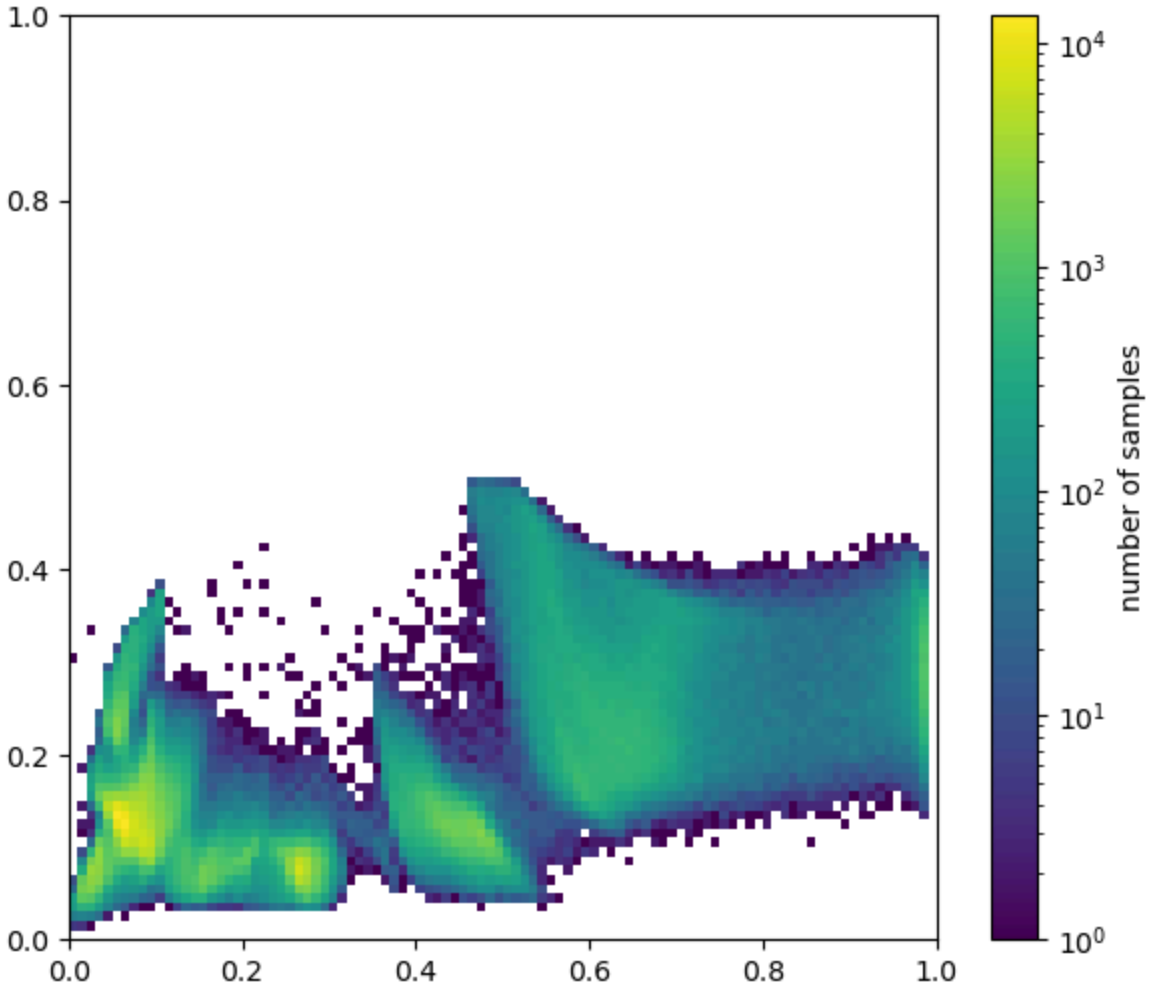


What does the data look like in the 2 dimensional space at the "pinch point" of the network?

In [27]:
```
embedded = model.shrinking(features).detach().numpy()
```

In [28]:
```
fig, ax = plt.subplots(figsize=(7, 6))

p = ax.hist2d(embedded[:, 0], embedded[:, 1], bins=(100, 100), range=((0, 1)
fig.colorbar(p[-1], ax=ax, label="number of samples")
ax.axis([0, 1, 0, 1])
```

```
None
```



The model found some clumps; some clusters of different-looking jets.

Do these correspond to the `'g'`, `'q'`, `'t'`, `'w'`, `'z'` categories, the physically different hadronization mechanisms?

In [29]:
```
targets = torch.tensor(hls4ml_lhc_jets_hlf["target"].cat.codes.values).long(
```

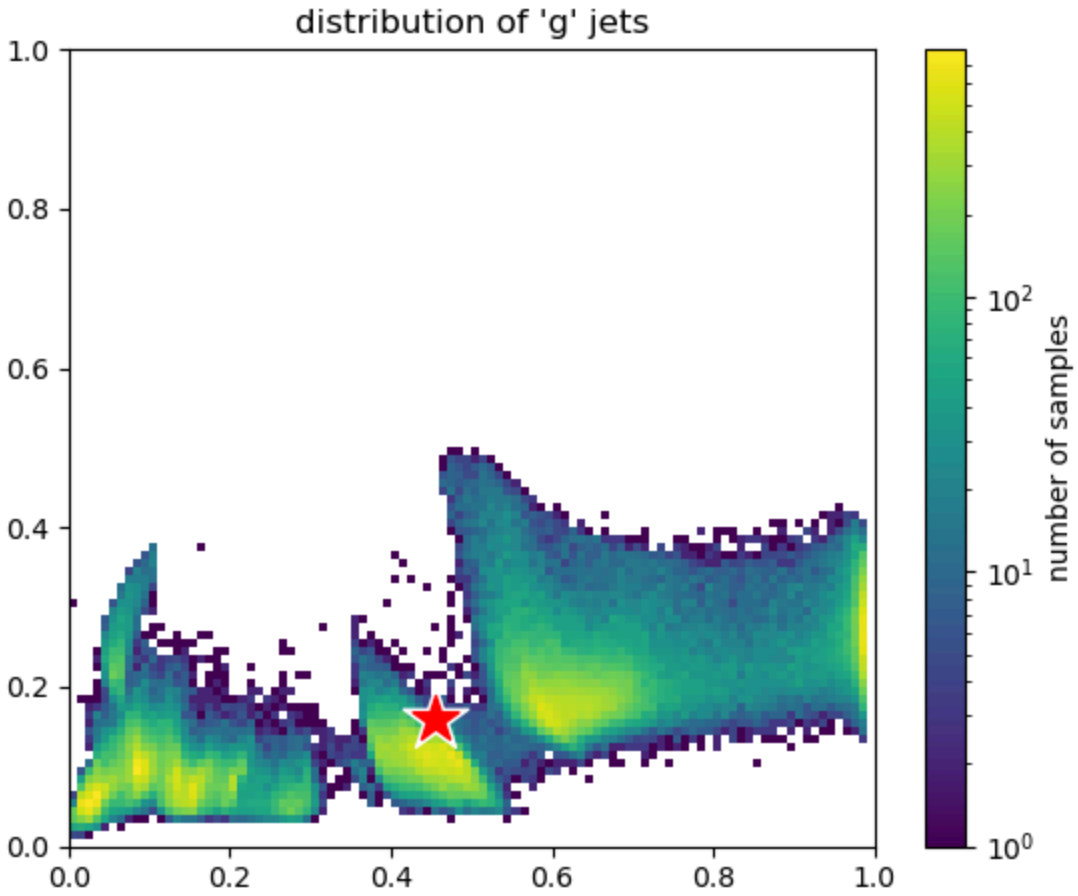
In [30]:
```
hls4ml_lhc_jets_hlf["target"].cat.categories
```

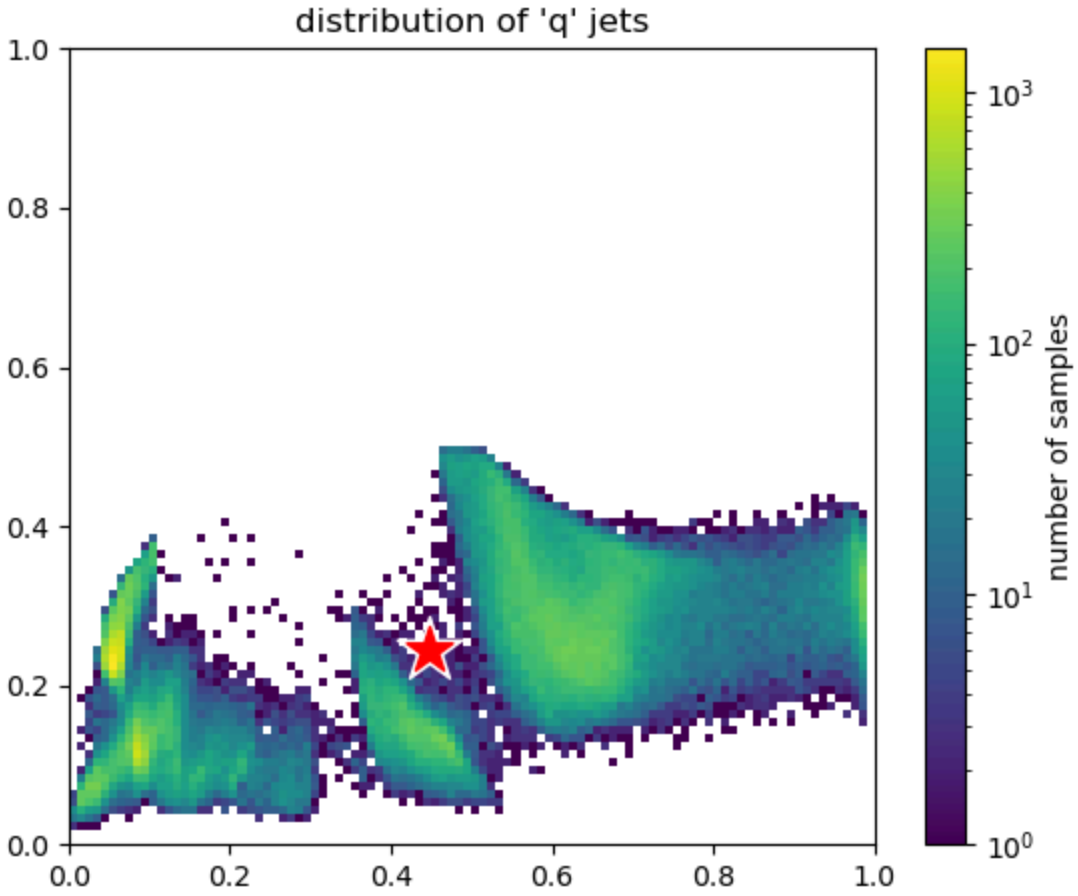
Out[30]:
```
Index(['g', 'q', 't', 'w', 'z'], dtype='object')
```

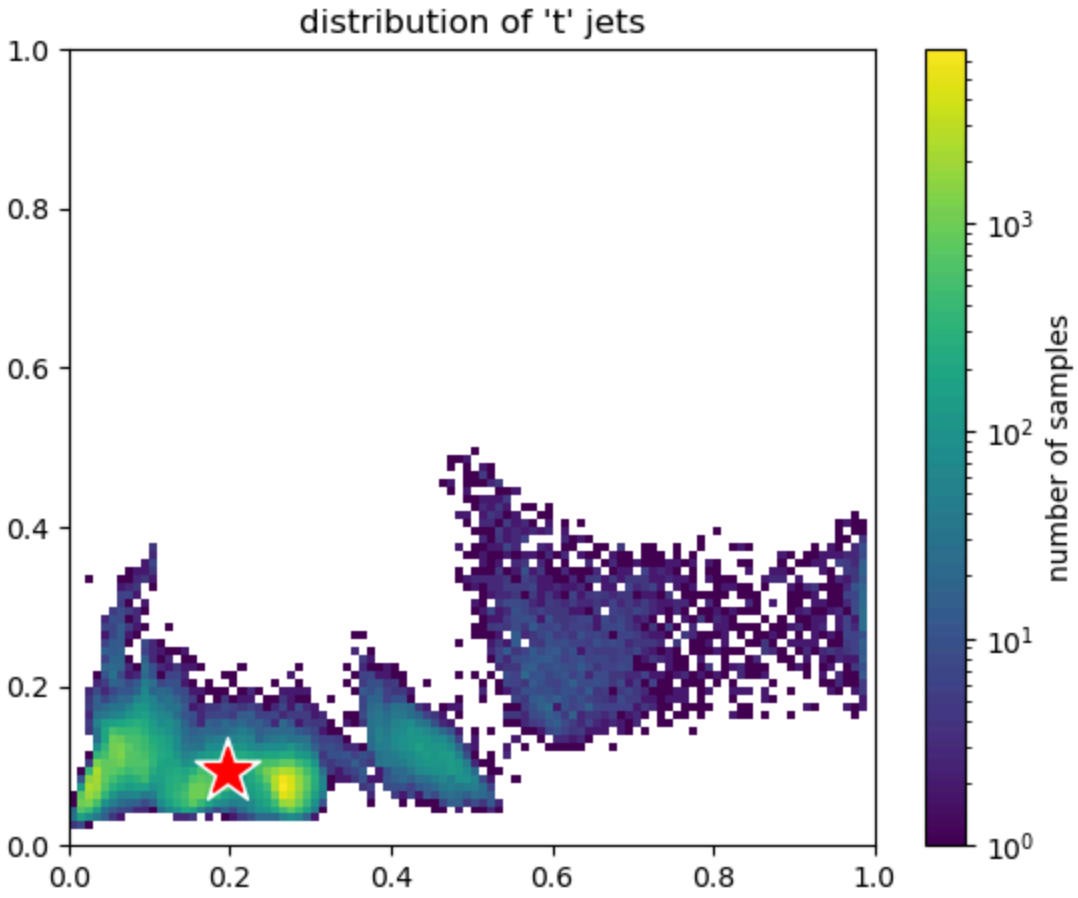
In [31]:
```
embedded_g = model.shrinking(features[targets == 0]).detach().numpy()
embedded_q = model.shrinking(features[targets == 1]).detach().numpy()
embedded_t = model.shrinking(features[targets == 2]).detach().numpy()
```

