Introduction to Machine Learning for Materials Science

Martin Uhrin, Université Grenoble Alpes

Overview

Day 1

1. Review of machine learning algorithms

2. Machine learning pipline in practice

Day 2

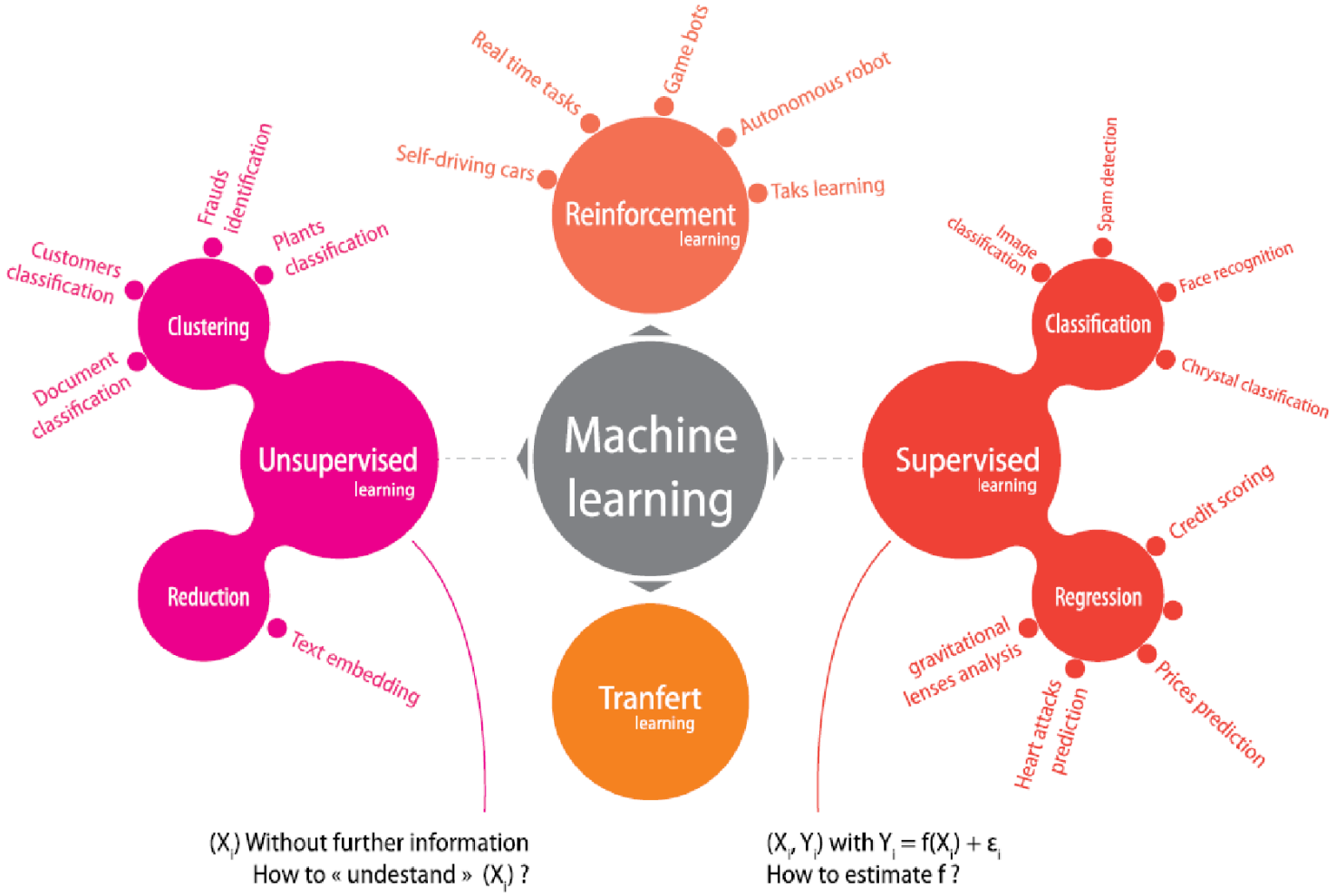
3. Development atomistic ML models

4. Examples of real research projects

Review of machine learning algorithms

- Supervised:
    - linear regression
    - polynomial regression
    - decision trees
    - neural networks
- Unsupervised
    - Dimensionality reduction
        - principle component analysis
        - autoencoders
    - Clustering
        - k-means
        - Gaussian mixture model
        - DBSCAN

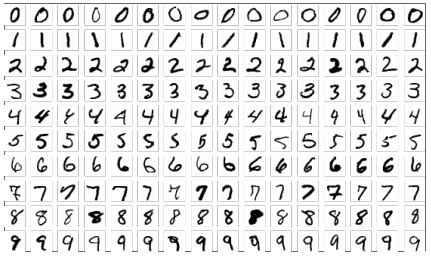
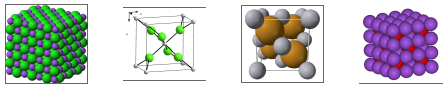
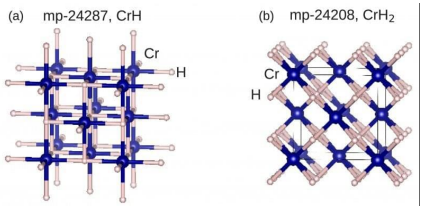
Taxonomy of ML methods



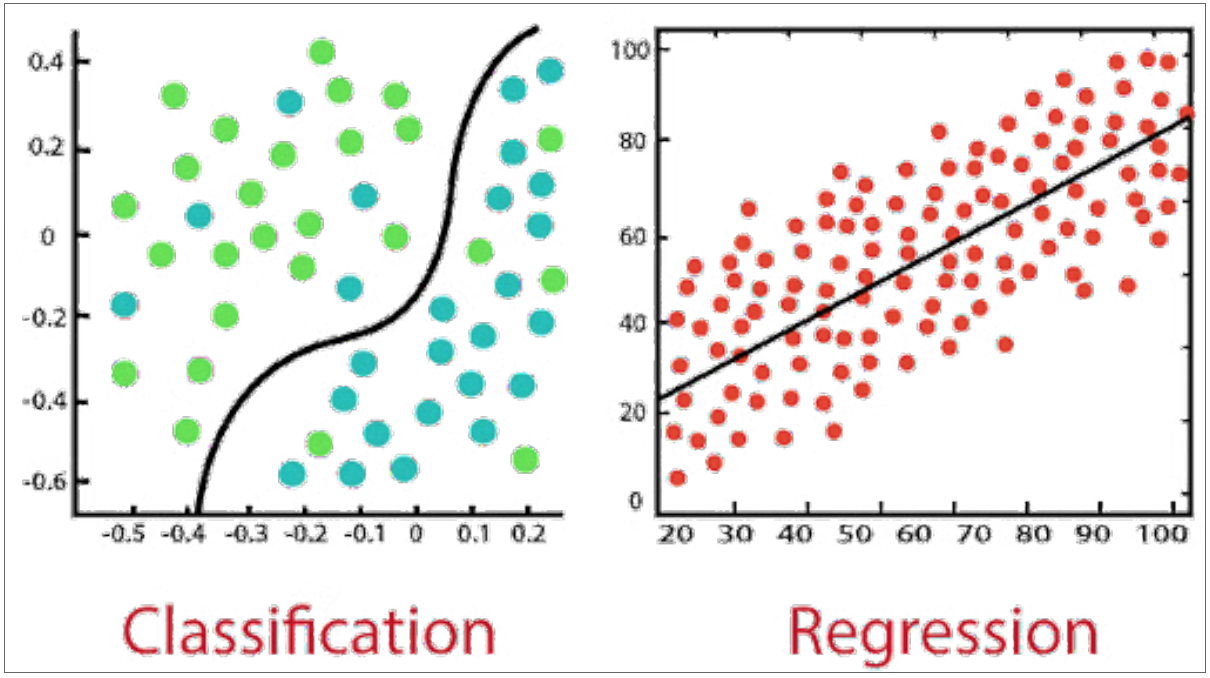
Supervised learning

Want to learn some function f that maps our inputs x to our desired outputs y (*labels*), i.e. $f : x \mapsto y$

Some examples:

Inputs	Outputs	Type
	$\{$ 0, 1 . . ., 9 $\}$	Classification
$\{$ Num. rooms, location, m^2, age, . . ., $\}$	Price (€)	Regression
	$E, \rho, B_0, \ldots$	Regression
	T_c $> 293K$	Classification
	$\{$ True, False $\}$	

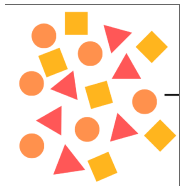
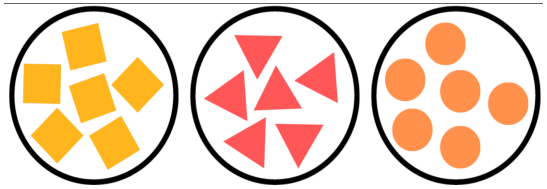
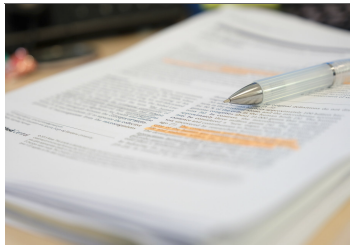
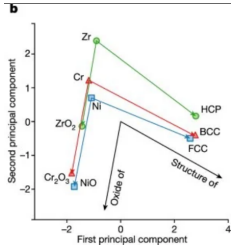
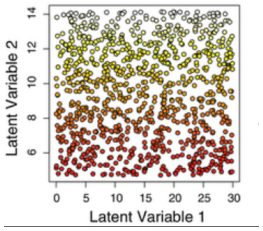
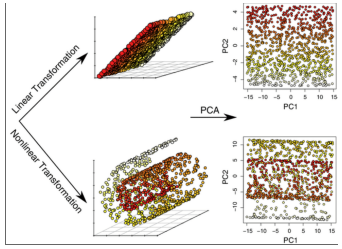
Classification vs regression



In **classification** we are trying to find which *class* a particular sample, x, belongs to. The task of the model is to estimate the *decision boundary* that best separates samples x into different classes.
In **regression** we are trying to find the value of y for a given sample x. The task of the model is to estimate the relationship between dependent variable x, and one or more independent variables y.

Unsupervised learning

In contrast to supervised learning, in *unsupervised learning* we have no labels and so our algorithm learns to find patterns exclusively from the inputs.
Some examples:

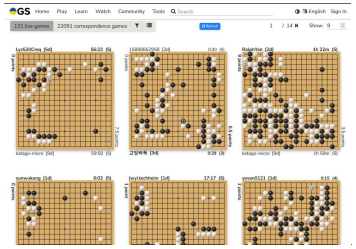
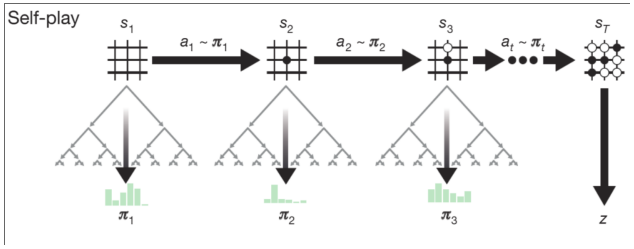
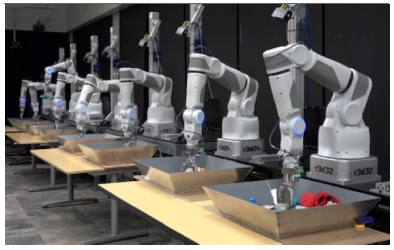
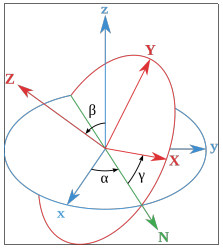
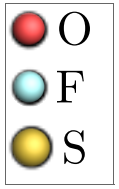
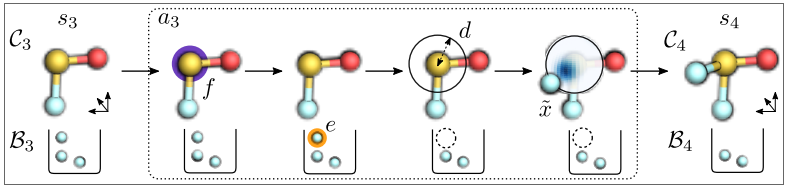
Inputs	Outputs	Type
		Clustering
		Clustering
		Dimensionality reduction

Dimensionality reduction vs clustering

In **dimensionality reduction** we are trying to find a reduced-dimensionality representation of x that retains *meaningful properties* of the original data, X. The task of the model is to find a mapping $f : x \mapsto x'$, such that there is minimal loss of information.
In **clustering** we are trying to find a a way of grouping inputs into clusters, such that members of the same cluster are more similar to eachother than they are to the other samples. The task of the model is to lean a mapping from the data to a cluster $f : x \mapsto c$.

Reinforcement learning

In reinforcement learning, an *intelligent agent* takes *actions* in a dynamic environment with the goal of maximising *cumulative reward*.
Some examples:

Inputs	Outputs
	
	
	

In **reinforcement learning** we are trying to find the actions that maximise the cumulative reward. The task of the model is to learn a probability P for selecting an action a given a current state of the environment (and possibly the state of the agent), s, i.e. $P(a|s)$. This is called the *policy*.

Supervised learning

Let's use get a dataset to motivate our discussion of supervised learning

In [2]:

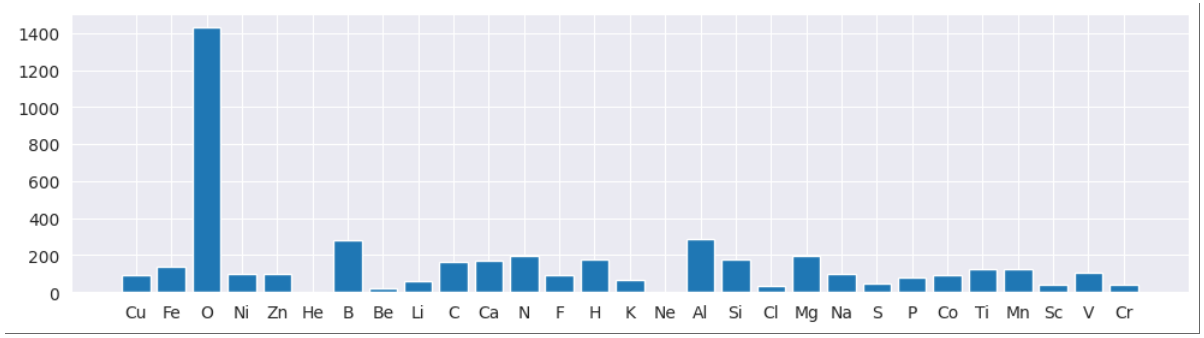
```
import pandas as pd

df = pd.read_csv('data/mpdata.csv')
df.sample(5)
```

Out[2]:

	material_id	energy	volume	nsites	energy_per_atom	pretty_formula	spacegroup	band_gap	density	total_magnetization
5044	mp-25008	-126.871112	219.480379	18	-7.048395	Ca2Mn2O5	55	0.4344	4.085958	15.999358
1142	mp-626279	-272.195849	427.137714	54	-5.040664	Ca(H8O5)2	1	3.8913	1.680989	-0.001464
5549	mp-542614	-186.507093	260.619374	24	-7.771129	VCo3	194	0.0000	8.706328	-0.003611
1054	mp-1014324	-134.237672	172.839829	16	-8.389855	C3N	74	0.0000	1.922968	-0.000011
6173	mp-625150	-60.179820	86.476445	8	-7.522477	ScHO2	36	4.4120	2.994111	0.000003

In [3]:

```
fig = plt.figure(figsize=(12, 3))
full_composition = df.dropna()["pretty_formula"].map(lambda formula: pymatgen.core.Composition(formula)).sum()
comp_dict = full_composition.as_dict()
fig.gca().bar(comp_dict.keys(), comp_dict.values());
```


1400
1200
1000
800
600
400
200
0
Cu Fe O Ni Zn He B Be Li C Ca N F H K Ne Al Si Cl Mg Na S P Co Ti Mn Sc V Cr

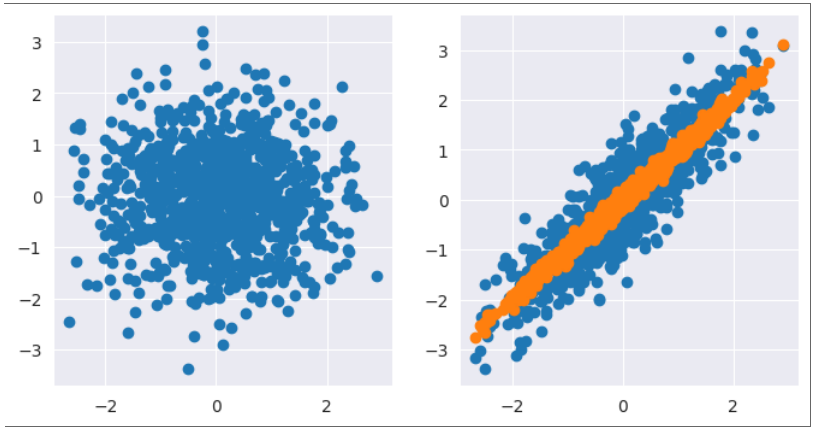
Significance of correlation

Before we begin, let's discuss *correlation* which can indicate a physical relationship between variables.
Definition: two variables are *correlated* if they share a statistical dependence or relationship.