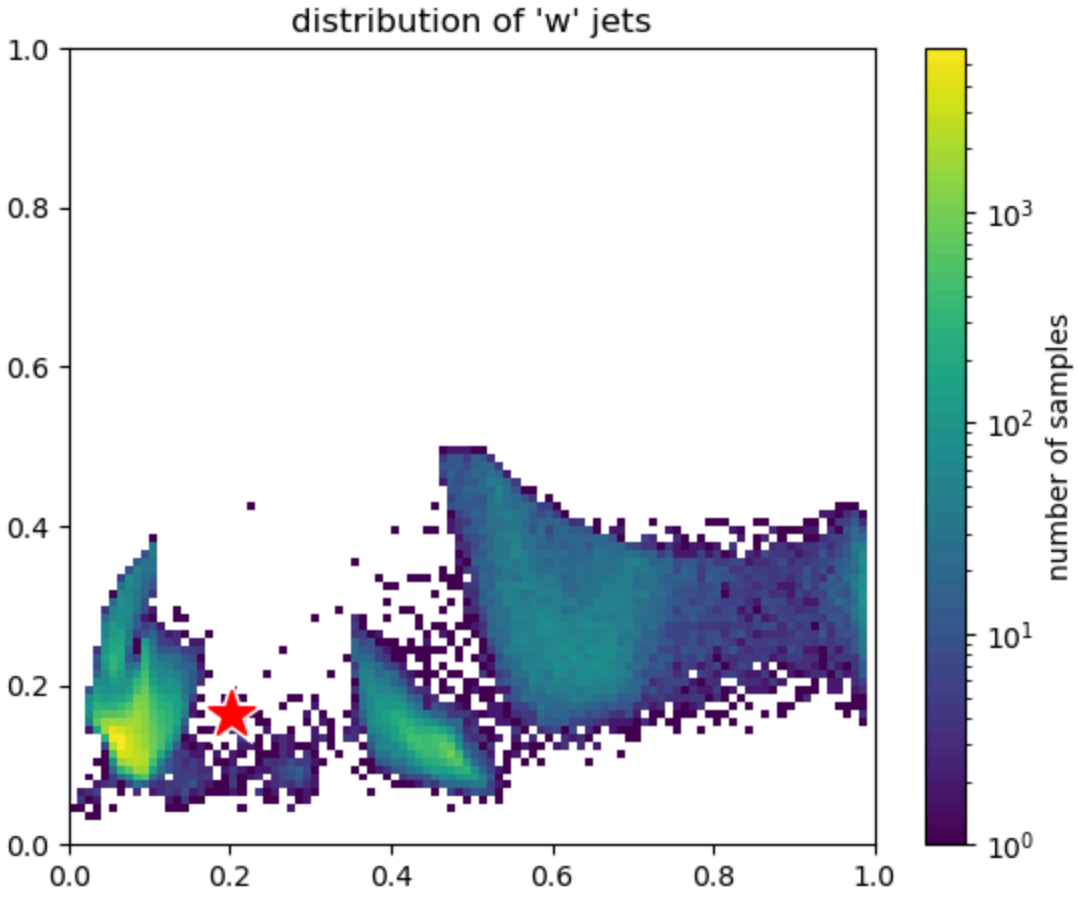
```
embedded_w = model.shrinking(features[targets == 3]).detach().numpy()
embedded_z = model.shrinking(features[targets == 4]).detach().numpy()
```

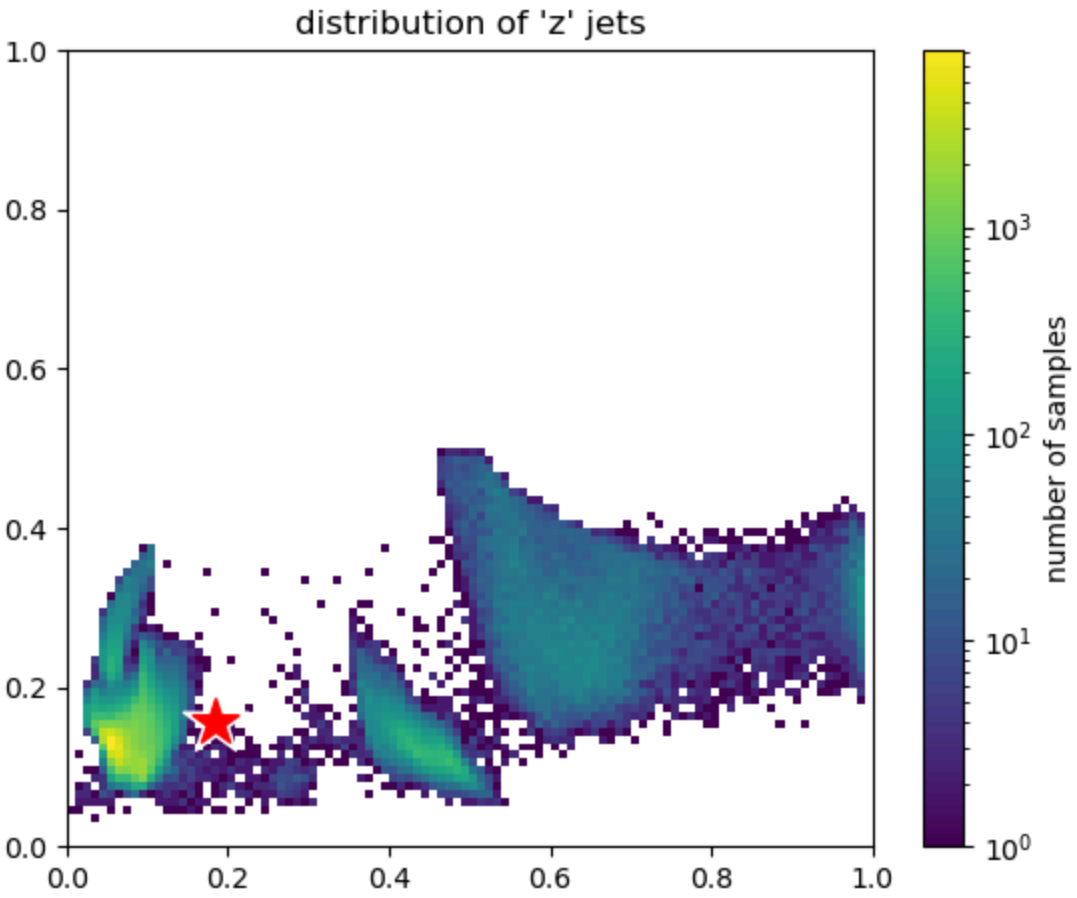
In [32]:
```
fig, axs = plt.subplots(5, 1, figsize=(6.5, 30))

ps = []
for (ax, (name, embedded)) in zip(axs, [
    ["g", embedded_g], ["q", embedded_q], ["t", embedded_t], ["w", embedded_
]):
    ps.append(ax.hist2d(embedded[:, 0], embedded[:, 1], bins=(100, 100), rar
    fig.colorbar(ps[-1][-1], ax=ax, label="number of samples")
    ax.scatter([embedded[:, 0].mean()], [embedded[:, 1].mean()], marker="*",
    ax.scatter([embedded[:, 0].mean()], [embedded[:, 1].mean()], marker="*",
    ax.set_title(f"distribution of '{name}' jets")
    ax.axis([0, 1, 0, 1])

None
```


distribution of 'g' jets
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
10^2
10^1
10^0
number of samples


distribution of 'q' jets
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
10^3
10^2
10^1
10^0
number of samples


distribution of 't' jets
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
10^3
10^2
10^1
10^0
number of samples


distribution of 'w' jets
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
10^3
10^2
10^1
10^0
number of samples



Not quite. `'g'`, `'q'`, `'t'` populate different clusters from each other, although the model split them up with more granularity.

The `'w'`, `'z'` are different from the quark-gluon jets, but not different from each other.

It would be interesting to map these clusters back to the original 16-dimensional jets, to understand what these phenominological clusters mean, but not now.

Moving on!

Variational autoencoder

Since we're interested in clusters in the autoencoder's "pinch point," why not encode them as distributions?

- An ordinary autoencoder maps input data to *points* in a small-dimensional space, such as $(x_1, x_2, \ldots x_n)$.
- A variational autoencoder maps input data to *parameters of distributions*, such as $(\mu_1, \sigma_1, \mu_2, \sigma_2, \ldots, \mu_n, \sigma_n)$.

Values in the next layer are randomly generated from these distributions.

Thus, there are now three types of vector-transformation in the neural networks we have considered:

1. linear transformations
2. non-linear activation functions
3. random generation from distribution parameters.

Moving on!

Convolutional neural network

The jet substructure dataset that we have been fitting consists of 16 hand-crafted features:

In [33]:
```
list(hls4ml_lhc_jets_hlf["data"].columns)
```

Out[33]:
```
['zlogz',
 'c1_b0_mmdt',
 'c1_b1_mmdt',
 'c1_b2_mmdt',
 'c2_b1_mmdt',
 'c2_b2_mmdt',
 'd2_b1_mmdt',
 'd2_b2_mmdt',
 'd2_a1_b1_mmdt',
 'd2_a1_b2_mmdt',
 'm2_b1_mmdt',
 'm2_b2_mmdt',
 'n2_b1_mmdt',
 'n2_b2_mmdt',
 'mass_mmdt',
 'multiplicity']
```

But what if we don't know what are the best features to use?

What if these 16 aren't the best features?

Suppose, instead, we start with the raw data (ECAL and HCAL clusters).

The jet substructure is presented in its lowest-level form: images.

This file contains individual jet images, labeled by `'g'`, `'q'`, `'t'`, `'w'`, `'z'`.

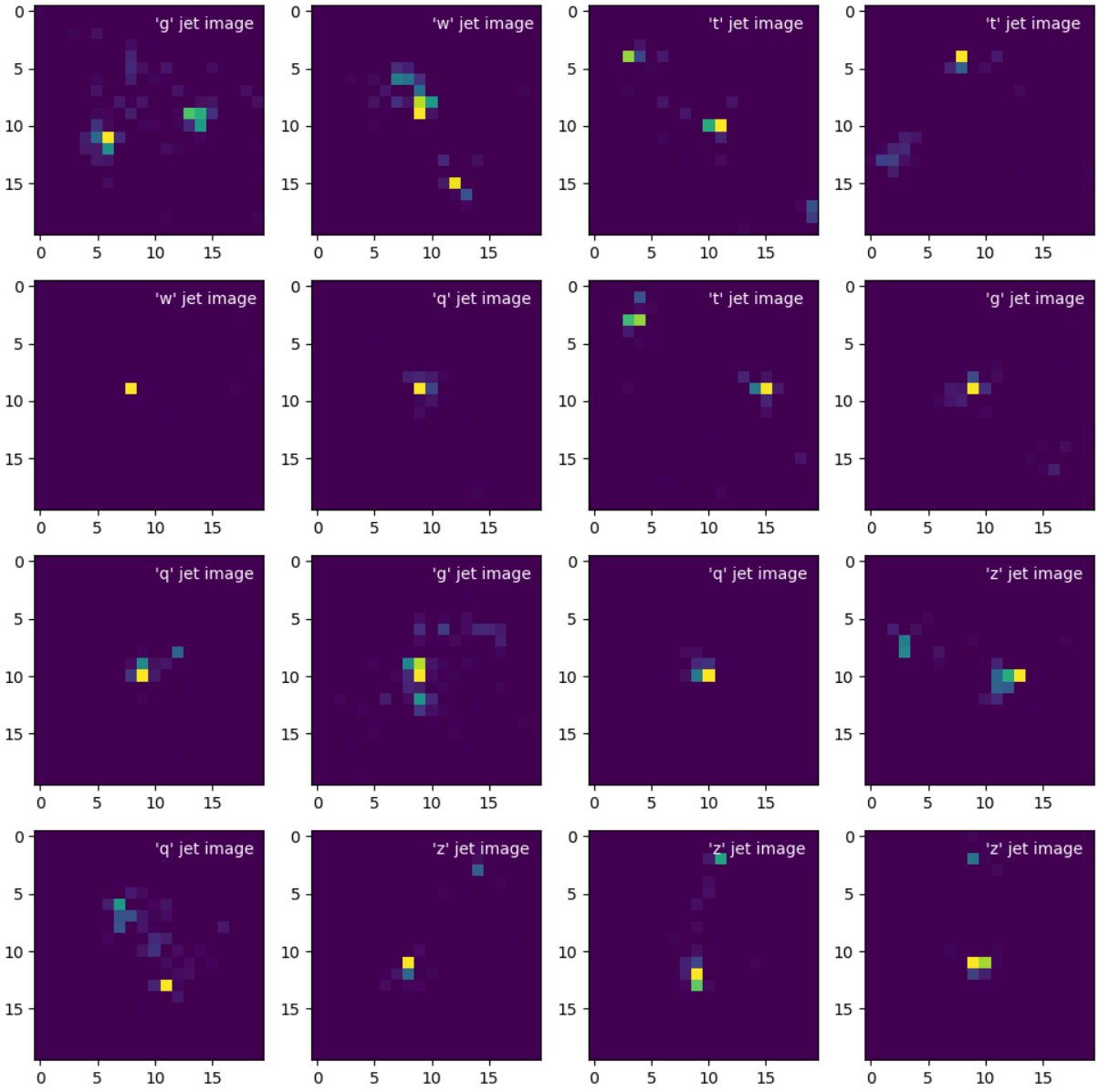
In [34]:
```
with h5py.File("../data/jet-images.h5") as file:
    jet_images = file["images"][:]
    jet_labels = file["labels"][:]

jet_label_order = ["g", "q", "t", "w", "z"]
```

There are $80\,000$ images with 20×20 pixels each.

In [35]:
```
jet_images.shape
```

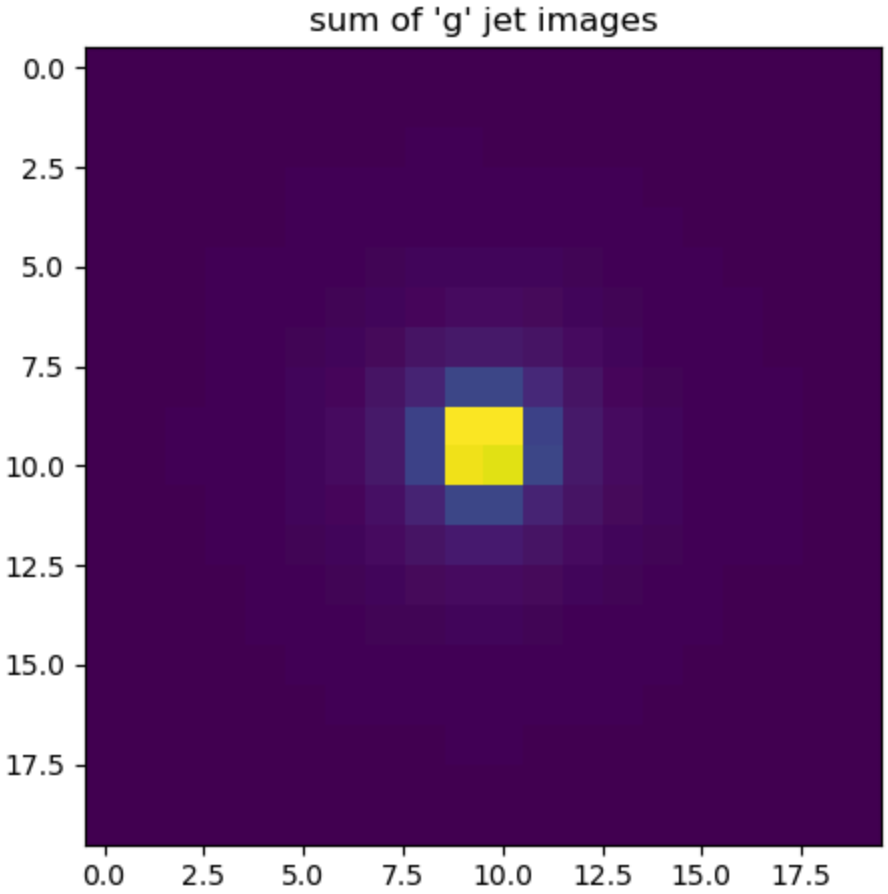
Out[35]:
```
(80000, 20, 20)
```

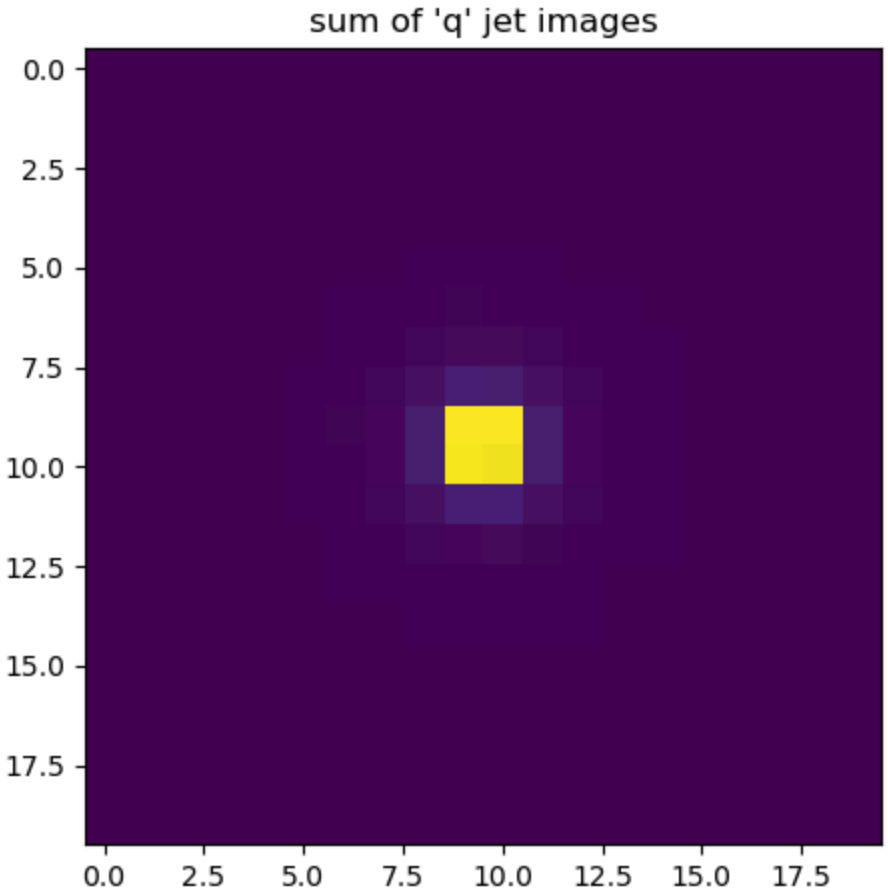
In [36]:
```
fig, axs = plt.subplots(4, 4, figsize=(12, 12))

for i, ax in enumerate(axs.flatten()):
    ax.imshow(jet_images[i])
    ax.text(10, 1.5, f"'{jet_label_order[jet_labels[i]]}' jet image", color=

None
```

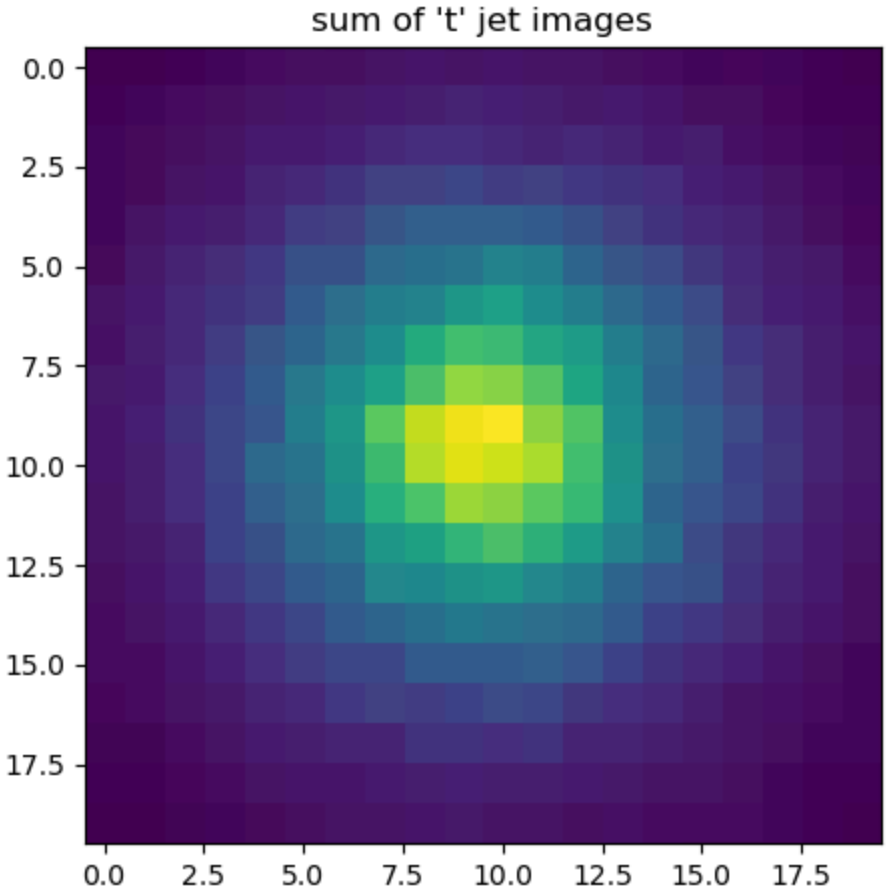


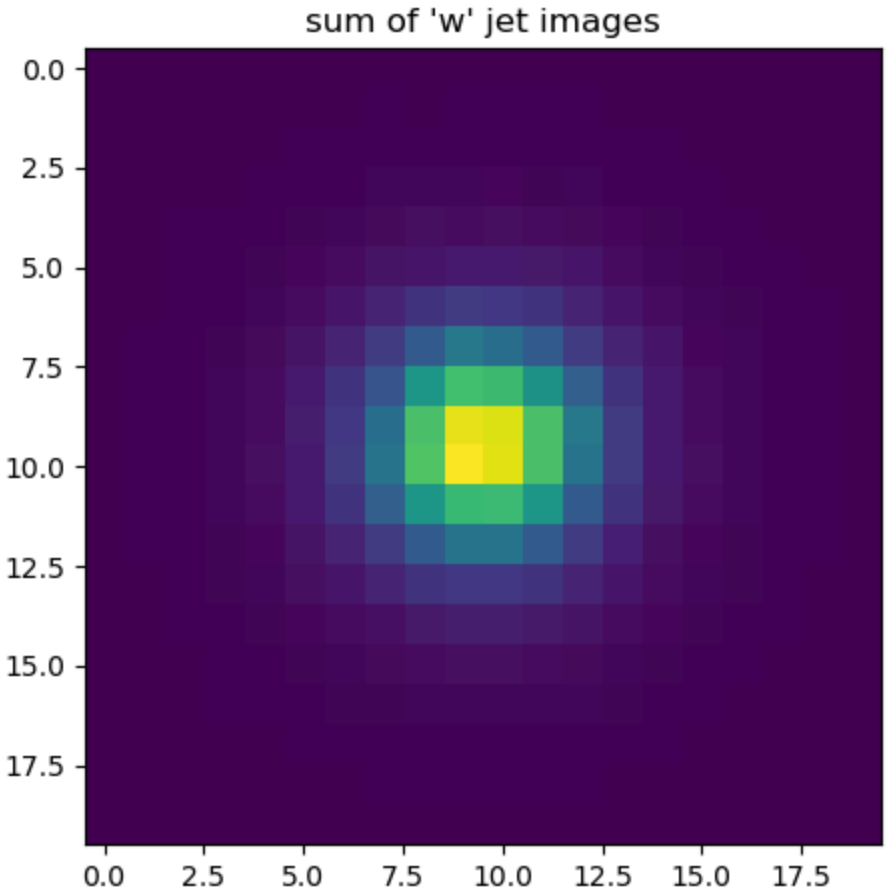
In [37]:
```
fig, axs = plt.subplots(5, 1, figsize=(6, 30))

for i, ax in enumerate(axs):
    ax.imshow(np.sum(jet_images[jet_labels == i], axis=0))
    ax.set_title(f"sum of '{jet_label_order[i]}' jet images")
```

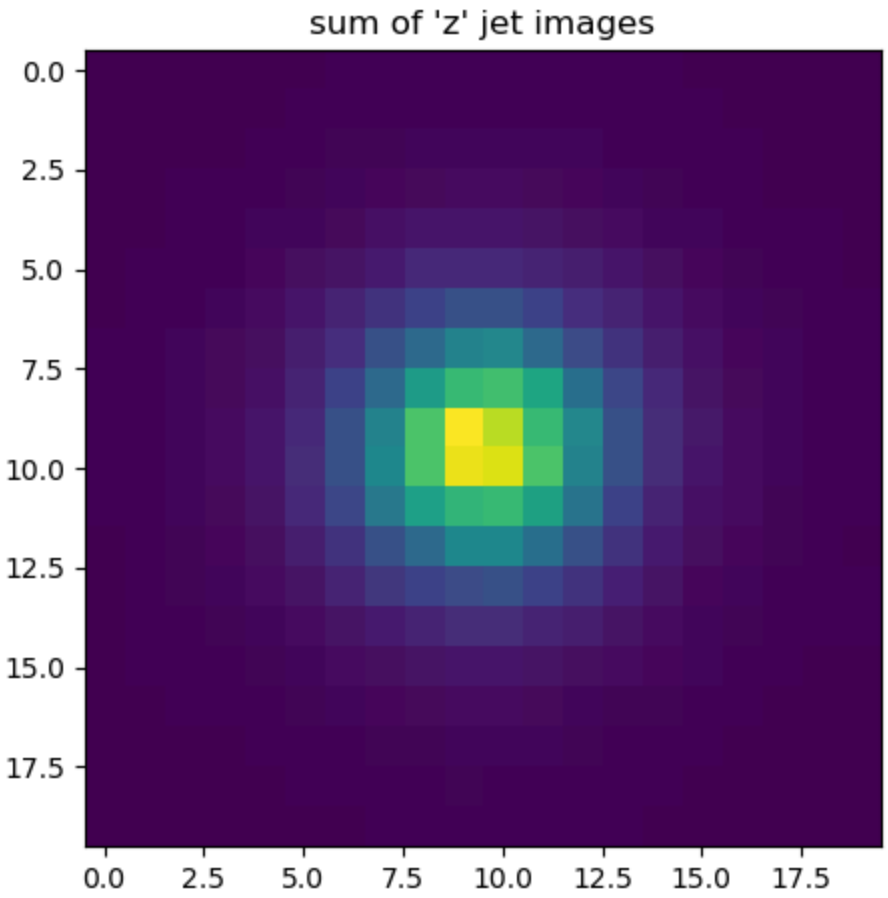
```
None
```


sum of 'g' jet images
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


sum of 'q' jet images
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


sum of 't' jet images
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


sum of 'w' jet images
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

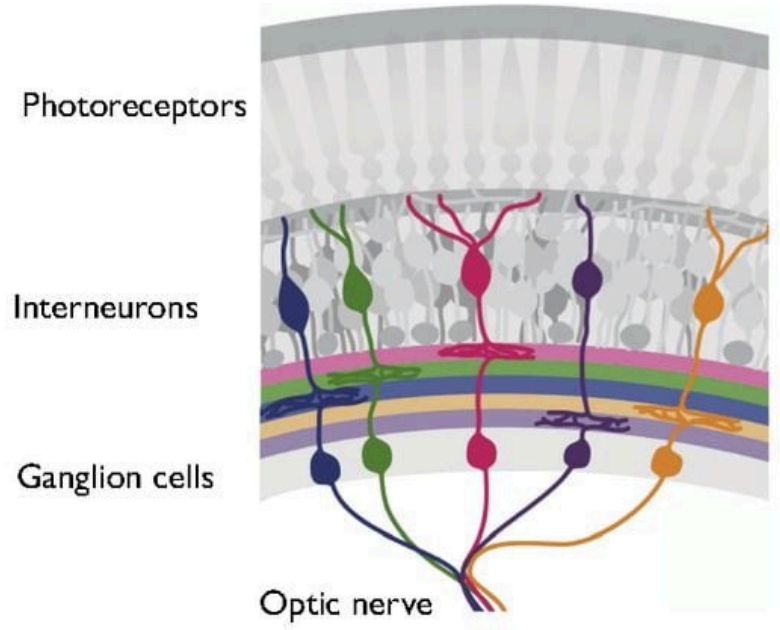


Instead of a 16-dimensional input space, this dataset has a 400-dimensional input space (20 pixels times 20 pixels).

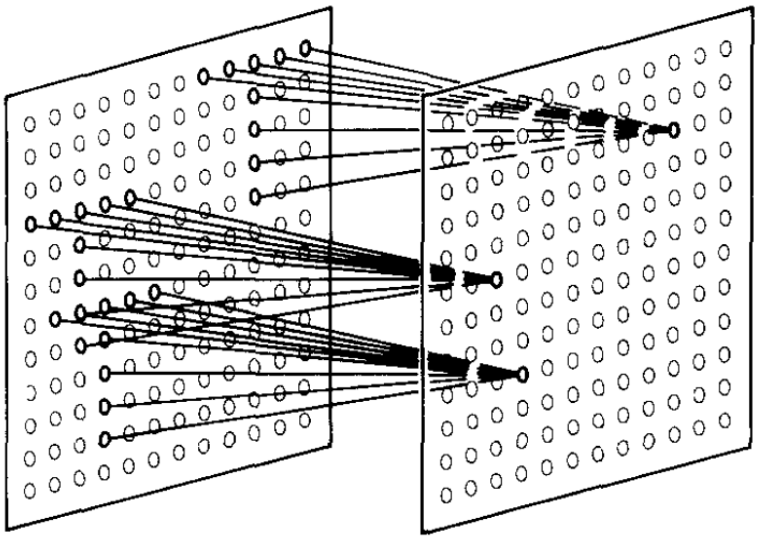
Just one fully connected layer would be a matrix with $400^2 = 160\,000$ parameters to fit.

We only have $80\,000$ images, so it would be highly overfitted.

Taking inspiration from biology (again):



Photoreceptors for distant spatial points are not directly connected. Only nearby points are connected in the first layer.



A linear transformation of only nearby points is known as a convolution, so this is called a **Convolutional Neural Network** (CNN).

Before talking about it in detail, let's run a convolutional network on the jet images.

In [38]:
```
jet_images_tensor = torch.tensor(jet_images)[:, np.newaxis, :, :]
jet_labels_tensor = torch.tensor(jet_labels)
```

PyTorch wants this shape: (number of images, number of channels, height in pixels, width in pixels), so we make the 1 channel explicit with `np.newaxis`.

(An RGB image would have 3 channels, etc.)

In [39]:
```
jet_images_tensor.shape
```

Out[39]:
```
torch.Size([80000, 1, 20, 20])
```

A PyTorch nn.Conv2d has enough tunable parameters to describe a fixed-size convolution matrix (3×3 below) from a number of input channels (1 below) to a number of output channels (1 below).

The number of parameters *does not* scale with the size of the image.

In [40]:
```
list(nn.Conv2d(1, 1, 3).parameters())
```

Out[40]:
```
[Parameter containing:
 tensor([[[[ 0.2542, -0.1283,  0.3229],
           [-0.0586, -0.0620,  0.1211],
           [ 0.0347,  0.0209, -0.1415]]]], requires_grad=True),
 Parameter containing:
 tensor([0.1391], requires_grad=True)]
```

A PyTorch nn.MaxPool2d scales down an image by a fixed factor, by taking the maximum value in every $n \times n$ block.

It has *no* tunable parameters.

Although not strictly necessary, it's a generally useful practice to pool convolutions, to reduce the total number of parameters and sensitivity to noise.

In [41]:
```
list(nn.MaxPool2d(2).parameters())
```

Out[41]:
```
[]
```

The general strategy is to reduce the size of the image with each convolution (and max-pooling) while increasing the number of channels, so that the spatial grid gradually becomes an abstract vector.

Then do a normal fully-connected network to classify the vectors.

In [49]:
```
class ConvolutionalClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.convolutional1 = nn.Sequential(
            nn.Conv2d(1, 5, 5),     # 1 input channel → 5 output channels, 5
```

```
            nn.ReLU(),              #     input image: 20×20, convoluted ima
            nn.MaxPool2d(2),        # scales down by taking the max in 2×2 s
        )
        self.convolutional2 = nn.Sequential(
            nn.Conv2d(5, 10, 5),    # 5 input channels → 10 output channels,
            nn.ReLU(),              #     input image: 8×8, convoluted image
            nn.MaxPool2d(2),        # scales down by taking the max in 2×2 s
        )
        self.fully_connected = nn.Sequential(
            nn.Linear(10 * 2*2, 30),
            nn.ReLU(),
            nn.Linear(30, 20),
            nn.ReLU(),
            nn.Linear(20, 10),
            nn.ReLU(),
            nn.Linear(10, 5),
        )

    def forward(self, x):
        return self.fully_connected(torch.flatten(self.convolutional2(self.c

model_without_softmax = ConvolutionalClassifier()
```

Although this has a lot of parameters ($3\,505$), it's less than the number of images ($80\,000$), which is much less than the number of inputs.

In [50]:
```
num_model_parameters = 0
for tensor_parameter in model_without_softmax.parameters():
    num_model_parameters += tensor_parameter.detach().numpy().size

num_model_parameters, len(jet_images_tensor)
```

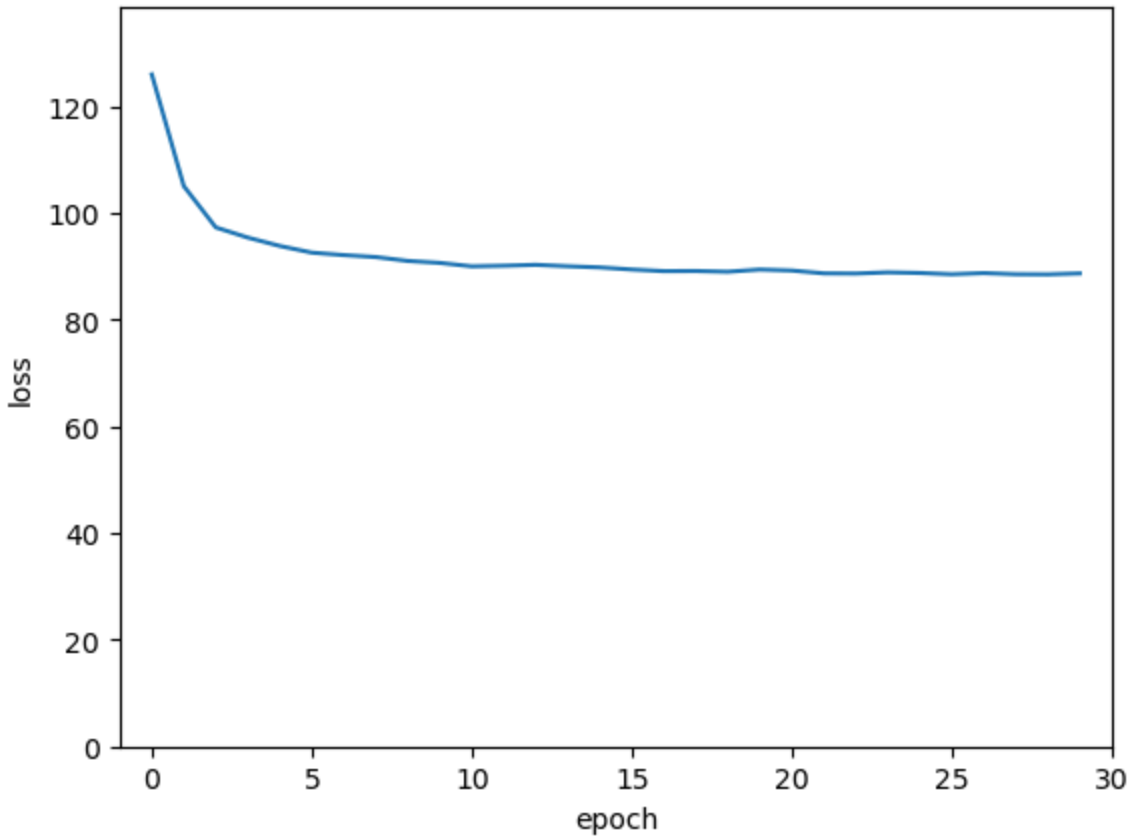
Out[50]:
```
(3505, 80000)
```

In [51]:
```
NUM_EPOCHS = 30
BATCH_SIZE = 1000

loss_function = nn.CrossEntropyLoss()

optimizer = optim.Adam(model_without_softmax.parameters(), lr=0.03)

loss_vs_epoch = []
for epoch in range(NUM_EPOCHS):
    total_loss = 0

    for start_batch in range(0, len(jet_images_tensor), BATCH_SIZE):
        stop_batch = start_batch + BATCH_SIZE

        optimizer.zero_grad()

        predictions = model_without_softmax(jet_images_tensor[start_batch:st
        loss = loss_function(predictions, jet_labels_tensor[start_batch:stop
        total_loss += loss.item()

        loss.backward()
```

```
        optimizer.step()

    loss_vs_epoch.append(total_loss)
    print(f"{epoch = } {total_loss = }")
```

```
epoch = 0 total_loss = 125.98868465423584
epoch = 1 total_loss = 105.05784094333649
epoch = 2 total_loss = 97.30704808235168
epoch = 3 total_loss = 95.40055525302887
epoch = 4 total_loss = 93.83064866065979
epoch = 5 total_loss = 92.57916164398193
epoch = 6 total_loss = 92.11763167381287
epoch = 7 total_loss = 91.75242173671722
epoch = 8 total_loss = 91.02377593517303
epoch = 9 total_loss = 90.6557742357254
epoch = 10 total_loss = 90.00963199138641
epoch = 11 total_loss = 90.12515044212341
epoch = 12 total_loss = 90.29330635070801
epoch = 13 total_loss = 90.02583837509155
epoch = 14 total_loss = 89.80913865566254
epoch = 15 total_loss = 89.41609859466553
epoch = 16 total_loss = 89.11247527599335
epoch = 17 total_loss = 89.13321208953857
epoch = 18 total_loss = 88.99730014801025
epoch = 19 total_loss = 89.40856397151947
epoch = 20 total_loss = 89.22476196289062
epoch = 21 total_loss = 88.69160795211792
epoch = 22 total_loss = 88.66105616092682
epoch = 23 total_loss = 88.88048851490021
epoch = 24 total_loss = 88.76704370975494
epoch = 25 total_loss = 88.52432489395142
epoch = 26 total_loss = 88.74389314651489
epoch = 27 total_loss = 88.53080296516418
epoch = 28 total_loss = 88.50077390670776
epoch = 29 total_loss = 88.68323290348053
```

In [52]:
```
fig, ax = plt.subplots()

ax.plot(range(len(loss_vs_epoch)), loss_vs_epoch)
ax.set_xlim(-1, len(loss_vs_epoch))
ax.set_ylim(0, 1.1*max(loss_vs_epoch))
ax.set_xlabel("epoch")
ax.set_ylabel("loss")

None
```



Let's see the accuracy, in terms of the confusion matrix.

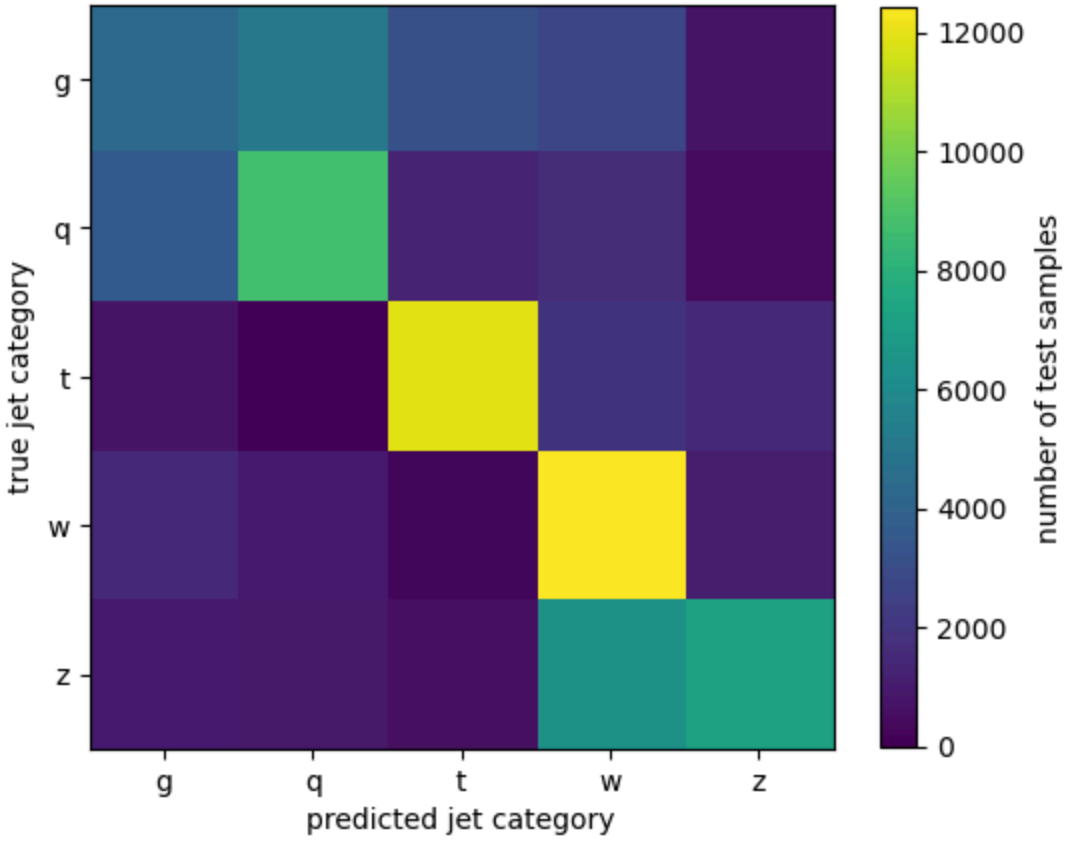
In [53]:
```
model_with_softmax = nn.Sequential(
    model_without_softmax,
    nn.Softmax(dim=1),
)
```

In [54]:
```
predictions_tensor = model_with_softmax(jet_images_tensor)

confusion_matrix = np.array(
    [
        [
            (predictions_tensor[jet_labels_tensor == true_class].argmax(axis
            for prediction_class in range(5)
        ]
        for true_class in range(5)
    ]
)
confusion_matrix
```

Out[54]:
```
array([[ 4433,  5166,  3105,  2629,   694],
       [ 3652,  8662,  1347,  1652,   415],
       [  635,    95, 11915,  1927,  1503],
       [ 1502,   956,   195, 12425,  1022],
       [  998,   869,   556,  6444,  7203]])
```

In [55]:
```
fig, ax = plt.subplots(figsize=(6, 6))

image = ax.imshow(confusion_matrix, vmin=0)
```

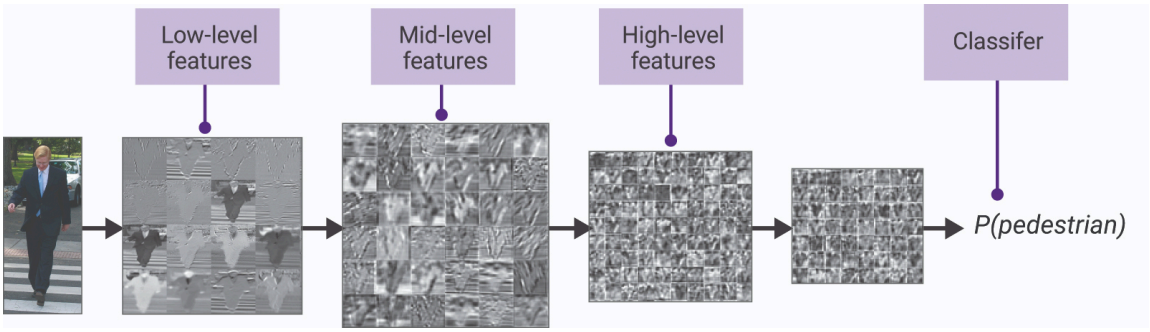
```
fig.colorbar(image, ax=ax, label="number of test samples", shrink=0.8)

ax.set_xticks(range(5), jet_label_order)
ax.set_yticks(range(5), jet_label_order)

ax.set_xlabel("predicted jet category")
ax.set_ylabel("true jet category")

None
```



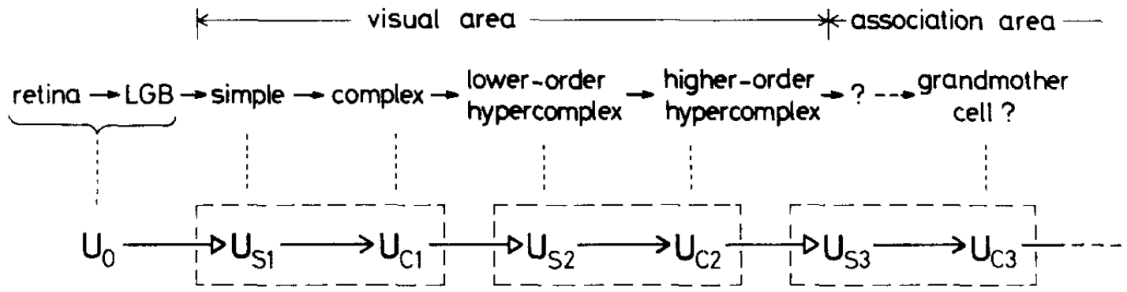
Long before neural networks, (hand-coded) convolutions were used to detect edges in images.

These are low-level features of the image.

By repeating this process, convolutional neural networks



Kunihiko Fukushima, *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position* (1980).



"Grandmother cell" refers to an old hypothesis that, in the human brain, *one cell* encodes a very high-level concept like one's grandmother.

It sounds far fetched, but when Google trained an unsupervised convolutional network on a large set of YouTube videos, they were surprised by *one neuron* that projected back onto the image space like this:

This is what started the whole "AI discovers cats on the internet" thing.

There are now five types of vector-transformation in the neural networks we have considered:

1. linear transformations
2. non-linear activation functions
3. random generation from distribution parameters
4. small set of learned convolution parameters applied to large images
5. reducing an image size with pooling.

Moving on!

Ragged data

Suppose you have data like the following—how would you pass this into a neural network?

In [56]:
```
event_data = ak.from_parquet("../data/SMHiggsToZZTo4L.parquet")
event_data
```

Out[56]: [{run: 1, luminosityBlock: 156, event: 46501, PV: {...}, electron: [], ...},
{run: 1, luminosityBlock: 156, event: 46502, PV: {...}, electron: [...], ...},
{run: 1, luminosityBlock: 156, event: 46503, PV: {...}, electron: [...], ...},
{run: 1, luminosityBlock: 156, event: 46504, PV: {...}, electron: ..., ...},
{run: 1, luminosityBlock: 156, event: 46505, PV: {...}, electron: [...], ...},
{run: 1, luminosityBlock: 156, event: 46506, PV: {...}, electron: ..., ...},
{run: 1, luminosityBlock: 156, event: 46507, PV: {...}, electron: ..., ...},
{run: 1, luminosityBlock: 156, event: 46508, PV: {...}, electron: ..., ...},
{run: 1, luminosityBlock: 156, event: 46509, PV: {...}, electron: [...], ...},
{run: 1, luminosityBlock: 156, event: 46510, PV: {...}, electron: [], ...},
...,
{run: 1, luminosityBlock: 996, event: 298792, PV: {...}, electron: [...], ...},
{run: 1, luminosityBlock: 996, event: 298793, PV: {...}, electron: ..., ...},
{run: 1, luminosityBlock: 996, event: 298794, PV: {...}, electron: [], ...},
{run: 1, luminosityBlock: 996, event: 298795, PV: {...}, electron: [...], ...},
{run: 1, luminosityBlock: 996, event: 298796, PV: {...}, electron: [], ...},
{run: 1, luminosityBlock: 996, event: 298797, PV: {...}, electron: ..., ...},
{run: 1, luminosityBlock: 996, event: 298798, PV: {...}, electron: [], ...},
{run: 1, luminosityBlock: 996, event: 298799, PV: {...}, electron: [...], ...},
{run: 1, luminosityBlock: 996, event: 298800, PV: {...}, electron: [...], ...}]

```
type: 299973 * {
    run: int32,
    luminosityBlock: int64,
    event: uint64,
    PV: Vector3D[
        x: float32,
        y: float32,
        z: float32
    ],
    electron: var * Momentum4D[
        pt: float32,
        eta: float32,
        phi: float32,
        mass: float32,
        charge: int32,
        pfRelIso03_all: float32,
        dxy: float32,
        dxyErr: float32,
        dz: float32,
        dzErr: float32
    ],
```

```
    muon: var * Momentum4D[
        pt: float32,
        eta: float32,
        phi: float32,
        mass: float32,
        charge: int32,
        pfRelIso03_all: float32,
        pfRelIso04_all: float32,
        dxy: float32,
        dxyErr: float32,
        dz: float32,
        dzErr: float32
    ],
    MET: Momentum2D[
        pt: float32,
        phi: float32
    ]
}
```

In [57]:
```
event_data.muon.pt
```

Out[57]:
```
[[63, 38.1, 4.05],
 [],
 [],
 [54.3, 23.5, 52.9, 4.33, 5.35, 8.39, 3.49],
 [],
 [38.5, 47],
 [4.45],
 [],
 [],
 [],
 ...,
 [37.2, 50.1],
 [43.2, 24],
 [24.2, 79.5],
 [],
 [9.81, 25.5],
 [32.6, 43.1],
 [4.32, 4.36, 5.63, 4.75],
 [],
 []]
-----------------------------------------
type: 299973 * var * float32
```

PyTorch has a torch.nested.nested_tensor that can represent ragged numerical data.

(Notice that we have to turn this into Python lists first! PyTorch's issue #112509 asks to fix this, and TensorFlow's tf.RaggedTensor doesn't have this problem.)

In [58]:
```
muon_pt_tensor = torch.nested.nested_tensor(ak.to_list(event_data.muon.pt))
```

```
/Users/jpivarski/miniforge3/lib/python3.11/site-packages/torch/nested/__init
__.py:166: UserWarning: The PyTorch API of nested tensors is in prototype st
age and will change in the near future. (Triggered internally at /Users/runn
er/miniforge3/conda-bld/libtorch_1719361060788/work/aten/src/ATen/NestedTens
orImpl.cpp:180.)
  return _nested.nested_tensor(
```

In [59]:
```
muon_pt_tensor[0]
```

Out[59]:
```
tensor([63.0439, 38.1203,  4.0487])
```

In [60]:
```
muon_pt_tensor[1]
```

Out[60]:
```
tensor([])
```

In [61]:
```
muon_pt_tensor[2]
```

Out[61]:
```
tensor([])
```

In [62]:
```
muon_pt_tensor[3]
```

Out[62]:
```
tensor([54.3327, 23.5153, 52.8711,  4.3286,  5.3478,  8.3934,  3.4901])
```

Unfortunately, one of the few things that you can do with it is turn it into a regular array by padding. (See ak.pad_none and ak.fill_none in Awkward Array.)

In [63]:
```
torch.nested.to_padded_tensor(muon_pt_tensor, -1)
```

```
Out[63]: tensor([[63.0439, 38.1203,  4.0487,  ..., -1.0000, -1.0000, -1.0000],
                 [-1.0000, -1.0000, -1.0000,  ..., -1.0000, -1.0000, -1.0000],
                 [-1.0000, -1.0000, -1.0000,  ..., -1.0000, -1.0000, -1.0000],
                 ...,
                 [ 4.3161,  4.3588,  5.6327,  ..., -1.0000, -1.0000, -1.0000],
                 [-1.0000, -1.0000, -1.0000,  ..., -1.0000, -1.0000, -1.0000],
                 [-1.0000, -1.0000, -1.0000,  ..., -1.0000, -1.0000, -1.0000]])
```

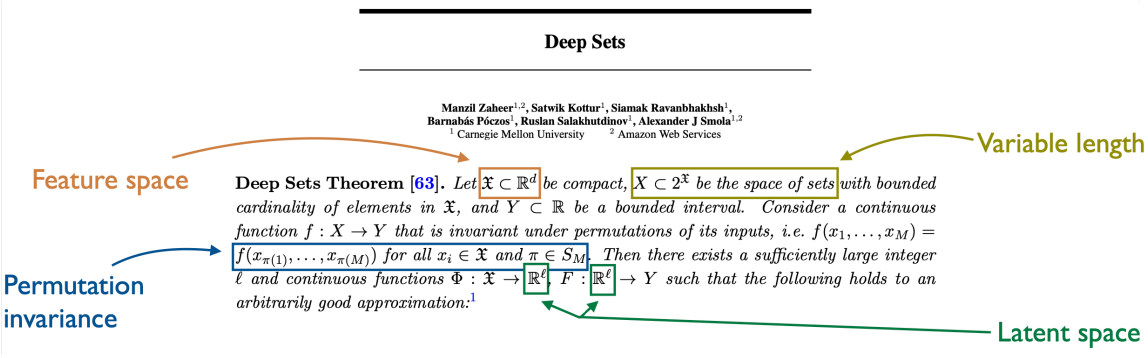
Other than computational inefficiency (iterating in Python, padded arrays in memory, extra-wide neural network layers to transform padded arrays), there are problems with data like this from a machine learning point of view.

A network trained on the padded tensor above would learn that many of the values on the right are `-1`, and it would learn *the exact order* of muon values (which might or might not be sorted).

We want a model to learn about the muons as *unsorted collections of objects*.

We want **permutation invariance**.

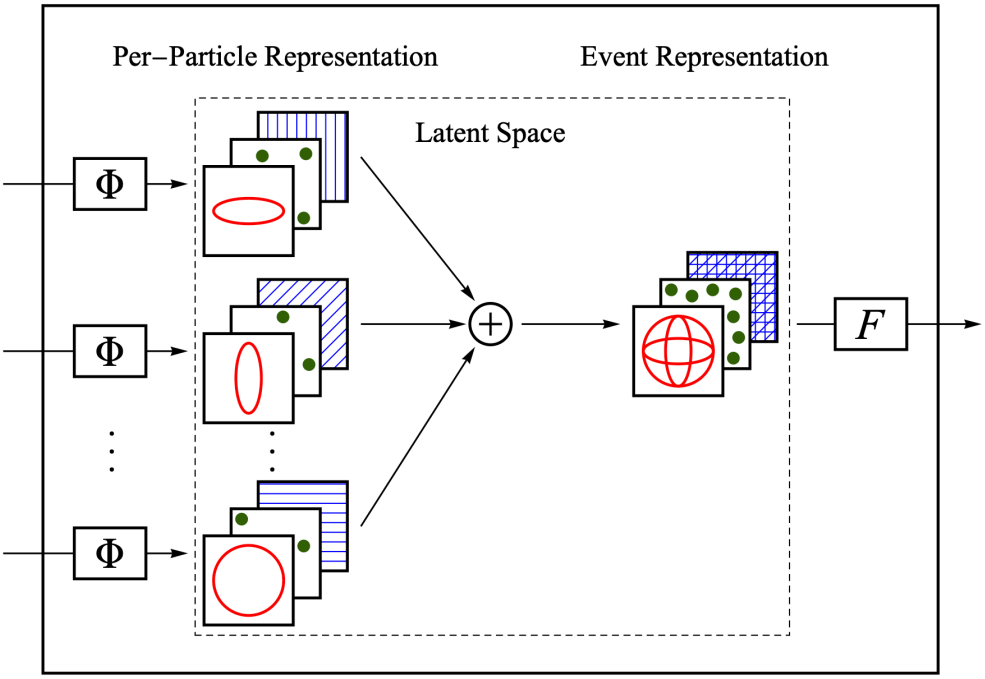
Zaheer, Kottur, Ravanbakhsh, Poczos, Salakhutdinov, & Smola, *Deep Sets* (2017):



That is, to approximate a function $f(x_1, x_2, \ldots, x_M)$ with no dependence on the order of x_1, x_2, ... x_M, you can transform each x_i into an independent vector, sum them, and then transform that vector.

These two transformations, Φ and F, can be neural networks, for complete generality.

$$f(x_1, x_2, \ldots, x_M) = F\left(\sum_{i=1}^{M} \Phi(x_i)\right)$$



The Φ functions should expand the x_i vectors to a larger space so that enough information is preserved when they're summed.

The larger the typical number of x_i (e.g. modeling all tracks, rather than just muons), the larger the output dimensionality of Φ should be.

The **latent space** (above) is the same kind of embedding that we saw in the pinch point of the autoencoder.

In [64]:
```
muon_kinematics = event_data["muon", ["pt", "eta", "phi"]]

muon_kinematics_tensor = torch.tensor(
    ak.to_numpy(ak.flatten(muon_kinematics)).view(np.float32).reshape(-1, 3)
)
muon_kinematics_tensor[:3]
```

Out[64]:
```
tensor([[63.0439, -0.7187,  2.9680],
        [38.1203, -0.8795, -1.0325],
        [ 4.0487, -0.3208,  1.0385]])
```

In [65]:
```
event_data[0, "muon", ["pt", "eta", "phi"]]
```

Out[65]:
```
[{pt: 63, eta: -0.719, phi: 2.97},
 {pt: 38.1, eta: -0.879, phi: -1.03},
 {pt: 4.05, eta: -0.321, phi: 1.04}]
-----------------------------------
type: 3 * {
  pt: float32,
  eta: float32,
  phi: float32
}
```

In [66]:
```
Phi = nn.Sequential(
    nn.Linear(3, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
)

F = nn.Sequential(
    nn.Linear(10, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
    nn.ReLU(),
)

start = 0
for i, count in enumerate(ak.num(muon_kinematics)):
    one_event = muon_kinematics_tensor[start : start + count]
    start += count

    prediction = F(torch.sum(Phi(one_event), axis=0, keepdims=True))
```

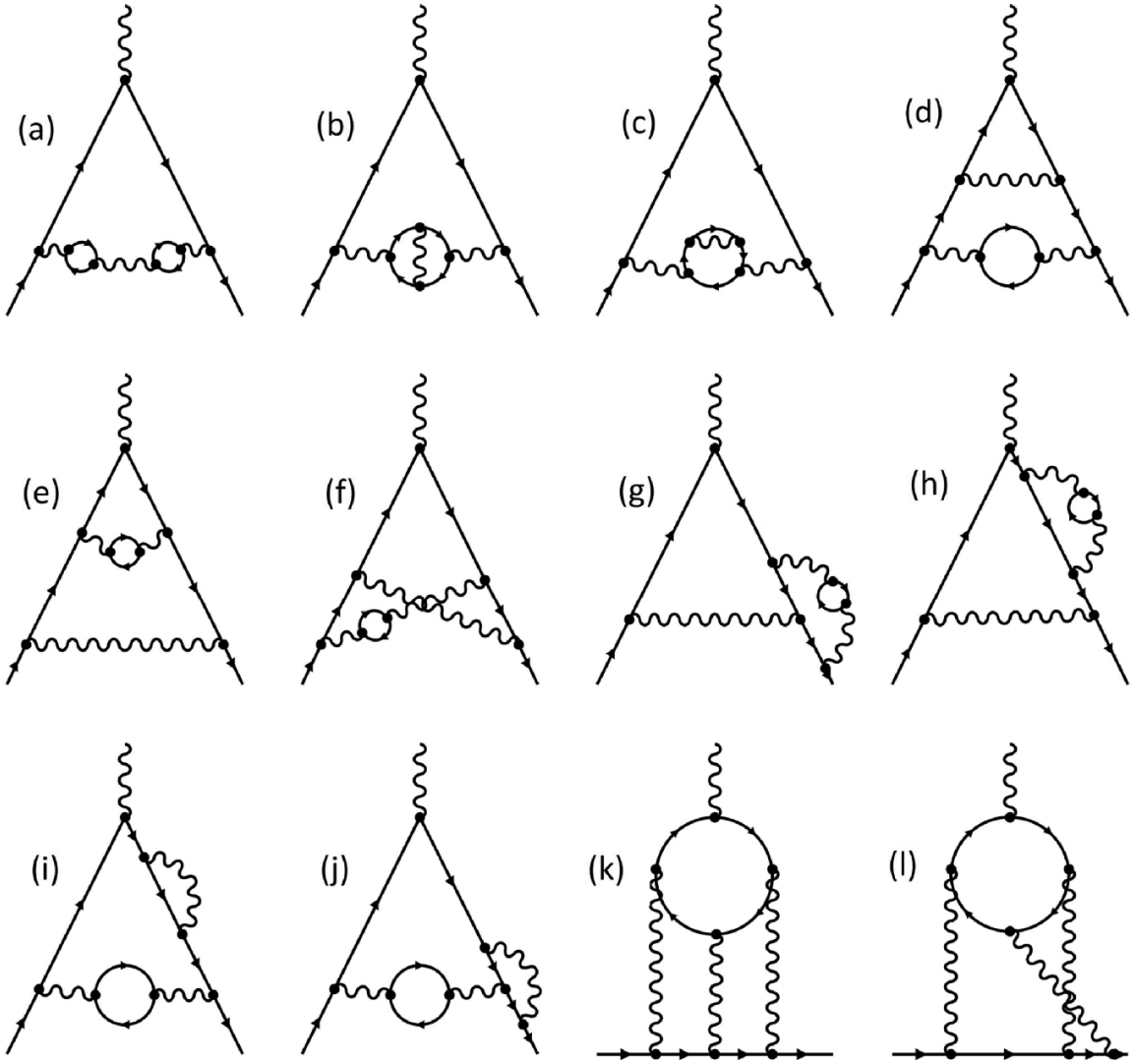
The growing list of vector-transformations that can be used in neural networks:

1. linear transformations
2. non-linear activation functions
3. random generation from distribution parameters
4. small set of learned convolution parameters applied to large images
5. reducing an image size with pooling
6. adding arbitrarily many neural network outputs to make the next neural network input.

Moving on!

Graph neural networks

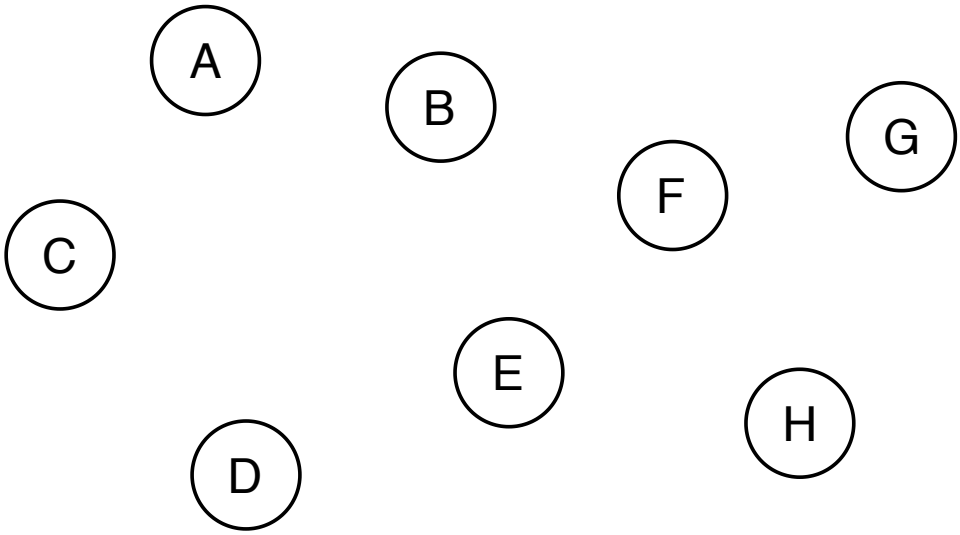
What's a graph?



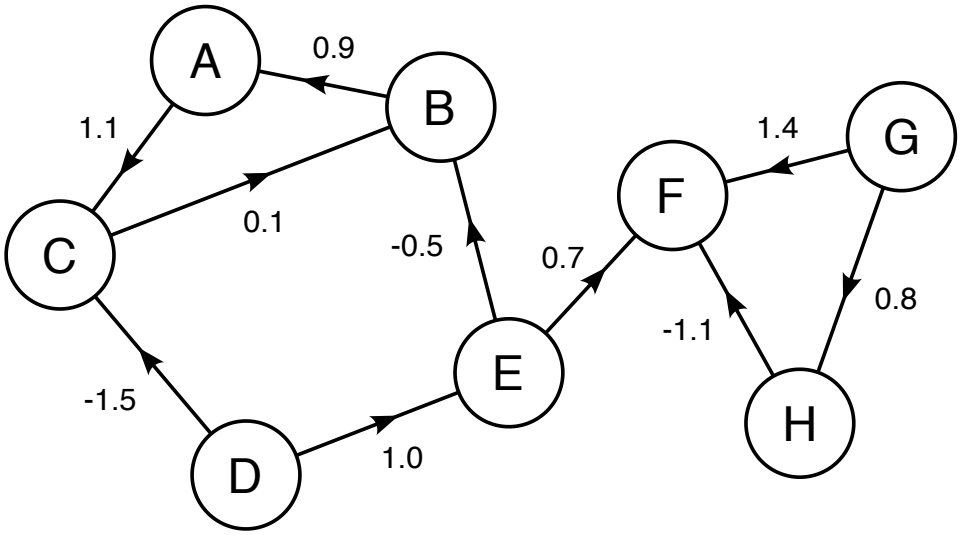
A graph consists of distinct nodes (points) connected by edges (lines), in which only the connections matter, not where they're located/how they're drawn on a page.

- Nodes may have properties, such as a label.
- Edges may have properties, such as directions and weights.

What's the difference between a set



and a graph?

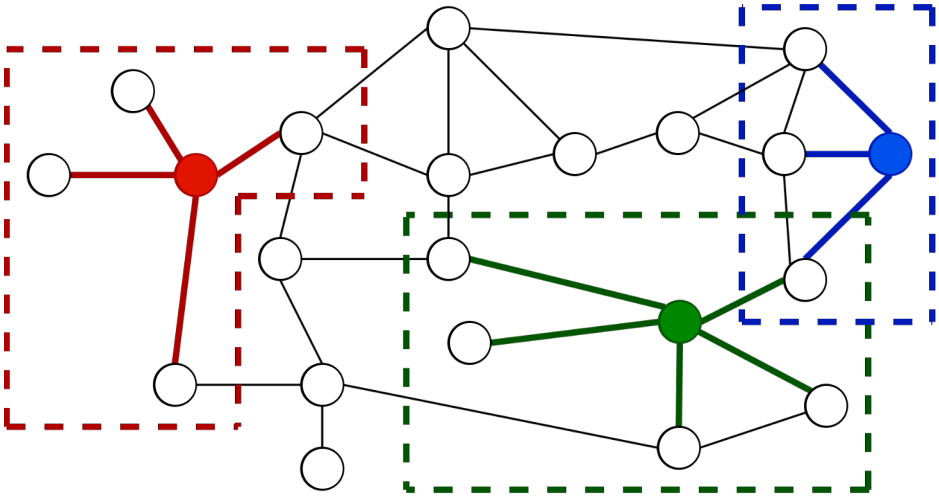


A graph is a set with edges.

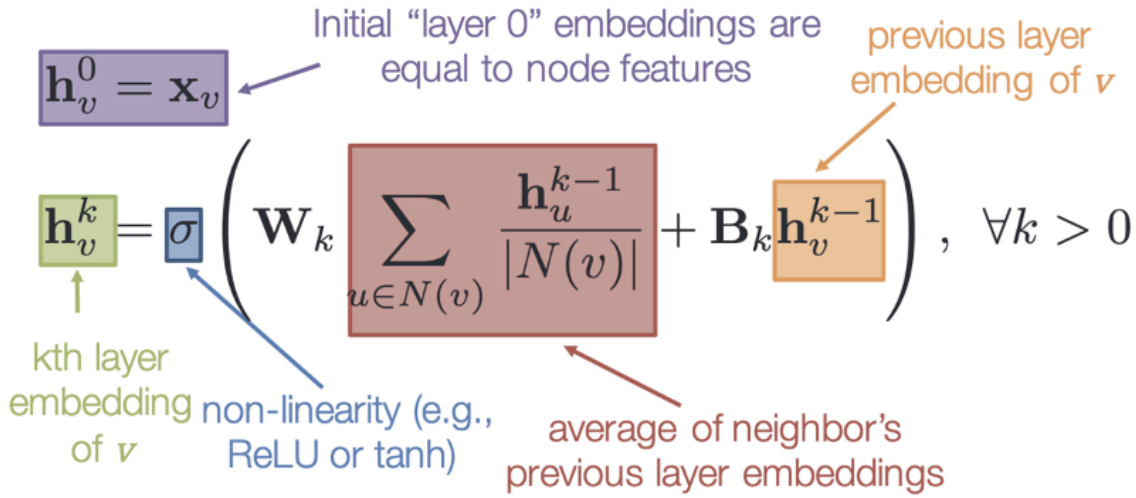
Graphs should have the same **permutation invariance** as sets.

We can model graph data in a way that is similar to DeepSets by adding an extra step that handles the edges.

Instead of adding $\Phi(x_i)$ for all nodes x_i equally, as in DeepSets, **Graph Neural Networks** (GNNs) sum over individual neighborhoods (in various ways).



Ward, Joyner, Lickfold, Guo, & Bennamoun, *A Practical Tutorial on Graph Neural Networks* (2020).



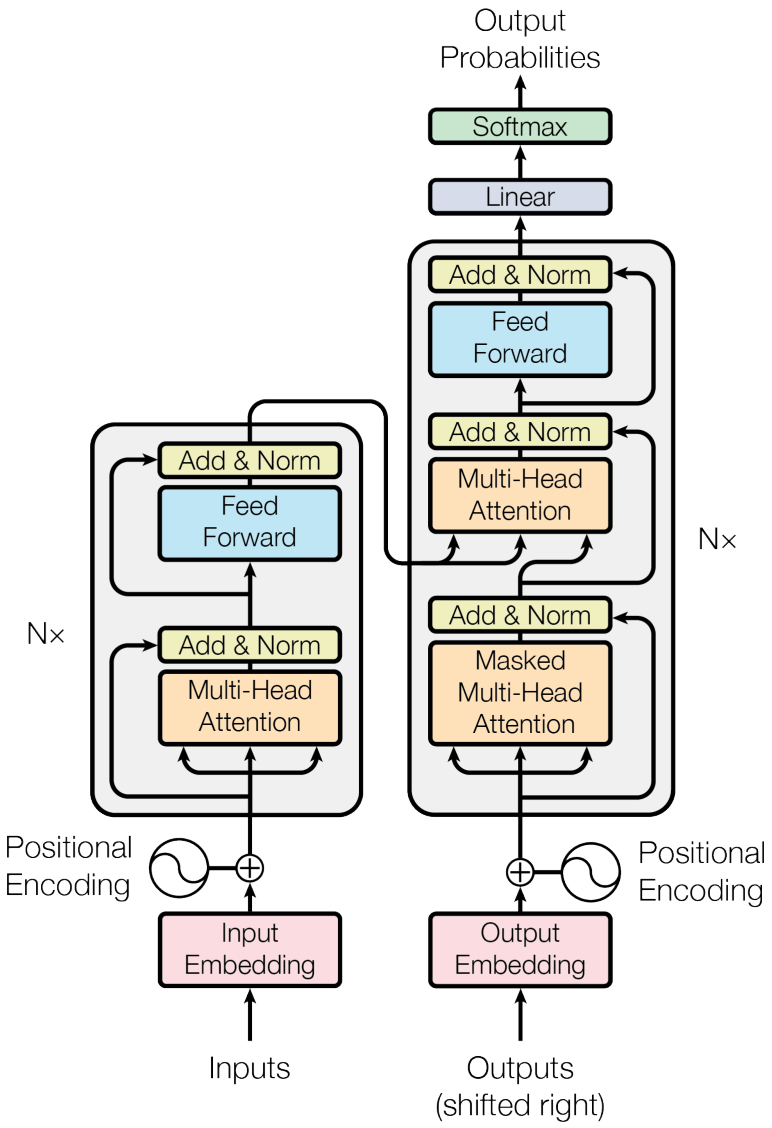
The design space for GNNs is huge, but they involve using ordinary neural networks to make **latent spaces** (a.k.a. embedding spaces) and summing (or maximizing) over edge-connected neighbors in the graph.

Papers and webpages are filled with diagrams and equations with lots of subscripts that try to express this connectivity.

(See the Neural Network Zoo.)

Transformers (such as ChatGPT)

You'll see this diagram everywhere—it's called a **transformer** architecture:



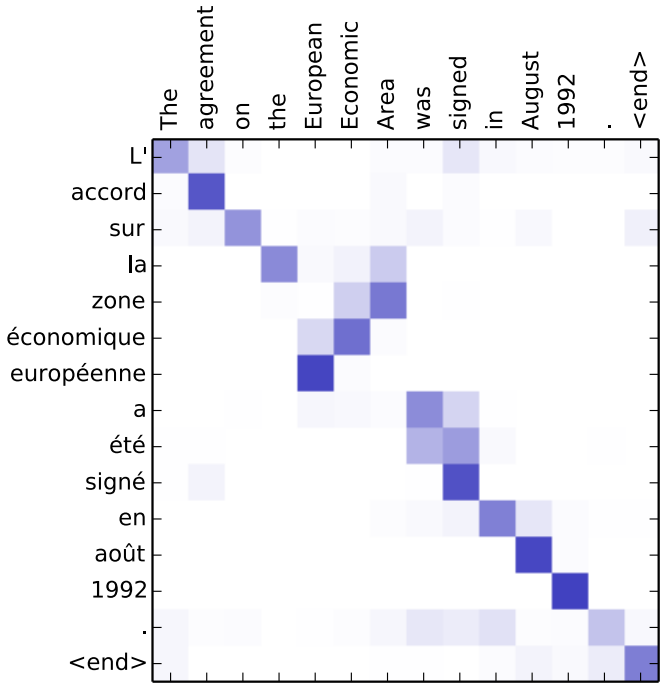
Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, & Polosukhin, *Attention Is All You Need* (2017).

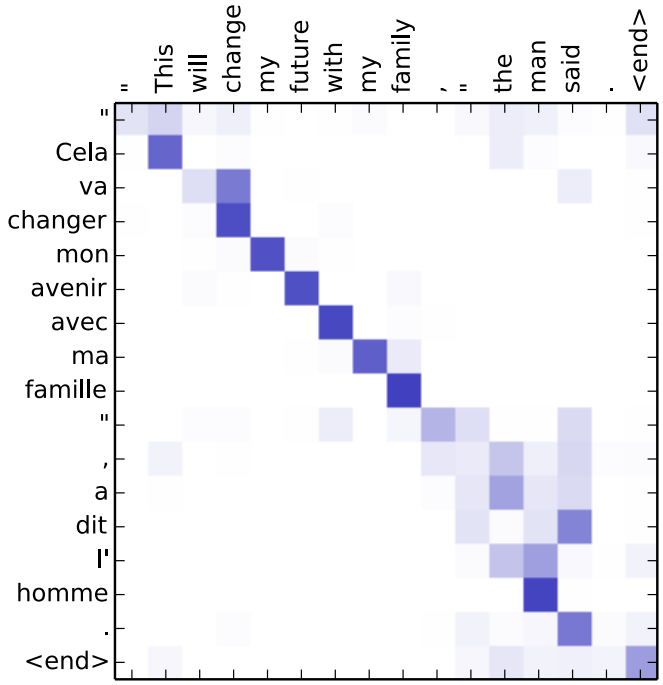
Although the most famous application of the transformer architecture is ChatGPT and other Large Language Models (LLMs), it is a generalization of GNNs and is likely applicable to HEP.

The key part of the transformer architecture is **attention**. (Hence the paper title, "Attention is all you need.")

Attention is a dynamic weight between all pairs of inputs, learned in the same optimization with the data themselves.

It was developed in the context of human language translation (Bahdanau, Cho, & Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate* (2014)):

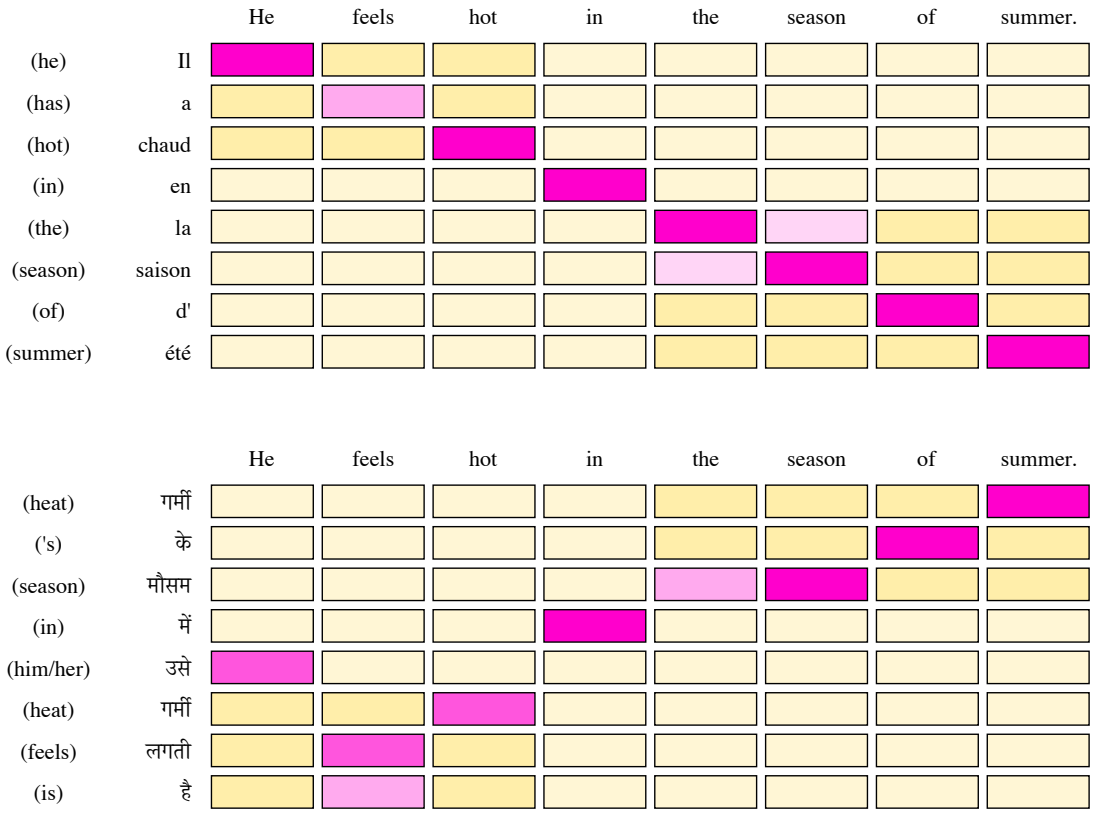




> la zone économique européenne

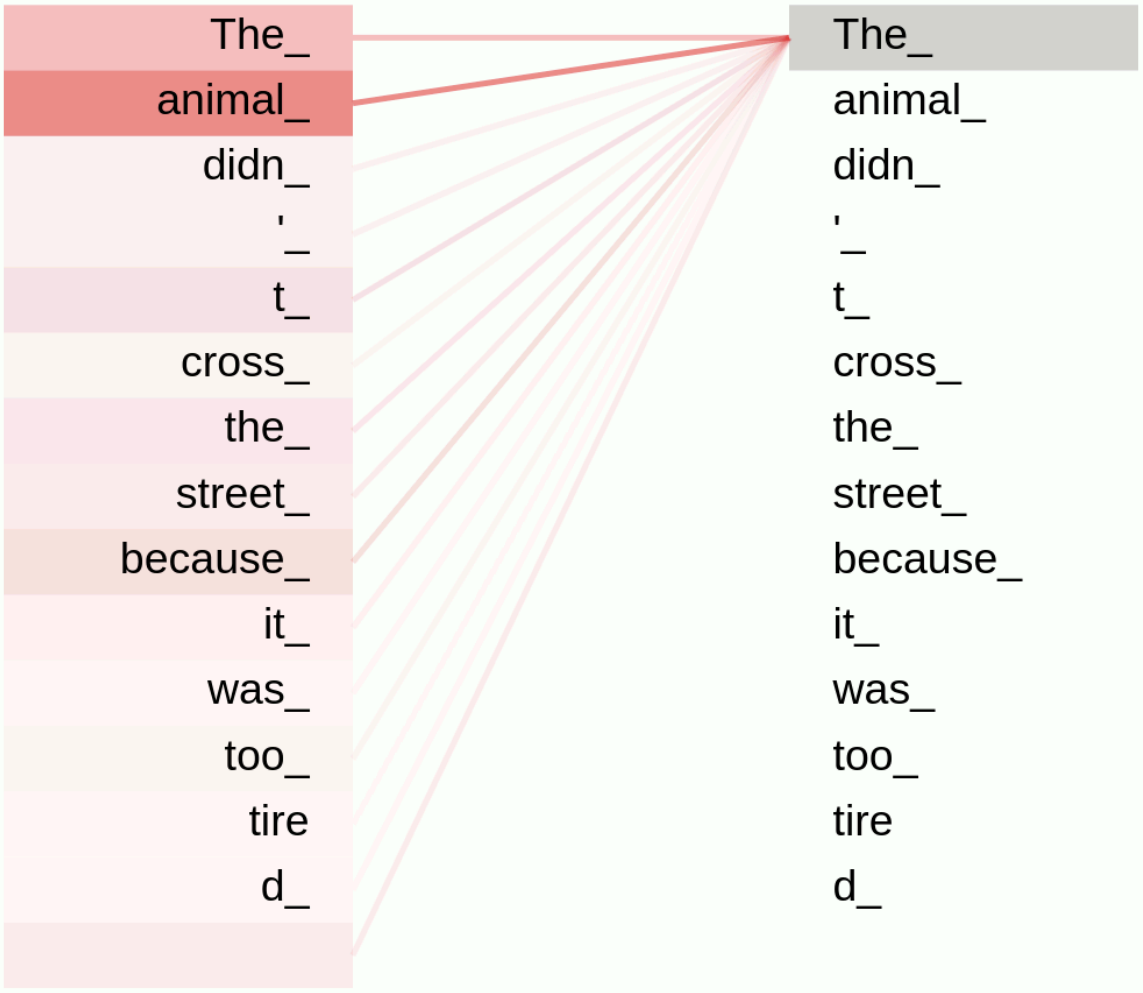
> the European Economic Area

Or more dramatically:

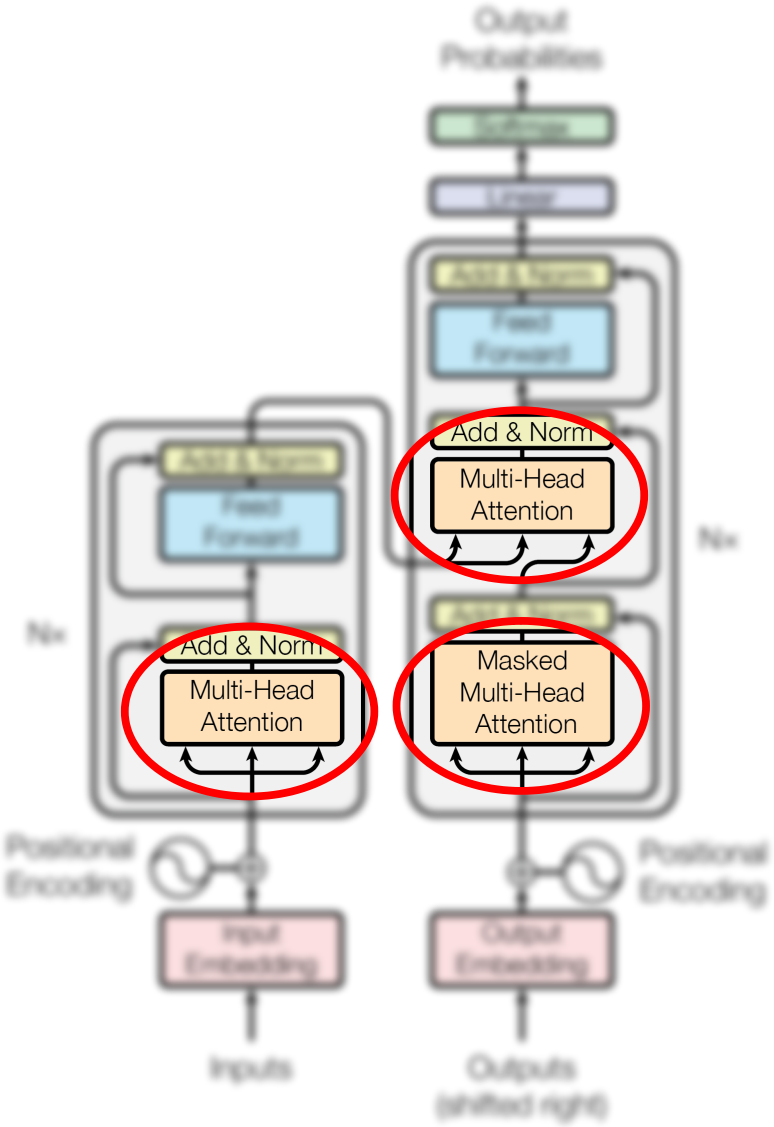


Autocomplete engines (like ChatGPT) use an attention distribution between a sentence and itself: **self-attention** instead of **cross-attention**.

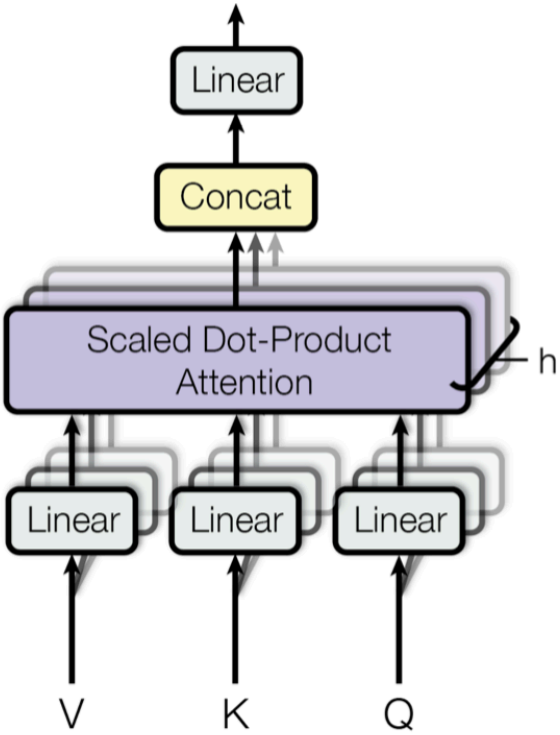
Notice that "it" maps to the two nouns in the sentence, but more strongly to "animal."



Attention mechanisms appear in three places in the transformer architecture.



"Multi-head" attention is a concatenation of N attention results.



It adds one more type of vector manipulation to the list:

1. linear transformations
2. non-linear activation functions
3. random generation from distribution parameters
4. small set of learned convolution parameters applied to large images
5. reducing an image size with pooling
6. adding arbitrarily many neural network outputs to make the next neural network input
7. concatenate vectors (make a $n_1 + n_2$-dimensional space from n_1-dimensional and n_2-dimensional spaces).

Basically, any array-oriented manipulation (that you can differentiate through, to help the optimizer) is fair game.

Closing remarks

Machine learning doesn't magically take away all the difficulty (or interestingness!) of programming: it replaces one set of issues with another.

traditional programming (craftsmanship)	machine learning (farming)
type correctness	defining loss functions
mutable state	tweaking optimizers, batch sizes
data structures, algorithms	under & overfitting
modularization, separation of concerns	regularization
API design	training, validation, testing
concurrency	deciding what is a good fit
memory management	designing the architecture
...	...

And some problems are better suited to traditional programming, while others are better suited to machine learning. With traditional programming, you can determine what the

program does exactly. With machine learning, you can grow more complex systems than a human mind could ever develop.

Final thoughts: Andrej Karpathy, *A Recipe for Training Neural Networks* (2019).