Let's create some fake data to see one way to measure correlation

In [4]:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 2, figsize=(8, 4))
N = 1000
x = np.random.normal(size=N)  # Normally distributed random x
y = np.random.normal(size=N)  # Normally distributed random y
y2 = x + np.random.normal(scale=0.5, size=N) # x data with Gaussian noise added
y3 = x + np.random.normal(scale=0.1, size=N) # x data with less Gaussian noise added
axs[0].scatter(x, y)
axs[1].scatter(x, y2)
axs[1].scatter(x, y3);
```



How can we quantify the degree of correlation in these examples?

One useful way is the **correlation coefficient** defined as

$$R = \frac{\big\langle (x-\mu_x)(y-\mu_y) \big\rangle}{\sigma_x \sigma_y} = \frac{\langle xy \rangle - \mu_x \mu_y}{\sigma_x \sigma_y}$$

where

$$\langle xy \rangle = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} xy P(x,y) dx dy$$

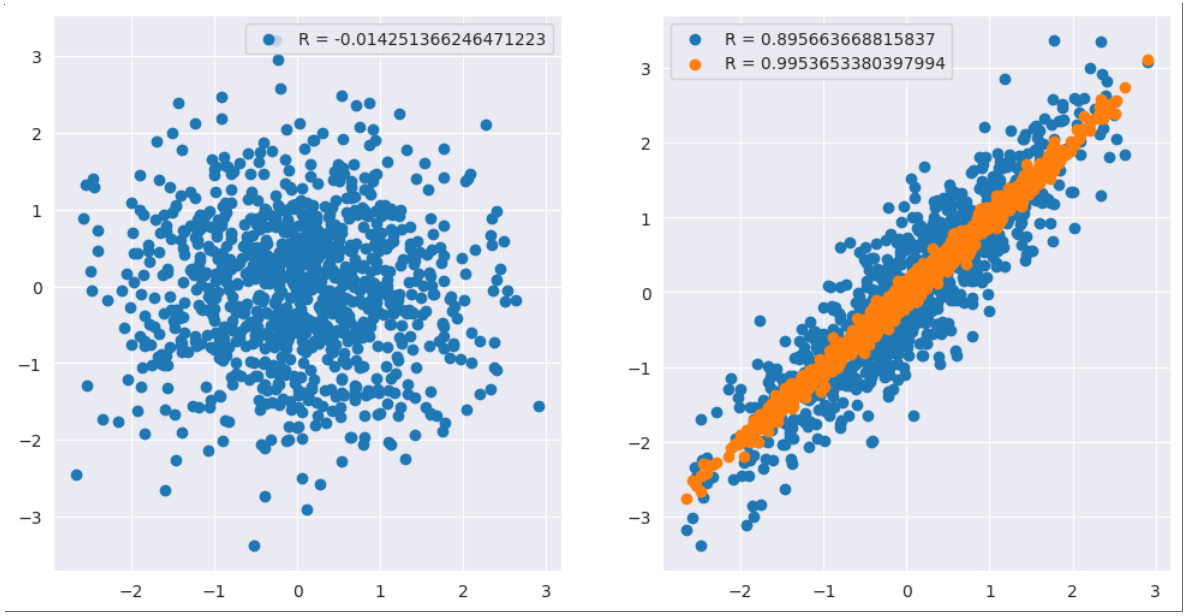
which takes on the following values:

- $R = 0$ means **no correlation**, $P(x,y)$ is separable into $f(x)g(y)$, hence $\langle xy \rangle = \langle x \rangle \langle y \rangle = \mu_x \mu_y$
- $R = +1$ means **complete correlation**, $y = Cx$
- $R = -1$ means **complete anti-correlation**, $y = -Cx$

Let's see this for our examples

In [5]:

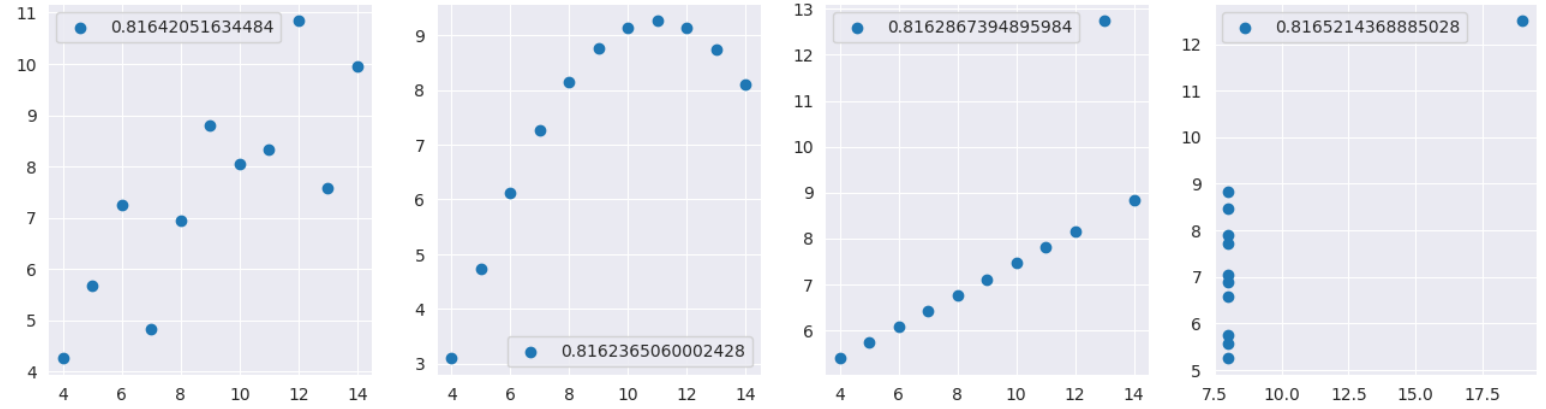
```
c = np.corrcoef(x, y)[0, 1]
c2 = np.corrcoef(x, y2)[0, 1]
c3 = np.corrcoef(x, y3)[0, 1]

fig, axs = plt.subplots(1, 2, figsize=(12, 6))
axs[0].scatter(x, y, label=f"R = {c}")
axs[1].scatter(x, y2, label=f"R = {c2}")
axs[1].scatter(x, y3, label=f"R = {c3}")
axs[0].legend();
axs[1].legend();
```



> ***Warning:*** *Be careful though, R-factor is not always enough!*

In [6]:

```
data = [
    [[10, 8.04], [8, 6.95], [13, 7.58], [9, 8.81], [11, 8.33], [14, 9.96], [6, 7.24], [4, 4.26], [12, 10.84], [7, 4.82],[5, 5.68]],
    [[10, 9.14], [8, 8.14], [13, 8.74], [9, 8.77], [11, 9.26], [14, 8.1], [6, 6.13], [4, 3.1], [12, 9.13], [7, 7.26], [5, 4.74]],
    [[10, 7.46], [8, 6.77], [13, 12.74], [9, 7.11], [11, 7.81], [14, 8.84], [6, 6.08], [4, 5.39], [12, 8.15], [7, 6.42], [5, 5.73]],
    [[8, 6.58], [8, 5.76], [8, 7.71], [8, 8.84], [8, 8.47], [8, 7.04], [8, 5.25], [19, 12.5], [8, 5.56], [8, 7.91], [8, 6.89]]]

fig, axs = plt.subplots(1, 4, figsize=(16, 4))
for dataset, ax in zip(data, axs):
    dataset = np.array(dataset)
    x, y = dataset[:, 0], dataset[:, 1]
    ax.scatter(x, y, label=np.corrcoef(x, y)[0, 1])
    ax.legend()
```



What gives?

Well, these datasets have the **same mean and variance** and, therefore the same correlation factor!

In [7]:

```
for entry in data:
    print(f"μ = {np.mean(entry)} σ = {np.std(entry)}")
```

```
μ = 8.250454545454547 σ = 2.7272421589195983
μ = 8.250454545454545 σ = 2.72727215909085
μ = 8.249999999999998 σ = 2.726979681225766
μ = 8.250454545454545 σ = 2.72690713458694
```

Now that you've been warned, we can go back and look at the correlations in our starting dataset.

In [8]:

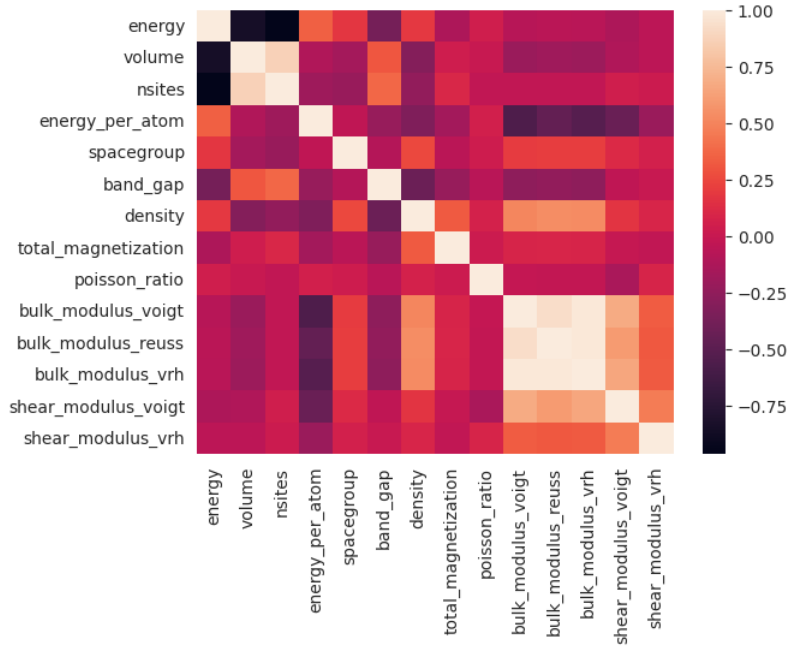
```
corr = df.corr(numeric_only=True); corr
```

Out[8]:

	energy	volume	nsites	energy_per_atom	spacegroup	band_gap	density	total_magnetization	poisson_r
energy	1.000000	-0.852020	-0.965198	0.345171	0.176241	-0.378738	0.185483	-0.121921	0.0459
volume	-0.852020	1.000000	0.862495	-0.110264	-0.162344	0.302136	-0.309343	0.034127	0.0000
nsites	-0.965198	0.862495	1.000000	-0.194668	-0.217266	0.371992	-0.248281	0.095329	-0.0328
energy_per_atom	0.345171	-0.110264	-0.194668	1.000000	-0.038482	-0.221015	-0.334075	-0.171508	0.0517
spacegroup	0.176241	-0.162344	-0.217266	-0.038482	1.000000	-0.093678	0.250417	-0.065560	0.0284
band_gap	-0.378738	0.302136	0.371992	-0.221015	-0.093678	1.000000	-0.421409	-0.220998	-0.0683
density	0.185483	-0.309343	-0.248281	-0.334075	0.250417	-0.421409	1.000000	0.322121	0.0655
total_magnetization	-0.121921	0.034127	0.095329	-0.171508	-0.065560	-0.220998	0.322121	1.000000	0.0230
poisson_ratio	0.045966	0.000059	-0.032862	0.051704	0.028444	-0.068310	0.065513	0.023007	1.0000
bulk_modulus_voigt	-0.080688	-0.208067	-0.032160	-0.565429	0.195654	-0.266141	0.501874	0.080165	-0.0194
bulk_modulus_reuss	-0.059895	-0.187330	-0.028786	-0.465262	0.205996	-0.249368	0.538017	0.089585	-0.0222
bulk_modulus_vrh	-0.071393	-0.201073	-0.030992	-0.523804	0.204483	-0.262229	0.529485	0.086458	-0.0212
shear_modulus_voigt	-0.128289	-0.110852	0.042327	-0.434698	0.113808	-0.036694	0.165460	-0.006023	-0.1434
shear_modulus_vrh	-0.056726	-0.052474	0.017493	-0.211633	0.058407	0.000743	0.086553	-0.031027	0.0826

In [9]:

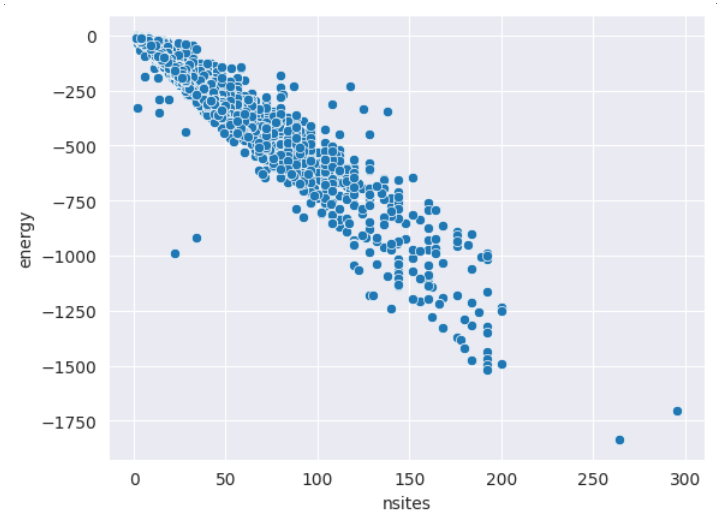
```
import seaborn as sns

sns.heatmap(corr);
```


energy
volume
nsites
energy_per_atom
spacegroup
band_gap
density
total_magnetization
poisson_ratio
bulk_modulus_voigt
bulk_modulus_reuss
bulk_modulus_vrh
shear_modulus_voigt
shear_modulus_vrh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

Let's have a quick look at the correlation between number of atms and total energy

In [10]:

```
sns.scatterplot(data=df, x="nsites", y="energy");
```



Regression

Linear regression

Suppose we assume that our data can be modelled as

$$y_i = \beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip} + \epsilon_i = \boldsymbol{x}_i^T \boldsymbol{\beta} + \epsilon_i$$

where x_{ip} are our independent samples (*regressors*), y_i are the observed values (*regressands*), and β_p are the fitting parameters and ϵ_i are the errors. We can write this in matrix notation as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

The equation is nothing other than a Taylor expansion of our function to first order.
But how do get get β?

Often, we are interested in the solution that minimises the sum of the square of the errors i.e.
$\beta_p = \text{argmin}_{\beta_p} \sum_i (\beta_p x_{ip} - y_i)^2$, thus, we want to minimise the loss $\mathcal{L}(\boldsymbol{\beta}) = \|\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{Y}\|^2$

$$
\begin{aligned}
&= (\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{Y})^T (\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{Y}) \\
&= \boldsymbol{Y}\boldsymbol{Y}^T - \boldsymbol{Y}^T \boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{\beta}^T \boldsymbol{X}^T \boldsymbol{Y} + \boldsymbol{\beta}\boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\beta}
\end{aligned}
$$

taking the derivative we get

$$
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\beta}} = -2\boldsymbol{X}^T \boldsymbol{Y} + 2\boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\beta}\,,
$$

finally, setting this to zero, we get the solution

$$
\boldsymbol{\beta} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{Y}.
$$

This is one of the few methods with a closed form solution, and makes a good starting point for many ML problems. Let's try this with scipy:

In [11]:

```
from sklearn import linear_model

X = df[['nsites']]
y = df['energy']

linear = linear_model.LinearRegression().fit(X, y)
```

Which looks like:

In [12]:

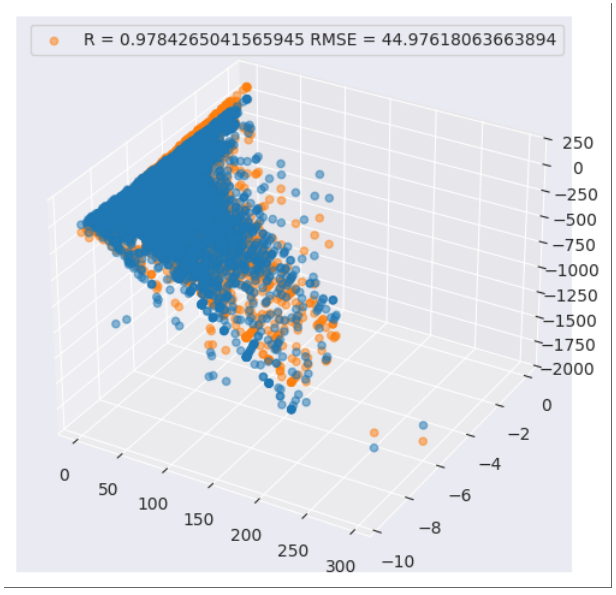
```
fig, ax = plt.subplots(figsize=(6, 6))
sns.scatterplot(data=df, x="nsites", y="energy", ax=ax)
y_prime = linear.predict(X)
ax.plot(X, y_prime, label=f"R = {np.corrcoef(y, y_prime)[0, 1]}, RMSE = {np.sqrt(((y - y_prime)**2).mean())}");
ax.legend();
```

MULTILINEAR FITTING

We can also use multiple input variables in which case X has mulitple columns:

In [13]:

```
X = df[['nsites', 'energy_per_atom']]  # Let's 'cheat' and use the energy_per_atom as well
y = df['energy']
linear2 = linear_model.LinearRegression().fit(X, y)

fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(projection='3d')
ax.scatter(X['nsites'], X['energy_per_atom'], y, alpha=0.5)

y_prime = linear2.predict(X)
ax.scatter(X['nsites'], X['energy_per_atom'], y_prime, alpha=0.5, label=f"R = {np.corrcoef(y, y_prime)[0, 1]} RMSE = {np.sqrt(((y - y_prime
ax.legend();
```



Advantages

- Fast
- Interpretable
- Predicatble extrapolation
- Can be easy to add uncertainty estimation

Disadvantages

- Many (most!) problems of interest are not linear

Polynomial regression

We need not stop our fit at first order, we can continue to arbitrary degree:

$$y_1 = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \ldots + \beta_d x_1^d + \epsilon_1$$

Or for n samples:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ 1 & x_3 & x_3^2 & \dots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix},$$

As before, we can solve for $\boldsymbol{\beta}$ using

$$\vec{\widehat{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\,\mathbf{X}^\mathsf{T}\vec{y},$$

Let's try this out with an example

In [14]:

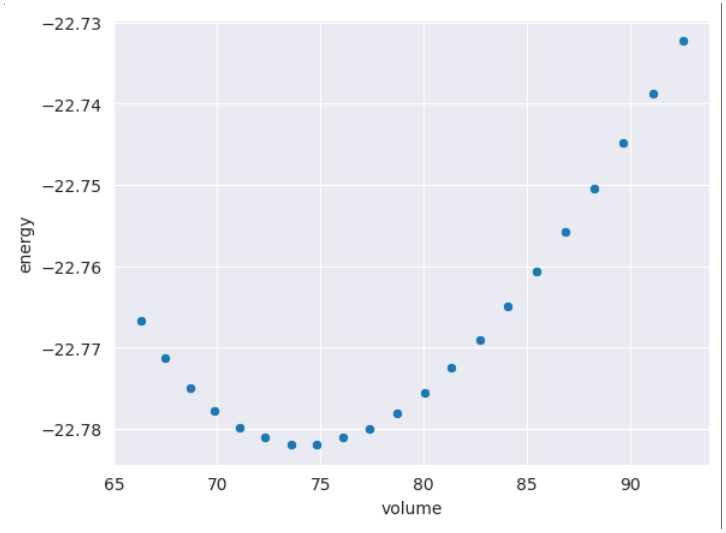
```
diamond = ase.io.read("data/C_diamond_00.out")
ase.visualize.view(diamond * 5, viewer='ngl')
```

Out[14]:

```
HBox(children=(NGLWidget(), VBox(children=(Dropdown(description='Show', options=('All', 'C'), value='All'), Dr…
```

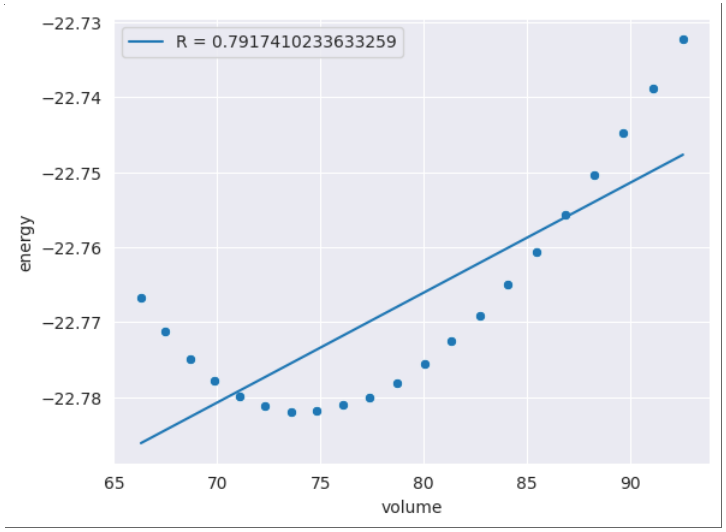
In [15]:

```
eos = pd.read_csv("data/etot_v_vol.dat", delimiter=r"\s+")
ax = sns.scatterplot(data=eos, x="volume", y="energy")
```



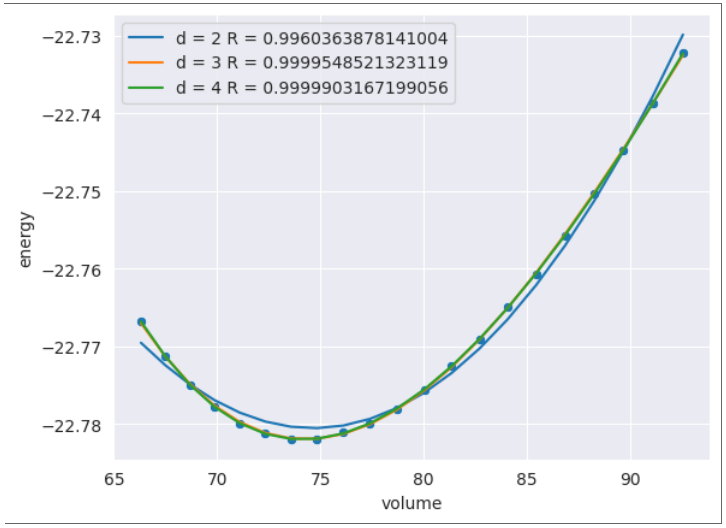
In [16]:

```
X = eos[['volume']]
y = eos['energy']
model = linear_model.LinearRegression().fit(X, y)

ax = sns.scatterplot(data=eos, x="volume", y="energy")
ax.plot(X, model.predict(X), label=f"R = {np.corrcoef(y, model.predict(X))[0, 1]}");
ax.legend();
```



In [17]:

```
from sklearn import preprocessing

ax = sns.scatterplot(data=eos, x="volume", y="energy")

for degree in range(2, 5):
    X_prime = preprocessing.PolynomialFeatures(degree=degree).fit_transform(X)
    model = linear_model.LinearRegression().fit(X_prime, y)
    ax.plot(X, model.predict(X_prime), label=f"d = {degree} R = {np.corrcoef(y, model.predict(X_prime))[0, 1]}");
ax.legend();
```



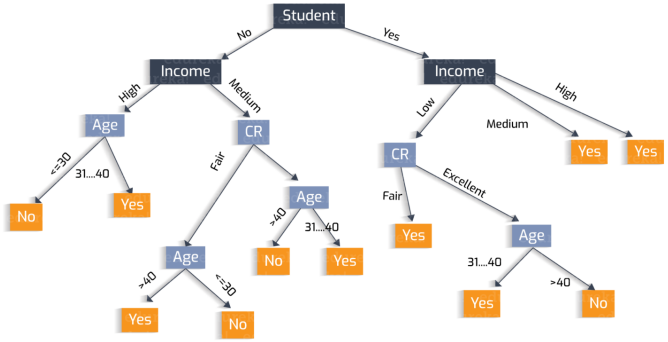
Advantages

- Similar advantages to linear models
- Works well on nonlinear problems

Disadvantages

- Need choose the *right* polynomial degree:
    - This influences the bias/variance tradeoff
    - Different degrees of (non-orthogonal) polynomials are coupled

Decision trees



There are a number of algorithms that we can use to create such a tree, here I give an outilne of the Classification and Regression Trees (CART) method:

1. We start with all of the data in a single node (the root)
2. Find the feature that, once split, will minimise the variability of the data within a node.
3. Repeat recursively until max depth reached.
4. Prune the tree by identifying branches that can be removed without significant loss in accuracy.

OK, but how do we know which feature will lead to a split that minimises variability?

The overall goal is to minimise the loss $\mathcal{L} = \sum_i |f(x_i) - y_i|^2$.
For each node $t \in T$, we can define the average y-value by

$$\bar{y} = \frac{1}{N(t)} \sum_{x_i \in t} y_i.$$

We can now define the (squared) error rate of node t by

$$r(t) = \frac{1}{N(t)} \sum_{x_i \in t} (y_i - \bar{y}(t))^2,$$

and the cost of node t by R(t) = r(t) p(t), where $p(t) = \frac{N(t)}{N}$.

Therefore

$$R(t) = \frac{1}{N} \sum_{x_i \in t} (y_i - \bar{y}(t))^2.$$

Now, let's suppose we split t into t_L and t_R,

$$R(t) \geq R(t_L) + R(t_R).$$

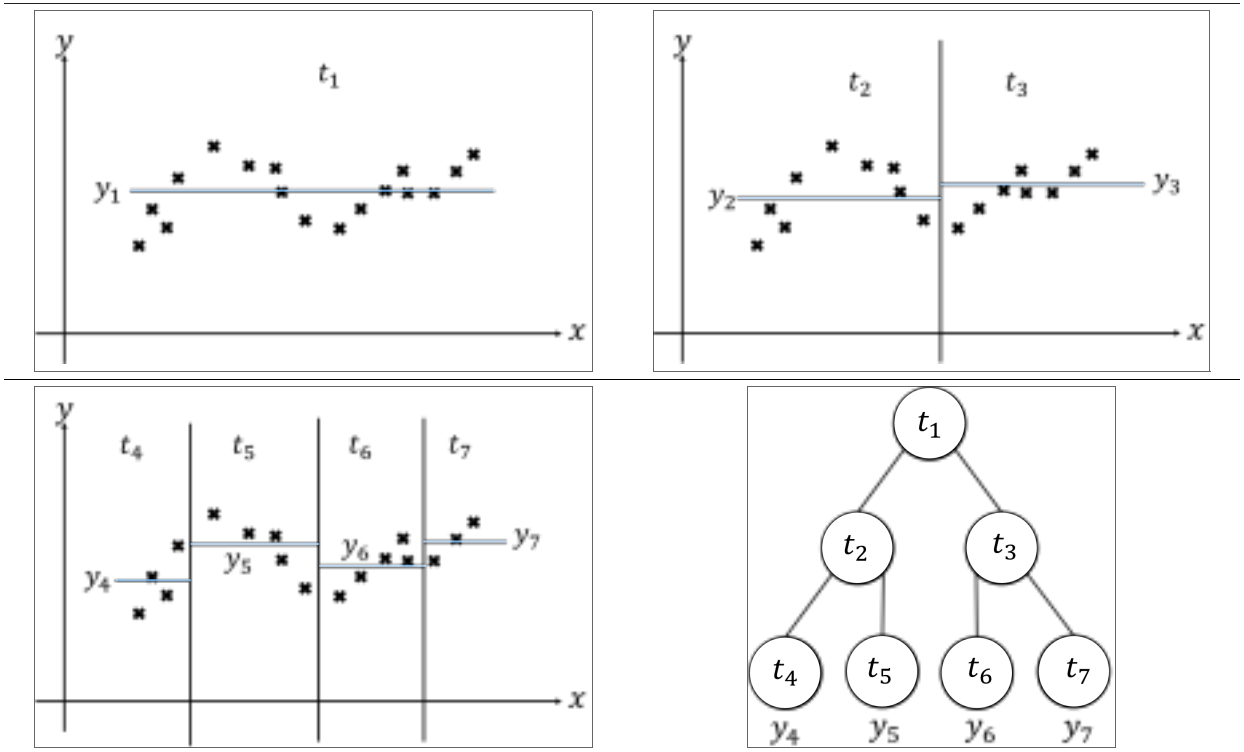
The equality holds *if and only if* $\overline{y}(t) = \overline{y}(t_L) = \overline{y}(t_R)$. For a particular split s, we can define the decrease in cost as:

$$\Delta R(s, t) = R(t) - R(t_L) - R(t_R).$$

Now, we accept the split, s^*, among all candidate splits that decreases the most:

$$\Delta R(s^*, t) = \max_s \Delta R(s, t).$$

The tree is grown to some $T_{\max}$ (e.g. until we reach some minimum number of elements in a node). Then we can prune back.
See **here** for a full derivation.
So, how does this look?



Let's try this out in code!

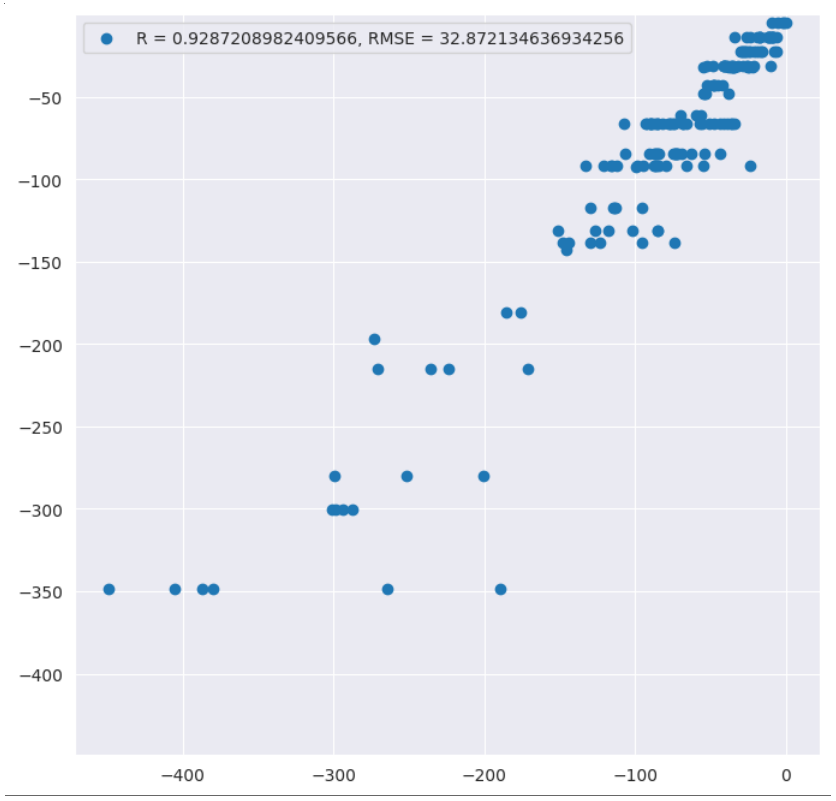
In [18]:

```
from sklearn import model_selection

numerical = df.select_dtypes(include='number').dropna()

X = numerical[numerical.columns.drop(['energy', 'energy_per_atom'])]
y = numerical['energy']

# Keep back 20% of the data for evaluating the model performance
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.2, random_state=0)
```

Let's create a model, train and then make a prediction on the test set

In [19]:

```
from sklearn import tree

# Create our regressor
dectree = tree.DecisionTreeRegressor(max_depth=5).fit(X_train, y_train)
fig, ax = plt.subplots(figsize=(8, 8))

# Make a prediction on the (held back) test set
y_tree = dectree.predict(X_test)
ax.scatter(y_test, y_tree,
    label=f"R = {np.corrcoef(y_test, y_tree)[0, 1]}, RMSE = {np.sqrt(((y_test - y_tree)**2).mean())}"
)
ax.set_ylim((y_test.min(), y_test.max()))
ax.legend();
```



Interpretability

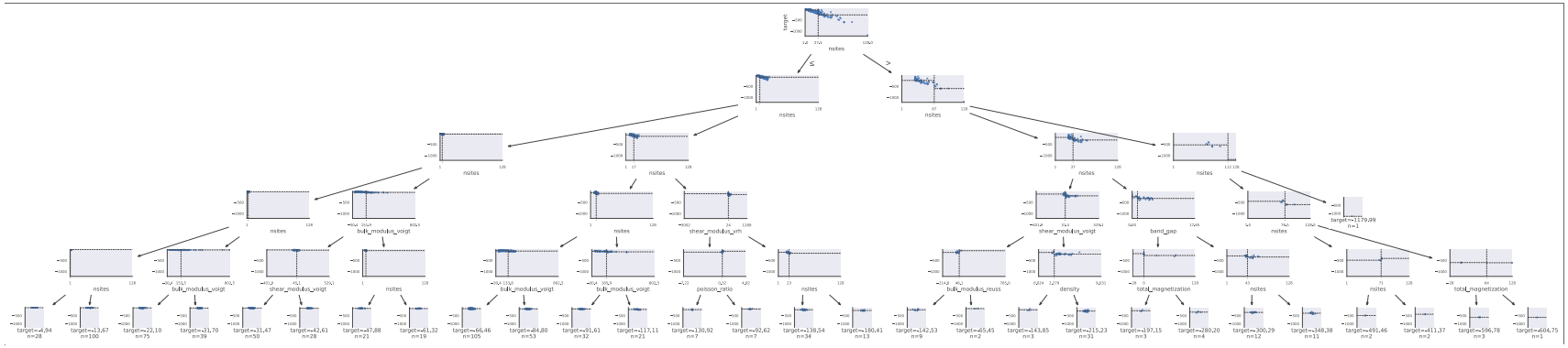
One of the nice things about decision trees is that they are interpretable, as we can introspect the boundaries learned during training

In [20]:

```
import dtreeviz

viz = dtreeviz.model(dectree, X_train, y_train, target_name="target", feature_names=X_train.columns)
viz.view()
```

```
/home/martin/.local/miniconda3/envs/milad/lib/python3.10/site-packages/sklearn/base.py:493: UserWarning: X does not
have valid feature names, but DecisionTreeRegressor was fitted with feature names
```

Out[20]:



Random forest

Random forest build on decision trees by training multiple tree models, each with a different subset of the training data. The idea is to combine the learning power of multiple models into one, and thereby reduce sensitivity to noise, as well as reducing the change of overfitting.

Let's give it a shot with our data, and see how it does

In [21]:

```
from sklearn import ensemble

# Create our regressor
randforest = ensemble.RandomForestRegressor(n_estimators=50, max_depth=5).fit(X_train, y_train)
fig, ax = plt.subplots(figsize=(8, 8))

# Make a prediction on the (held back) test set
y_forest = randforest.predict(X_test)

ax.scatter(
    y_test, y_tree, label=f"Tree R = {np.corrcoef(y_test, y_tree)[0, 1]}, RMSE = {np.sqrt(((y_test - y_tree)**2).mean())}")
ax.scatter(
    y_test, y_forest, label=f"Forest R = {np.corrcoef(y_test, y_forest)[0, 1]}, RMSE = {np.sqrt(((y_test - y_forest)**2).mean())}")

ax.set_ylim((y_test.min(), y_test.max()))
ax.legend();
```

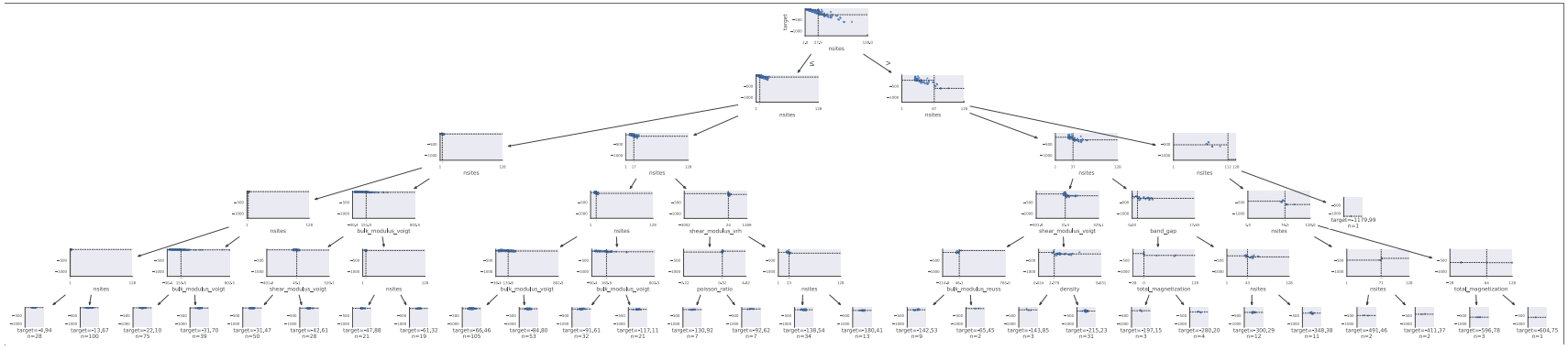

Tree R = 0.9287208982409566, RMSE = 32.872134636934256
Forest R = 0.9170405277756457, RMSE = 36.306525031516024
−50
−100
−150
−200
−250
−300
−350
−400
−400
−300
−200
−100
0

In [22]:

```
import dtreeviz

viz = dtreeviz.model(dectree, X_train, y_train, target_name="target", feature_names=X_train.columns)
viz.view()
```

```
/home/martin/.local/miniconda3/envs/milad/lib/python3.10/site-packages/sklearn/base.py:493: UserWarning: X does not
have valid feature names, but DecisionTreeRegressor was fitted with feature names
```

Out[22]:



Advantages

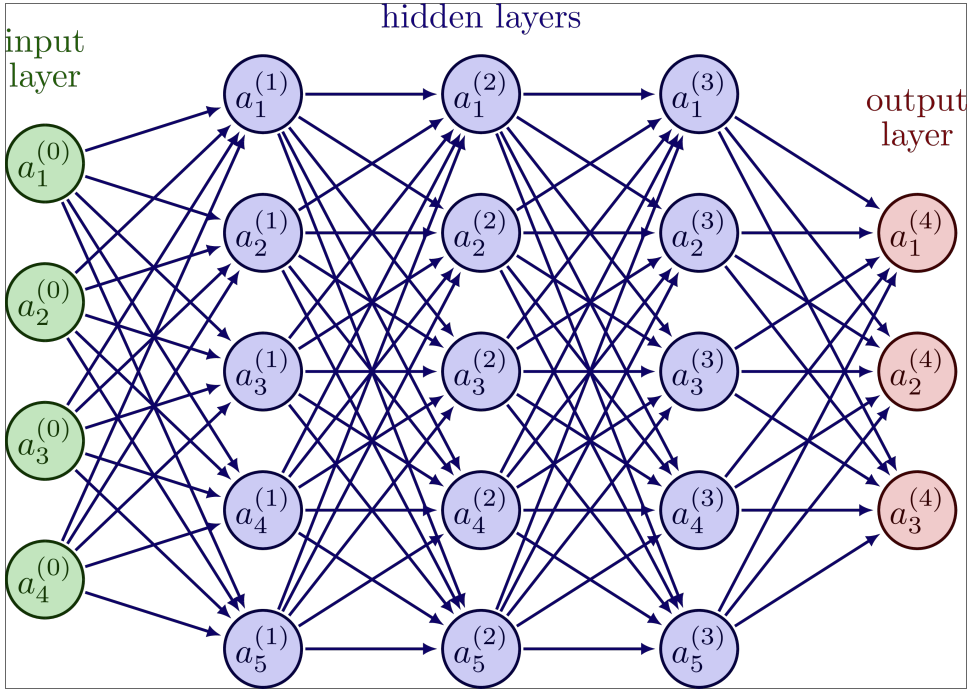
- Lightweight
- (Decision tree) Simplicity and interpretability
- Versatility (no need for feature scaling, can handle non-linear relationships)
- (Random forest) Resistant to overfitting
- (Random forest) Resistant to noise

Disadvantages

- (Decison tree) poor rsults on small datasets
- (Decision tree) overfitting
- Instability (small variations can produce completely different tree)
- Piecewise-linear (and therefore has many places where derivatives are not defined)
- (Random forest) poor interpretability

Neural networks

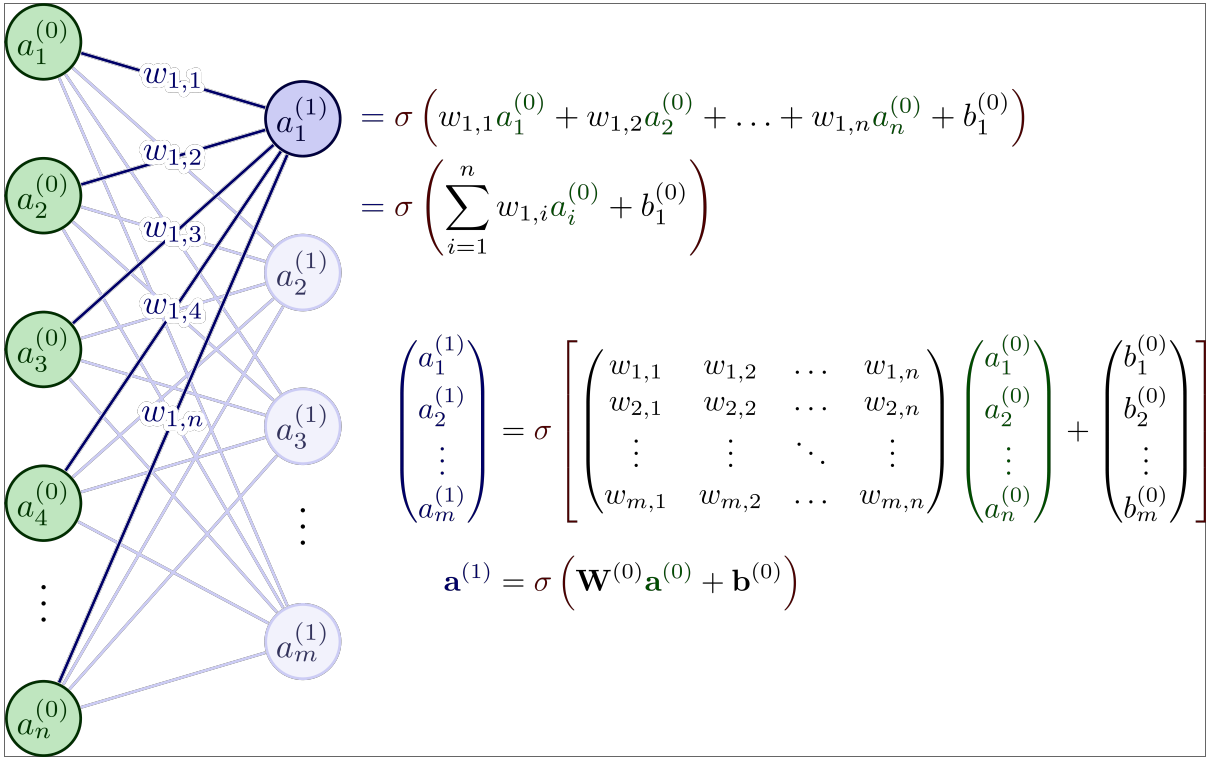
We've all heard about them, but how to they work?



where $a_i^{(l)}$ is the i^{th} *neuron* the l^{th} layer.

Propagation

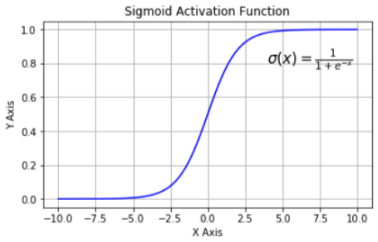
The procedure for (forward) propagating a value through the network is as follows

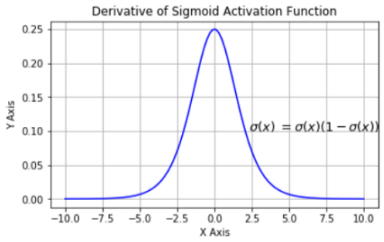


where $w_{j,i}$ is a weight that connects neuron i in the previous layer, with output j in the current layer, σ is the activation function.

Activation functions

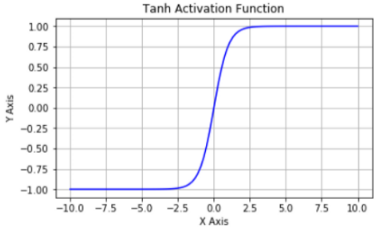
Come in many forms, some examples

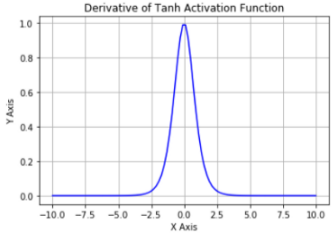




Sigmoid

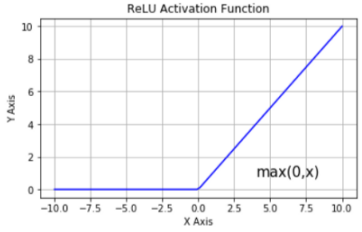
- Vanishing gradients
- Not zero centered

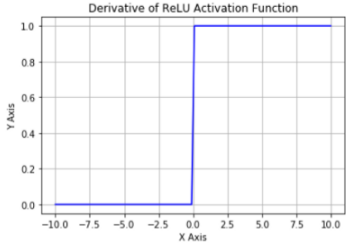




Tanh

- Vanishing gradients





ReLU

- Not zero centered

For problems in the physical sciences we want *smoothness*, both in the activation (at the very least) the first derivative! So ReLU is almost always a bad choice, if needed go for SiLU. We may also want to consider the symmetry of our features (more on this later).

Training

To *train* a neural network, we need to decide on the *goal* that is defined by the loss function, $\mathcal{L}$. The choice of $\mathcal{L}$ will depend on the learning task, and can be quite complicated.
Let's say we want to minimise our mean-square error ($\|\epsilon\|^2$), unlike linear and polynomial regression, there is **no closed-form solution** for $\boldsymbol{W}$ so we have to resort to numerical approaches.

Optimisation

Training is usually done using **optimisation** of the parameters such that the difference between predicted outputs and target outputs is minimised. The most common method for doing this is called **backpropagation**, which involves taking the derivatives of the loss with respect to the inputs by propagating the gradients through the layers of the network using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{j,i}} = \frac{\partial \mathcal{L}}{\partial a_i}\frac{\partial a_i}{\partial w_{j,i}} = \frac{\partial \mathcal{L}}{\partial a_i}\frac{\partial a_i}{\partial \mathrm{net}_i}\frac{\partial \mathrm{net}_i}{\partial w_{j,i}}$$

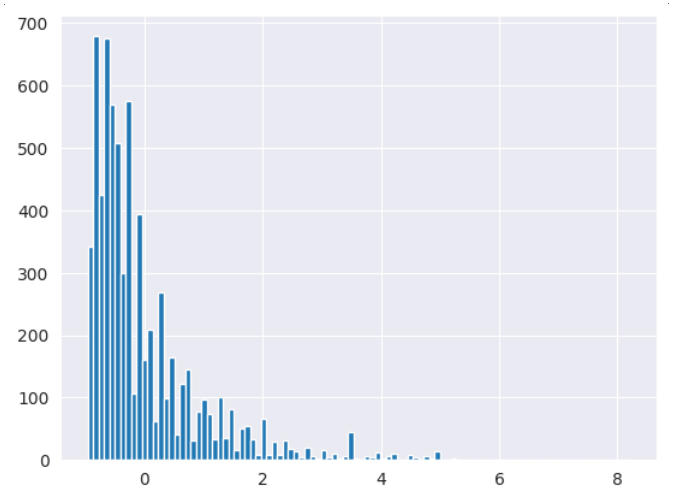
With the gradients in hand, we can update the weights by changing their values in the *direction* that lowers the loss. You will see this in almost all training loops:

- Calculate the loss
- Backpropagate gradients to get derivatives of loss wrt weights
- Use optimiser to update weights

There are *many* different optimisers that use various schemes to update the weights in an attempt to find a minimum more quickly.
Let's give it a shot!

In [23]:

```
from sklearn import preprocessing
from sklearn import neural_network

df_sorted = df.sort_values('nsites')
X = df_sorted[['nsites']]
y = df_sorted['energy']

# Use sklearns standard scalar to transform the data to mu = 0, sigma = 1
scaler = preprocessing.StandardScaler().fit(X)
X = scaler.transform(X)

plt.hist(X, bins=100);
```



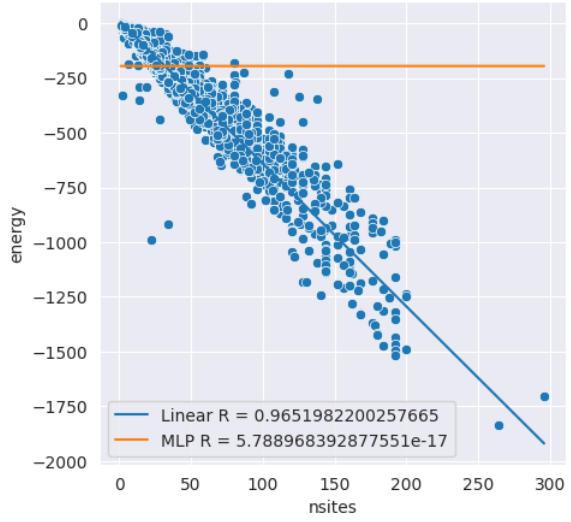
In [24]:

```
mlp = neural_network.MLPRegressor(hidden_layer_sizes=(2,), max_iter=10000).fit(X, y)
```

In [25]:

```
fig, ax = plt.subplots(figsize=(5, 5))
sns.scatterplot(data=df, x="nsites", y="energy", ax=ax)

# Linear
X_unscaled = scaler.inverse_transform(X)
y_prime = linear.predict(X_unscaled)
ax.plot(X_unscaled, y_prime, label=f"Linear R = {np.corrcoef(y, y_prime)[0, 1]}");

# MLP
y_prime = mlp.predict(X)
ax.plot(scaler.inverse_transform(X), y_prime, label=f"MLP R = {np.corrcoef(y, y_prime)[0, 1]}");
ax.legend();
```

```
/home/martin/.local/miniconda3/envs/milad/lib/python3.10/site-packages/sklearn/base.py:493: UserWarning: X does not
have valid feature names, but LinearRegression was fitted with feature names
```



Clearly, this is too easy!

Let's try something more complicated. Can we predict the energy of a material just from its composition?

First, we have to encode the composition in numerical form, to do this, we will use a vector where each entry correspond to the number of those elements found in the structure e.g. with the following encoding:

$$[N_{\mathrm{H}}, N_{\mathrm{He}}, N_{\mathrm{Li}}, N_{\mathrm{Be}}, N_{\mathrm{B}}, N_{\mathrm{C}}, N_{\mathrm{N}}, N_{\mathrm{O}}, \ldots, N_{\mathrm{Lr}}],$$

a structure with the composition Be_2O_6 would become

$$[0, 0, 0, 2, 0, 0, 0, 6, \ldots, 0],$$

Naturally, our vector typically only includes the elements that we find in our dataset, to keep the feature-dimensionality to a minimum.

In [26]:

```
valid = df.loc[df.pretty_formula.dropna().index]

all_elements = list(set().union(
    *valid["pretty_formula"].map(lambda formula: set(pymatgen.core.Composition(formula).elements)).tolist()
))


def create_one_hot(comp):
    arr = np.zeros(len(all_elements))
    for entry in comp.items():
        arr[all_elements.index(entry[0])] = entry[1]
    return arr
```

In [27]:

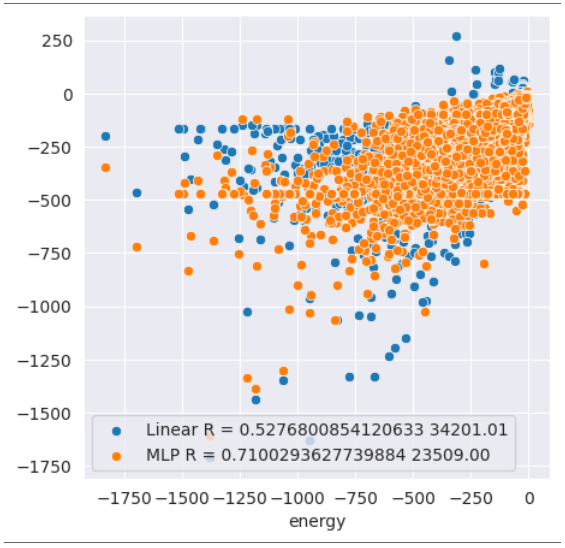
```
X = np.stack(
    valid['pretty_formula'].map(lambda formula: create_one_hot(pymatgen.core.Composition(formula))).tolist()
)
y = valid['energy']
```

In [28]:

```
linear = linear_model.LinearRegression().fit(X, y)
mlp = neural_network.MLPRegressor(hidden_layer_sizes=(20, 20, 20), max_iter=5000).fit(X, y)
```

In [29]:

```
fig, axs = plt.subplots(figsize=(5, 5))

y_prime = linear.predict(X)
ax = sns.scatterplot(x=y, y=y_prime, label=f"Linear R = {np.corrcoef(y, y_prime)[0, 1]} {metrics.mean_squared_error(y, y_prime):.2f}");

y_prime = mlp.predict(X)
ax = sns.scatterplot(x=y, y=y_prime, label=f"MLP R = {np.corrcoef(y, y_prime)[0, 1]} {metrics.mean_squared_error(y, y_prime):.2f}");

ax.legend();
```



Advantages

- Universal approximator (if we get a bad fit we can always add more layers!)
- Can be smooth, including derivatives to arbitrary order, C^n, this is very appealing for physical systems

Disadvantages

- Choice of hyperparameters not obvious, and often not transferable:
    - activation function
    - number of layers
    - number of neurons
    - etc
- Training can be difficult (many minima)

Unsupervised learning

> ***Reminder:*** *In unsupervised learning, we don't have any labels, rather we are looking for patterns and correlations in the data itself.*

Dimensionality reduction

Why do we need dimensionality reduction?
There are a few reasons:

- Reduce computational cost
- Control overfitting
- Visualisation of high-dimensional datasets
- Find correlations between features

Starting with our data, X,

In [30]:

```
df.select_dtypes(include='number').dropna()
```

Out[30]:

	energy	volume	nsites	energy_per_atom	spacegroup	band_gap	density	total_magnetization	poisson_ratio	bulk_modulus_voigt
0	-4.064600	11.852765	1	-4.064600	229	0.0000	8.902616	-7.000000e-07	0.437367	145.872296
1	-16.382096	47.264158	4	-4.095524	194	0.0000	8.930286	1.445000e-04	0.346651	141.896404
6	-4.099292	11.871896	1	-4.099292	225	0.0000	8.888270	-5.000000e-07	0.365698	139.375448
10	-48.167894	93.952272	8	-6.020987	194	0.9379	5.351412	9.971198e+00	0.343140	154.147190
12	-86.740306	153.399078	14	-6.195736	227	0.0000	5.179389	1.800474e+01	0.364961	159.130023
...	...	...	...	...	...	...	...	...	...	...
6923	-12.469544	60.517796	4	-3.117386	221	0.0000	6.697660	-1.966000e-04	0.384162	95.879213
6924	-8.834373	13.925765	1	-8.834373	225	0.0000	6.074377	1.225200e-03	0.547637	178.889958
6925	-9.086626	13.399459	1	-9.086626	229	0.0000	6.312967	5.500000e-06	0.419579	179.460907
6926	-13.066262	56.262728	4	-3.266565	221	0.0000	7.294935	-2.286000e-04	0.256895	112.623621
6927	-43.116667	126.401825	9	-4.790741	139	0.0000	6.973257	-1.325400e-03	0.306206	125.414420

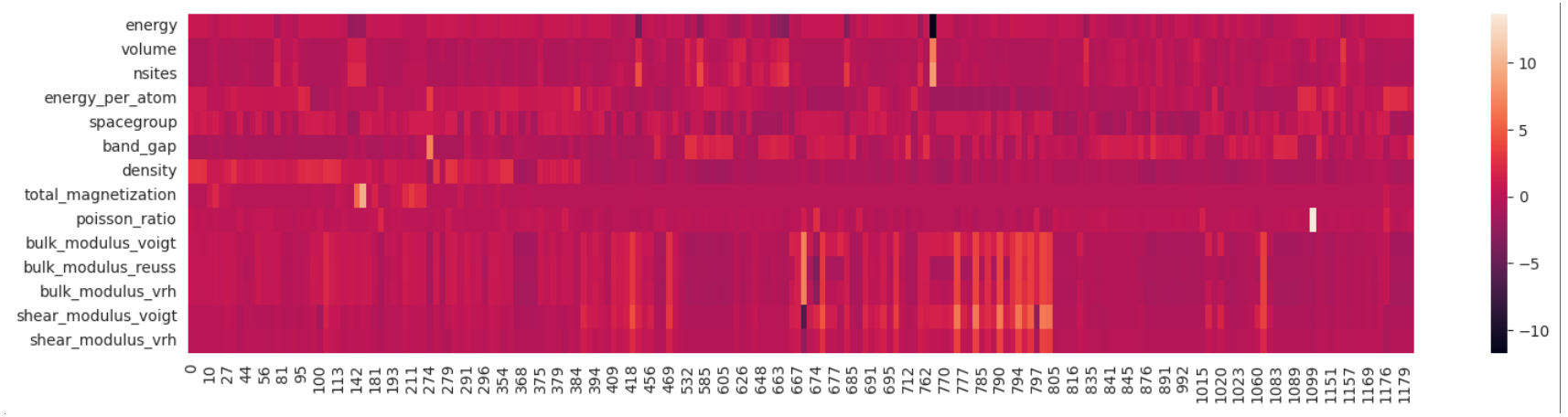
896 rows × 14 columns

In [31]:

```
# Normalise our values
numerical = df.select_dtypes(include='number').dropna()
normalised = (numerical - numerical.mean()) / numerical.std()
fig, ax = plt.subplots(figsize=(18, 4))
sns.heatmap(data=normalised.head(200).T, ax=ax)
```

Out[31]:

```
<Axes: >
```


energy
volume
nsites
energy_per_atom
spacegroup
band_gap
density
total_magnetization
poisson_ratio
bulk_modulus_voigt
bulk_modulus_reuss
bulk_modulus_vrh
shear_modulus_voigt
shear_modulus_vrh
0
10
27
44
56
81
95
100
113
142
181
193
211
274
279
291
296
354
368
375
379
384
394
409
418
456
469
532
585
605
626
648
663
667
674
677
685
691
695
712
762
770
777
785
790
794
797
805
816
835
841
845
876
891
992
1015
1020
1023
1060
1083
1089
1099
1151
1157
1169
1176
1179
10
5
0
−5
−10

Principal Component Analysis

PCA is a **linear** dimensional reduction technique. The first *principle component* is the linear correlation of our inputs that explains the most variance, the second is is what is left after removing the first component, and so on.
In general, we will perform linear transformation of our features, $t_{k(i)} = \boldsymbol{x}_{(i)}\boldsymbol{w}_{(k)}$. Starting with the first component, we want to find $\boldsymbol{w}_{(1)}$ that maximise variance, subject to the constraint that the norm of $\boldsymbol{w}$ is 1:

$$\begin{aligned}
\boldsymbol{w}_{(1)} &= \underset{\|\boldsymbol{w}\|=1}{\operatorname{argmax}} \sum_i (t_1)^2_{(i)} = \underset{\|\boldsymbol{w}\|=1}{\operatorname{argmax}} \sum_i (x_{(i)} \cdot \boldsymbol{w})^2 && (1)\\
&= \operatorname{argmax}_{\|\boldsymbol{w}\|=1} \left(\|\boldsymbol{X}\boldsymbol{w}\|^2 \right) = \operatorname{argmax}_{\|\boldsymbol{w}\|=1} \left(\boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} \right) && (2)\\
&= \underset{\|\boldsymbol{w}\|=1}{\operatorname{argmax}} \left(\frac{\boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w}}{\boldsymbol{w}^T \boldsymbol{w}} \right) && (3)
\end{aligned}$$

The last equation is a Rayleigh quotient which is maximised when $\boldsymbol{w}$ is equal to the eigenvector of $\boldsymbol{X}^T\boldsymbol{X}$ corresponding to the highest eigenvalue

We can now subtract the the first component

$$\hat{\boldsymbol{X}}_2 = \boldsymbol{X} - \boldsymbol{X}\boldsymbol{w}_{(1)}\boldsymbol{w}_{(1)}^T,$$

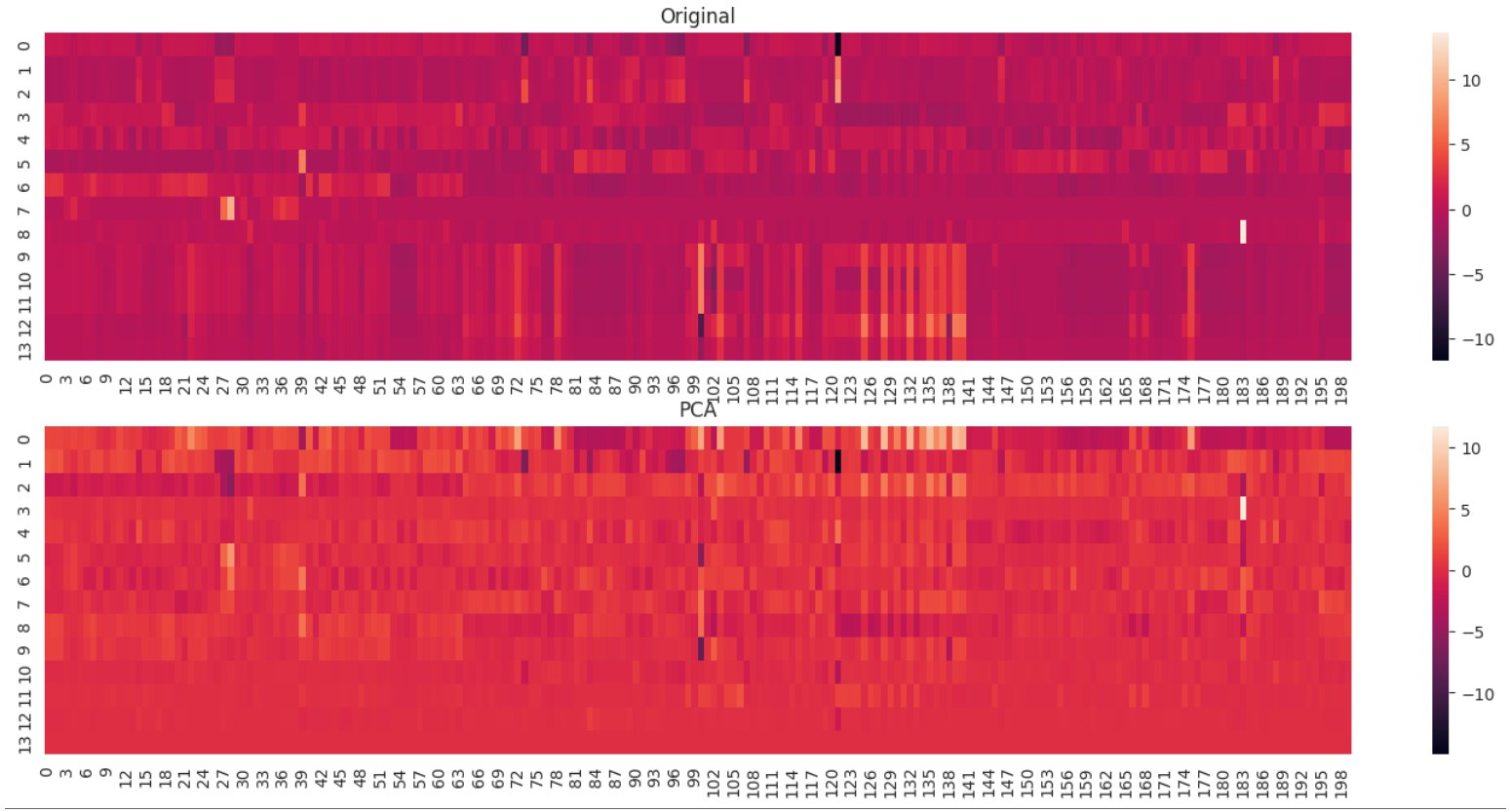
and repeat the procedure to find the second, and so on, for all k.
Let's have a look at this in action.

In [32]:

```
from sklearn import decomposition

X = normalised.values
pca = decomposition.PCA()
pca.fit(X)
X_prime = pca.transform(X)

fig, axs = plt.subplots(2, 1, figsize=(18, 8))

sns.heatmap(data=X[:200].T, ax=axs[0])
axs[0].set_title("Original")
sns.heatmap(data=X_prime[:200].T, ax=axs[1])
axs[1].set_title("PCA")
```

Out[32]:

```
Text(0.5, 1.0, 'PCA')
```

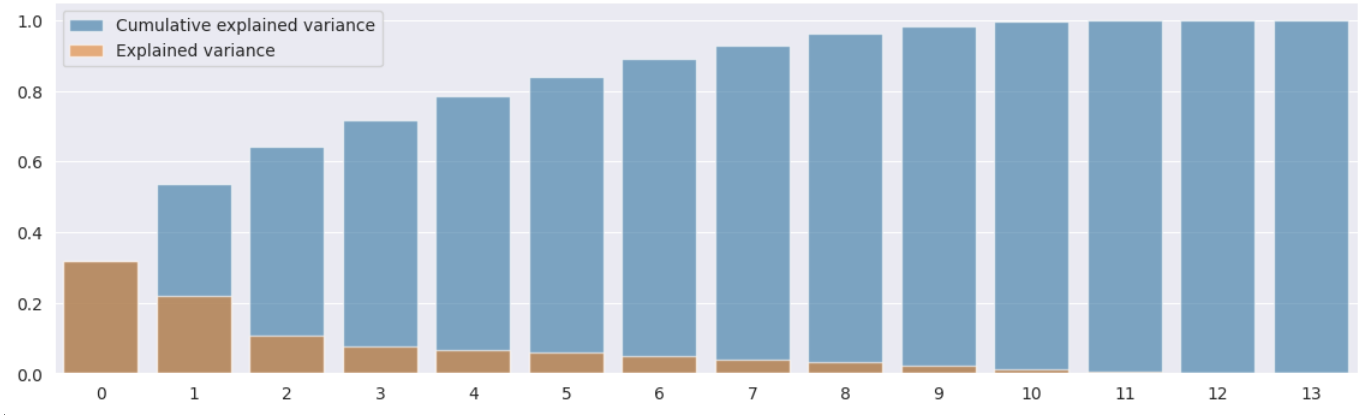

Original
PCA
0 1 2 3 4 5 6 7 8 9 10 11 12 13
0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78 81 84 87 90 93 96 99 102 105 108 111 114 117 120 123 126 129 132 135 138 141 144 147 150 153 156 159 162 165 168 171 174 177 180 183 186 189 192 195 198
10
5
0
−5
−10

In [33]:

```
fig, ax = plt.subplots(figsize=(14, 4))

sns.barplot(
    x=np.arange(pca.n_components_),
    y=pca.explained_variance_ratio_.cumsum(),
    alpha=0.6,
    ax=ax,
    label="Cumulative explained variance",
)
sns.barplot(
    x=np.arange(pca.n_components_),
    y=pca.explained_variance_ratio_,
    alpha=0.6,
    ax=ax,
    label="Explained variance",
)
ax.legend()
```

Out[33]:

```
<matplotlib.legend.Legend at 0x74ed2971ce80>
```



Let's train a mode using the the PCA vectors as features and compare to using all features.

In [34]:

```
numerical = df.select_dtypes(include='number').dropna()
normalised = (numerical - numerical.mean()) / numerical.std()

y = normalised['energy']

# Do not use 'energy' for fitting
X = normalised[normalised.columns.drop('energy')].values
# Model fit using all features (except energy)
y_linear = linear_model.LinearRegression().fit(X, y).predict(X)
```

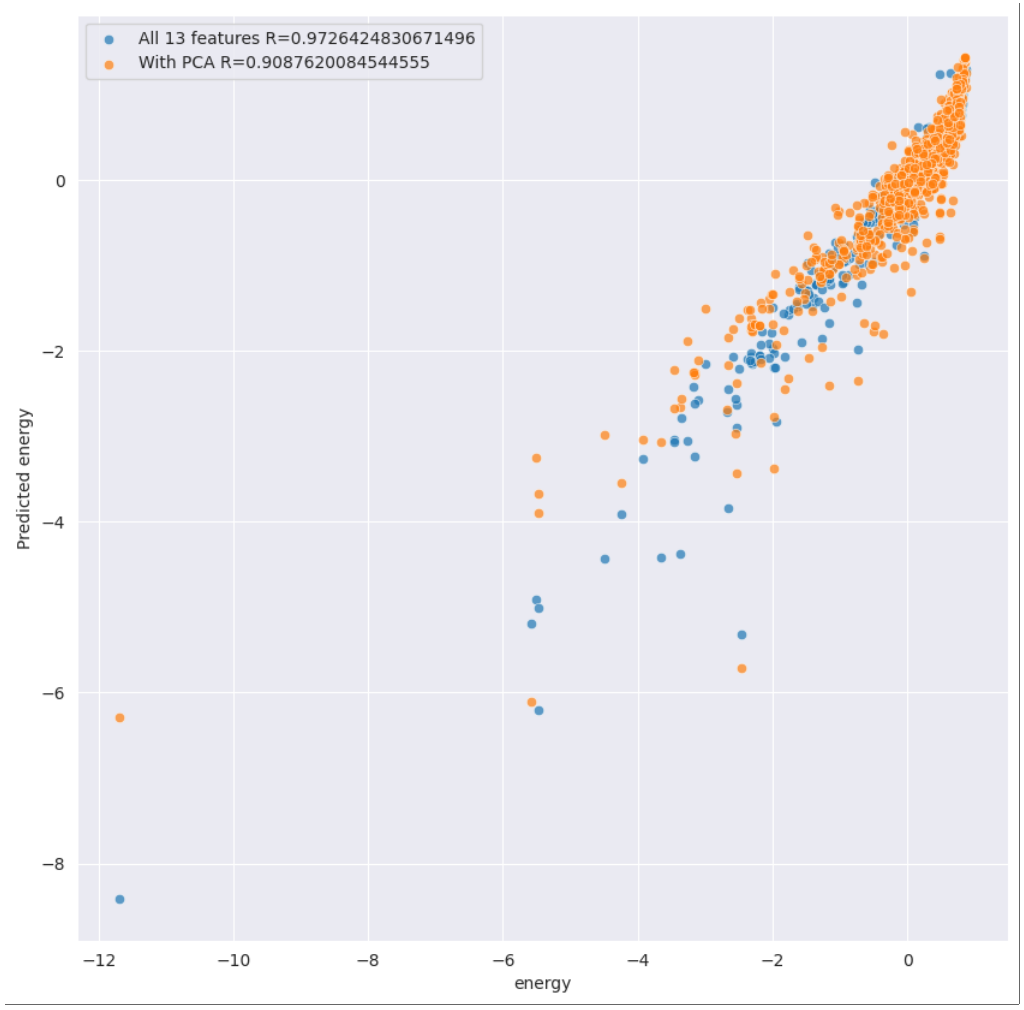
Fit a PCA and train on first few components

In [35]:

```
pca = decomposition.PCA().fit(X)
X_pca = pca.transform(X)[:, 0:4]  # Keep only the first 4
y_pca = linear_model.LinearRegression().fit(X_pca, y).predict(X_pca)
```

In [36]:

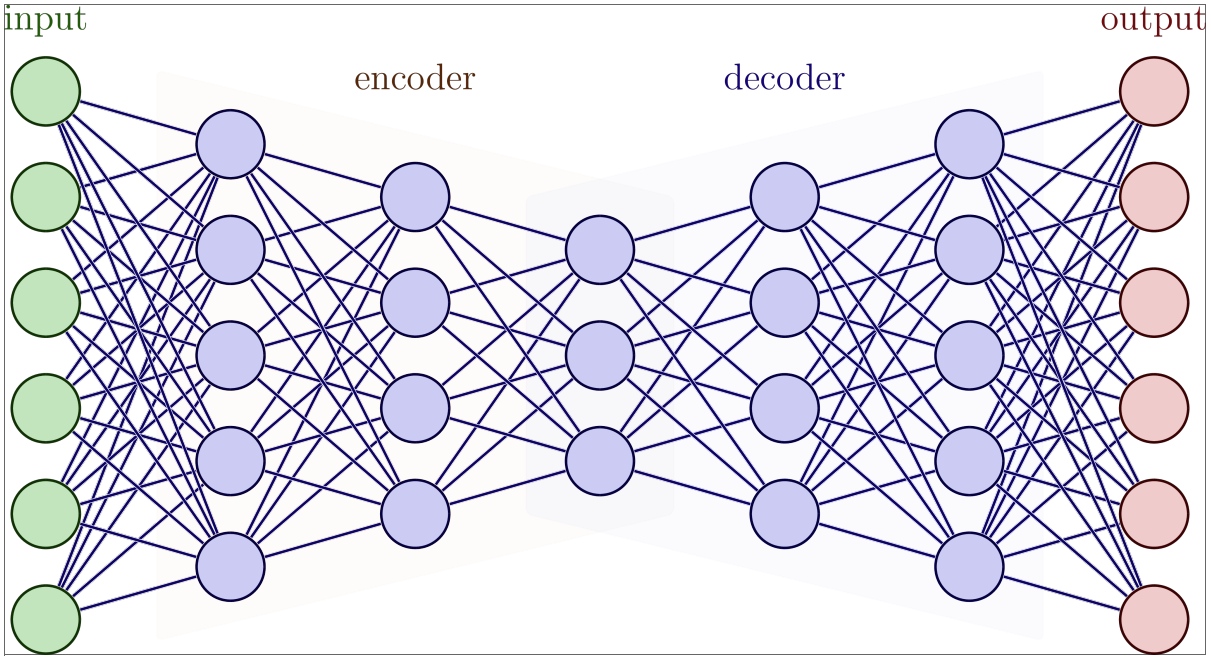
```
fig, ax = plt.subplots(figsize=(10, 10))
# ax.set_aspect("equal")

sns.scatterplot(x=y, y=y_linear, ax=ax, alpha=0.7, label=f"All {X.shape[1]} features R={np.corrcoef(y, y_linear)[0, 1]}")
sns.scatterplot(x=y, y=y_pca, ax=ax, alpha=0.7, label=f"With PCA R={np.corrcoef(y, y_pca)[0, 1]}")

ax.set_ylabel("Predicted energy")
ax.legend();
```


All 13 features R=0.9726424830671496
With PCA R=0.9087620084544555
Predicted energy
energy
0
−2
−4
−6
−8
−12
−10
−8
−6
−4
−2
0

Autoencoders

Can we use neural networks to perform dimensionality reduction?

Yes! One way, is the autoencoder:



This architecture is composed of two networks, the *encoder* and *decoder*, with a *bottleneck* in middle.

We start by defininig a loss function that measures the difference between the input, and the output after passing through the network $\mathcal{L}(x, f_\theta(x))$.
Training then tries to find the θ that allows the network to reconstruct the input as faithfully as possibly. This means that all the pertinent information is forced through the bottleneck, which gives us our compressed representation. Unlike PCA, this method is fully non-linear.

Let's define out network and some hyperparameters

In [37]:

```
from flax import nnx
import optax

rngs = nnx.Rngs(0)
bottleneck = 4  # Number of neurons in the bottleneck
act = nnx.silu  # Our activation function
encoder = nnx.Sequential(
    nnx.Linear(X.shape[1], 10, rngs=rngs),
    act,
    nnx.Linear(10, 8, rngs=rngs),
    act,
    nnx.Linear(8, bottleneck, rngs=rngs),
)
decoder = nnx.Sequential(
    nnx.Linear(bottleneck, 8, rngs=rngs),
    act,
    nnx.Linear(8, 10, rngs=rngs),
    act,
    nnx.Linear(10, X.shape[1], rngs=rngs),
)
autoencoder = nnx.Sequential(encoder, decoder)
```

```
An NVIDIA GPU may be present on this machine, but a CUDA-enabled jaxlib is not installed. Falling back to cpu.
```

Next, let's set up the training loop and run it!

In [38]:

```
@nnx.jit
def train_step(model, optimiser, x):
    def loss_fn(model):
        return ((model(x) - x) ** 2).mean()  # Mean square error

    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimiser.update(grads)
    return loss


optimiser = nnx.Optimizer(autoencoder, optax.adam(1e-3))
for i in range(5000):
    loss = train_step(autoencoder, optimiser, X)
print(i, loss)
```

```
4999 0.081445746
```

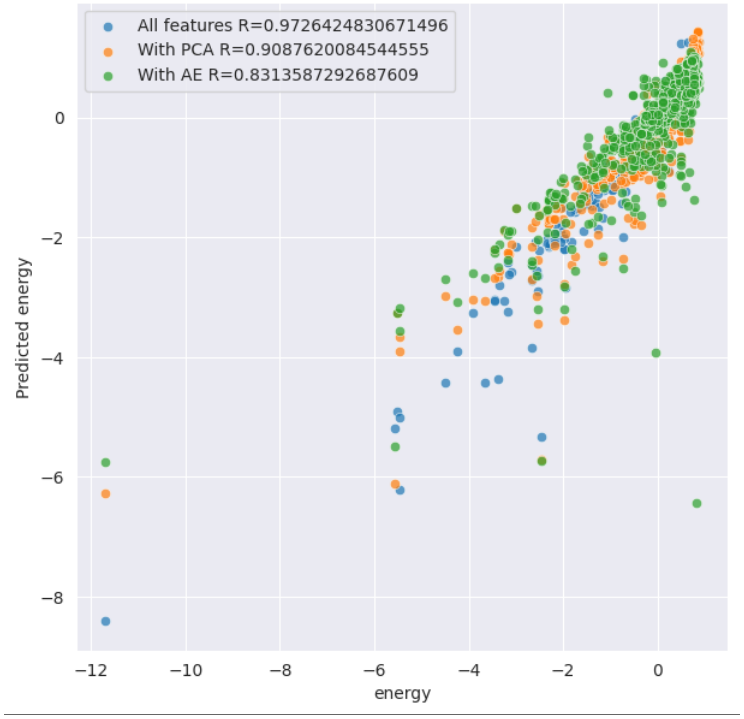
Now, let's try a linear fit with our encoded features

In [39]:

```
X_ae = encoder(X)
y_ae = linear_model.LinearRegression().fit(X_ae, y).predict(X_ae)
```

In [40]:

```
fig, ax = plt.subplots(figsize=(7, 7))

sns.scatterplot(x=y, y=y_linear, ax=ax, alpha=0.7, label=f"All features R={np.corrcoef(y, y_linear)[0, 1]}")
sns.scatterplot(x=y, y=y_pca, ax=ax, alpha=0.7, label=f"With PCA R={np.corrcoef(y, y_pca)[0, 1]}")
sns.scatterplot(x=y, y=y_ae, ax=ax, alpha=0.7, label=f"With AE R={np.corrcoef(y, y_ae)[0, 1]}")

ax.set_ylabel("Predicted energy")
ax.legend();
```

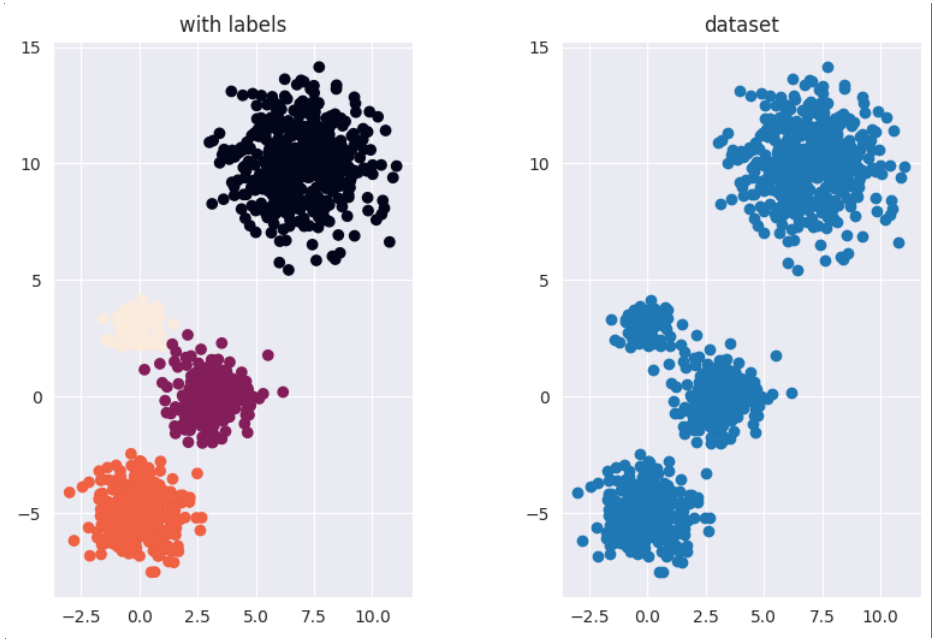


Clustering

Let's create some data

In [41]:

```
from sklearn import datasets, cluster

# The make_blobs function requires the centers, standard deviations and number of samples for each Gaussian to sample.
centers = [[7, 10], [3, 0], [0, -5], [0, 3]]
n_samples = [500, 200, 300, 100]
cluster_std = [1.5, 1, 1, 0.5]

X_blobs, y_blobs = datasets.make_blobs(n_samples=n_samples, random_state=0, cluster_std=cluster_std, centers=centers)

fig, axs = plt.subplots(1, 2, figsize=(10, 6))
axs[0].scatter(X_blobs[:, 0], X_blobs[:, 1], c=y_blobs); axs[0].set_title("with labels"); axs[0].set_aspect("equal")
axs[1].scatter(X_blobs[:, 0], X_blobs[:, 1]); axs[1].set_title("dataset"); axs[1].set_aspect("equal")
```



K-Means

Given some d dimensional data $(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n})$ **k-means** clustering tries to partition the observations into k sets $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ in such a way that the within-cluster sum of square distances is minimised i.e.

$$\underset{\mathbf{S}}{\operatorname{argmin}} \sum_{i}^{k} \sum_{x \in S_i} \|\mathbf{x} - \mu_\mathbf{i}\|^2$$

where μ is the centroid of points in S_i.
While finding the optimal solution is NP-hard, there are a number of efficient algorithms that converge to a solution.

LLOYD'S ALGORITHM

Starting with a set of k-means, $m_1^{(1)}, \ldots, m_k^{(1)}$, we iterate between two steps

1. **Assignment** Assign each data point to the nearest mean (usually using the Eucliean distance):

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\},$$

2. **Update step**: Now, we can recalculate the means (centroid of our clusters) for data assigned to each cluster:

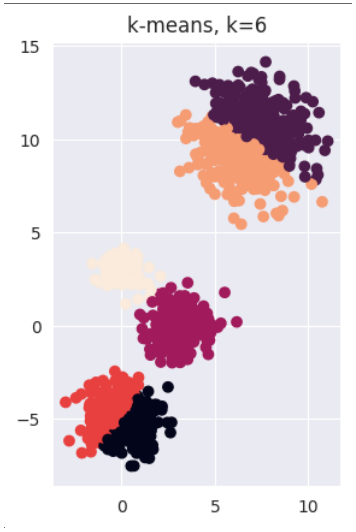
$$m_i^{(t+1)} = \frac{1}{\left| S_i^{(t)} \right|} \sum_{x_j \in S_i^{(t)}} x_j$$

We terminate when the assignments cease to change.

Let's have a look at this in action

In [42]:

```
n_clusters = 6
labels = cluster.KMeans(n_clusters=n_clusters, random_state=42).fit_predict(X_blobs)
plt.title("k-means, k=6")
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels)
plt.gca().set_aspect("equal")
```



You might notice that the top-right cluster has been split, this is because we specified that we are looking for 6 clusters. In fact, for k-means, we always have to tell it how many clusters we are looking for.

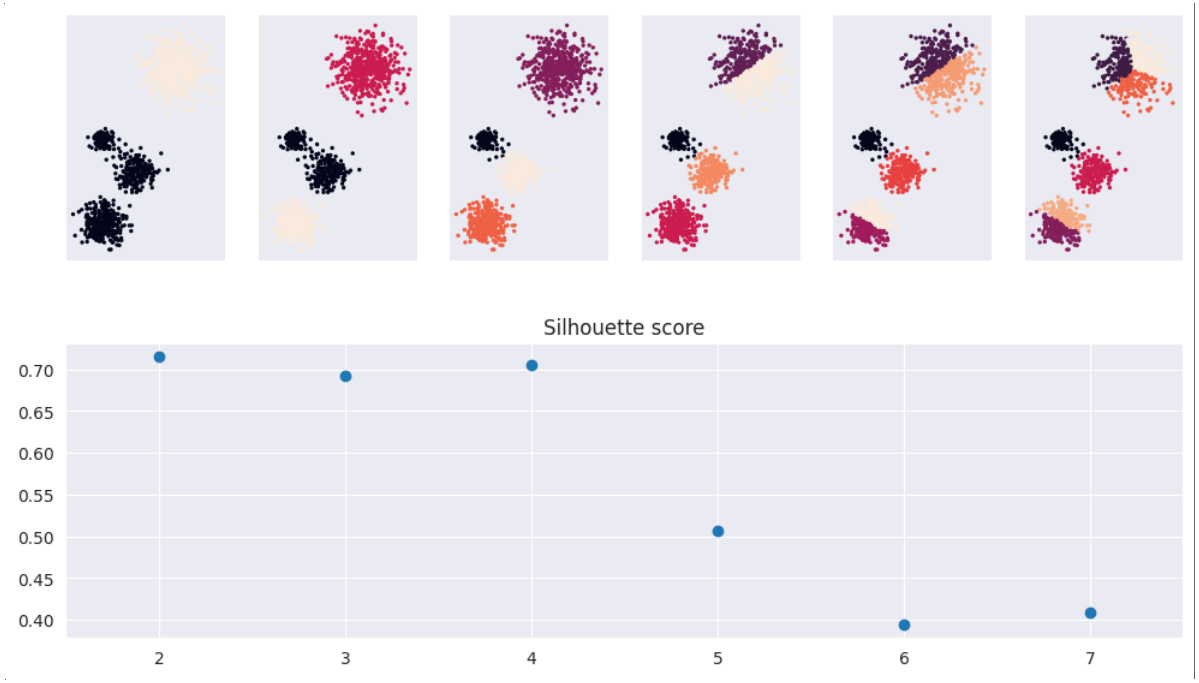
So how can we find the *optimum* number of clusters? There are a number of methods, one method that is commonly used is to calculate the **Silhouette coefficient**, defined as

$$s = \frac{b - a}{\max(a, b)}$$

where a is the mean distance between a sample and all other points in the same class, and b is the mean distance between a sample and all other points in the *next nearest cluster*.

In [43]:

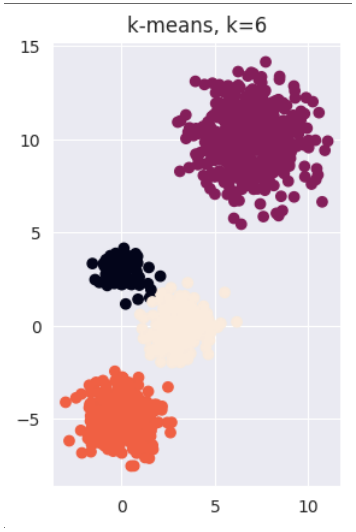
```
from sklearn import metrics

all_clusters = list(range(2, 8))

fig = plt.figure(figsize=(2 * len(all_clusters), 7))

silhouettes = []
for i, n_clusters in enumerate(all_clusters):
    ax = fig.add_subplot(2, len(all_clusters), i + 1)
    labels = cluster.KMeans(n_clusters=n_clusters, init="k-means++", random_state=1).fit_predict(X_blobs)

    # Calculate our scores
    silhouettes.append(metrics.silhouette_score(X_blobs, labels, metric="euclidean"))
    ax.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels, s=2)
    ax.axes.get_xaxis().set_visible(False); ax.axes.get_yaxis().set_visible(False); ax.set_aspect("equal")

ax = fig.add_subplot(2, 1, 2)
ax.scatter(all_clusters, silhouettes)
ax.set_xlim(all_clusters[0] - 0.5, all_clusters[-1] + 0.5)
ax.set_title("Silhouette score");
```


Silhouette score
0.70
0.65
0.60
0.55
0.50
0.45
0.40
2
3
4
5
6
7

For the Silhouette score we are looking for a jump io the score as we reduce the number of clusters.

In [44]:

```
n_clusters = 4
labels = cluster.KMeans(n_clusters=n_clusters, random_state=1).fit_predict(X_blobs)
plt.title("k-means, k=6")
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels)
plt.gca().set_aspect("equal")
```



Advantages

- Simplicity
- Guaranteed to converge
- Adapts smoothly to new examples
- Can be adapted to different shapes

Disadvantages

- Must choose k
- Sensitive to choise of initial $m_i^{(0)}$
- No concept of outliers (centroids will be 'dragged')
- Works poorly in high dimensional spaces

Gaussian Mixture Model

Suppose we assume our data comes from a mixture of independent Gaussians

$$p(\boldsymbol{\theta}) = \sum_{i}^{K} w_i \mathcal{N}(\boldsymbol{\mu_i}, \mathbf{\Sigma_i}),$$

but we don't know the weights, w_i, means $\mu_\mathbf{i}$, nor covariances $\mathbf{\Sigma_i}$, we only have the observed data $\mathbf{X}$ i.e. samples from the distribution.
Now, we can use the **expectation-maximisation** (EM) algorithm to find $\boldsymbol{\theta}$ (the set of μ_i, σ_i).

Expectation Maximisation

We want to maximise the marginal likelihood of the observed data. To do this, we will introduce a set of unobserved latent (or missing) values $\boldsymbol{Z}$ which would tell us which Gaussian each point belong to:

$$L(\boldsymbol{\theta}; \boldsymbol{X}) = p(\mathbf{X}|\boldsymbol{\theta}) = \int p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) d\mathbf{Z} = \int p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\theta}) p(\mathbf{Z}|\boldsymbol{\theta}) d\mathbf{Z},$$

where in the last step we made use of the chain rule $P(A, B) = P(A|B)P(B)$. Clearly, because we don't know $\boldsymbol{Z}$, we can't calculate this directly.

So, EM algorithm proceeds in two steps:

1. *Expectation step*: define an expectation value of the log likelihood of $\boldsymbol{\theta}$ based on the current values $\boldsymbol{\theta}^t$:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = E_{\mathbf{Z} \sim p(\cdot|\mathbf{X}, \boldsymbol{\theta}^{(t)})}[\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})]$$

   in this step, we find the $\mathbf{Z}$.
2. *Maximisation step*: find the parameters that maximise this:

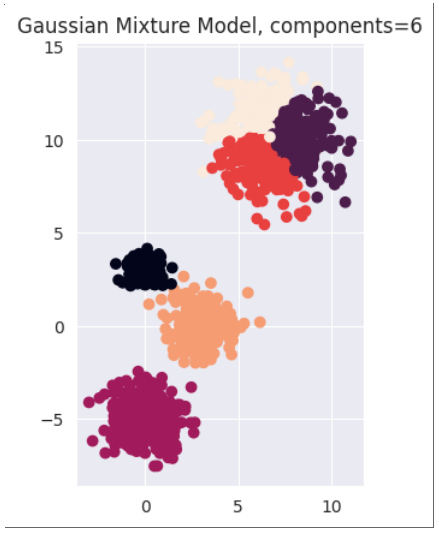
$$\boldsymbol{\theta}^{(t+1)} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, E_{\mathbf{Z} \sim p(\cdot|\mathbf{X}, \boldsymbol{\theta}^{(t)})}[\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})]$$

This repeats until a local maximum is found. See **here** for a more comprehensive explanation.

Let's try it in code!

In [45]:

```
from sklearn import mixture

labels = mixture.GaussianMixture(n_components=6).fit_predict(X_blobs)
plt.title("Gaussian Mixture Model, components=6")
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels)
plt.gca().set_aspect("equal")
```



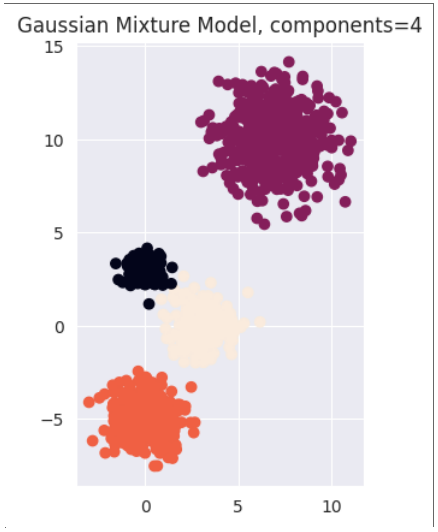
In [46]:

```
from sklearn import metrics

all_clusters = list(range(2, 8))

fig = plt.figure(figsize=(2 * len(all_clusters), 7))

silhouettes = []
for i, n_clusters in enumerate(all_clusters):
    ax = fig.add_subplot(2, len(all_clusters), i + 1)
    labels = mixture.GaussianMixture(n_components=n_clusters, random_state=1).fit_predict(X_blobs)

    # Calculate our scores
    silhouettes.append(metrics.silhouette_score(X_blobs, labels, metric="euclidean"))
    ax.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels, s=2)
    ax.axes.get_xaxis().set_visible(False); ax.axes.get_yaxis().set_visible(False); ax.set_aspect("equal")

ax = fig.add_subplot(2, 1, 2)
ax.scatter(all_clusters, silhouettes)
ax.set_xlim(all_clusters[0] - 0.5, all_clusters[-1] + 0.5)
ax.set_title("Silhouette score");
```


Silhouette score
0.70
0.65
0.60
0.55
0.50
0.45
0.40
2
3
4
5
6
7

In [47]:

```
from sklearn import mixture

labels = mixture.GaussianMixture(n_components=4, random_state=1).fit_predict(X_blobs)
plt.title("Gaussian Mixture Model, components=4")
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels)
plt.gca().set_aspect("equal")
```



In [48]:

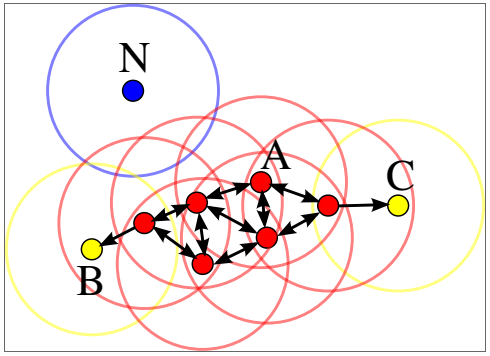
```
gmm = mixture.GaussianMixture(n_components=4, random_state=1).fit_predict(X_blobs)
km = cluster.KMeans(n_clusters=4, random_state=1).fit_predict(X_blobs)

fig, axs = plt.subplots(1, 2, figsize=(8, 4))
axs[0].scatter(X_blobs[:, 0], X_blobs[:, 1], c=km)
axs[0].set_title("$k$-means")
axs[1].scatter(X_blobs[:, 0], X_blobs[:, 1], c=gmm)
axs[1].set_title("GMM");
```

DBSCAN

DBSCAN is a density based algorithm, and tries to classify points into one of three categories:

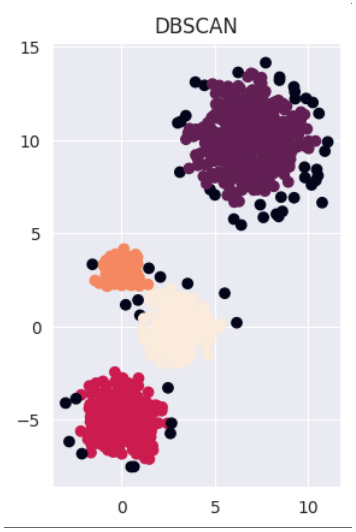
1. Core points - points located near the interior of a cluster (usually the higher-density area)
2. Non-core points - still part of acluster, located at the periphery (usually lower-density area)
3. Outliers - not part of any cluster, far from other pont

- A point p is a *core point* if at least minPts are within ditsance ϵ of it
- A point q is *directly reachable* from p if q is within distance ϵ. Points are only directly reachable from core points.
- A point q is *reachable* from p if there is a path $p_1, \ldots, p_n$ with $p_1 = p$ and $p_n = q$, where each p_{i+1} is directly reachable from p_i.
- All points not reachable from any other points are *outliers*.



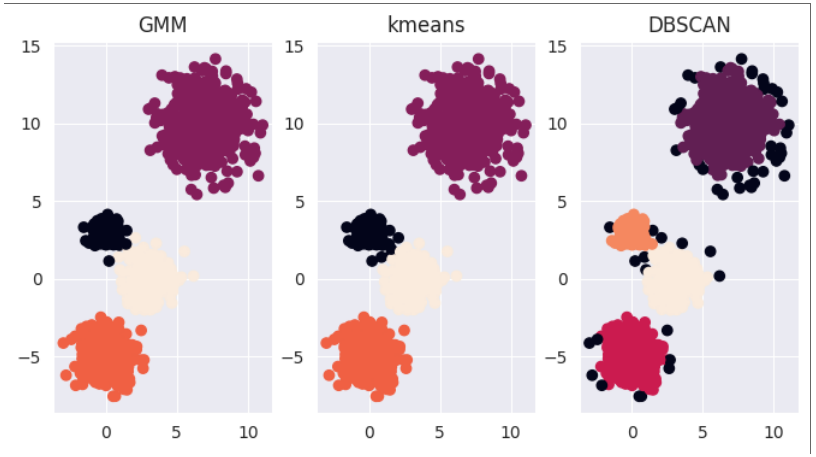
In [49]:

```
labels = cluster.DBSCAN(eps=0.6).fit_predict(X_blobs)
plt.title("DBSCAN")
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels)
plt.gca().set_aspect("equal")
```



In [50]:

```
methods = dict(
    GMM = mixture.GaussianMixture(n_components=4, random_state=1),
    kmeans = cluster.KMeans(n_clusters=4, random_state=1),
    DBSCAN = cluster.DBSCAN(eps=0.6),
)

fig, axs = plt.subplots(1, len(methods), figsize=(8, 4))
for ax, (name, method) in zip(axs, methods.items()):
    labels = method.fit_predict(X_blobs)
    ax.scatter(X_blobs[:, 0], X_blobs[:, 1], c=labels)
    ax.set_title(name)
```



Advantages

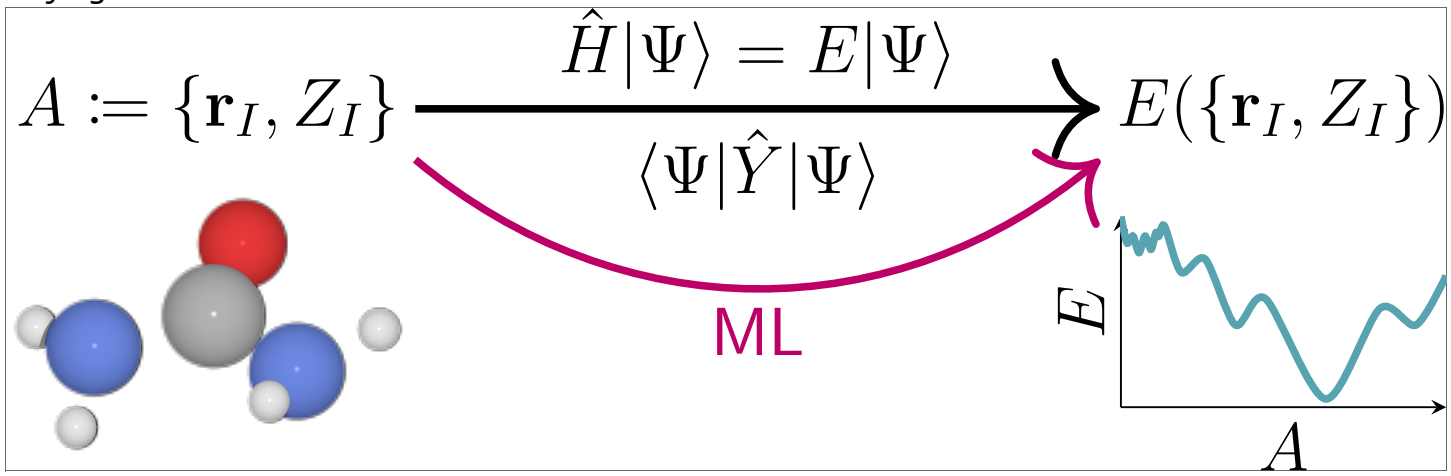
- Does not require us to specify k
- Can find clusters of arbitrary shape
- Is robust to outliers
- Requires only two parameters

Disadvantages

- Not fully deterministic
- Is sensitive to choice of distance function (especially in high-dimensional spaces)
- Cannot cope with large density differences
- Choosing ϵ can be difficult

Machine Learning for Atomistic Systems

What we're trying to do:



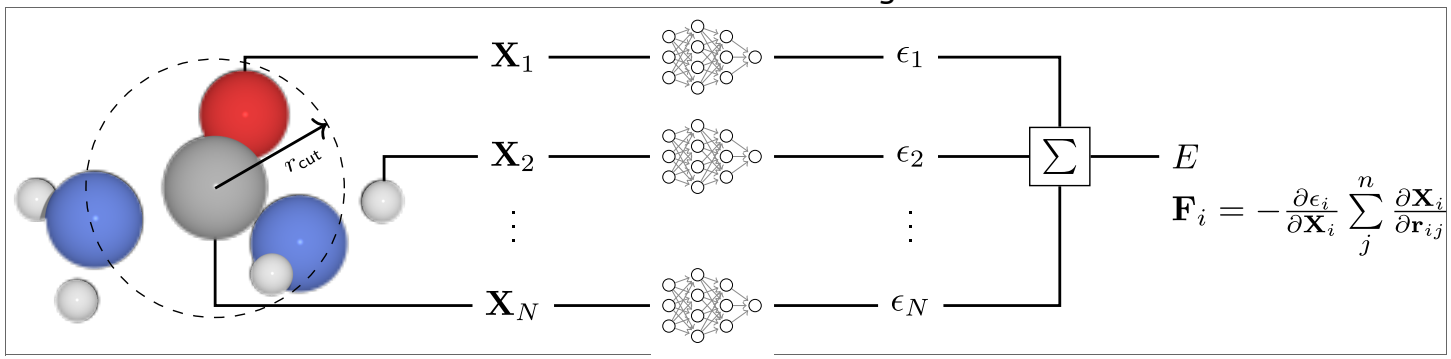
We want our model to have at least the following properties:

0. Size extensivity
1. Permutational invariance
2. Translational invariance
3. Rotational invariance

Nice to have (particularly for generative models)

4. Invertability (can recover atomic positions and species)
5. Univeral (not tuned to particular dataset)

Pretty much *the* standard model architecture that's used is the following:



In terms of our outputs, this satisfies the following criteria:

0. ☑ Size extensivity (we can have as many atomic environments, N, as we want)
1. ☑ Permutational invariance (because we sum at the end)
2. ☑ Translational invariance (because we use *local* atomic environments)
3. ☐ Rotational invariance (not really, we will come back to this!)

Creating a model for predicting properties from atoms

Let's start with a toy model: Lennard-Jones cluster

Data generation

In [4]:

```
def lennard_jones(r):
    return 4. * ((1. / r)**12 - (1 / r)**6)

def pdist(x):  # Get all the pair distances
    ind = jnp.triu_indices(x.shape[0], k=1)
    return jnp.linalg.norm(x[ind[1]] - x[ind[0]], axis=1)

def total_energy(pos):  # Sum all the LJ pair energies
    return lennard_jones(pdist(pos)).sum()

n_atoms = 8
pos =  3. * np.random.rand(n_atoms, 3)
show_array(pos, show_numbers=False)
```

xrot

0.00

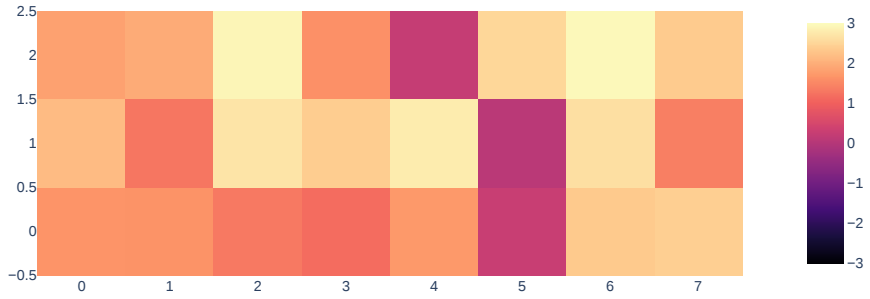
xtrans

0.00

swap

0

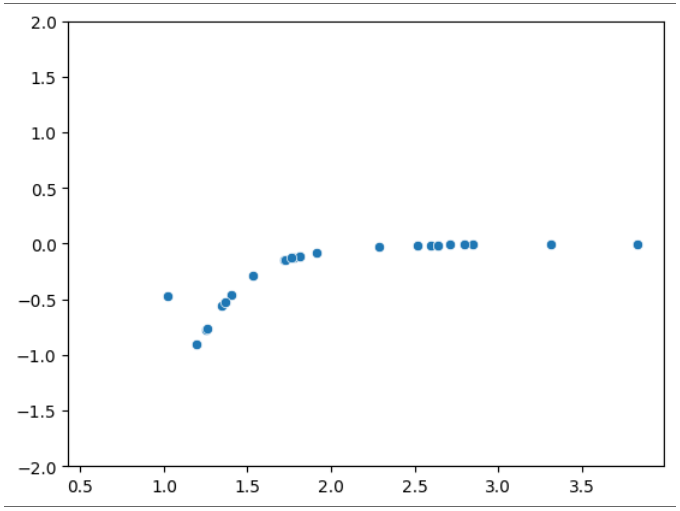
Out[4]:



In [5]:

```
dists = pdist(pos)
energies = lennard_jones(dists)
ax = sns.scatterplot(x=dists, y=energies)
ax.set_ylim(-2, 2);
```

```
An NVIDIA GPU may be present on this machine, but a CUDA-enabled jaxlib is not installed. Falling back to cpu.
```



In [6]:

```
view(ase.Atoms(positions=pos))
```

Out[6]:

Show All

Color scheme element

Ball size

0.50

Let's minimise the energy

In [7]:

```
from flax import linen
import optax

tx = optax.adam(learning_rate=5e-3)
opt_state = tx.init(pos)
grad_fn = jax.value_and_grad(total_energy)  #  Using automatic differentiation

prog = tqdm.tqdm(range(200))
for i in prog:
    energy, forces = grad_fn(pos)  # Calculate energy and forces
    updates, opt_state = tx.update(forces, opt_state)
    pos = optax.apply_updates(pos, updates)  # Update the positions
    if i % 10 == 0:
        prog.set_description(f"{i} {energy}")
```

```
190 -9.225740432739258: 100%|██████████| 200/200 [00:03<00:00,
61.53it/s]
```

In [8]:

```
view(ase.Atoms(positions=pos))
```

Out[8]:

Show All

Color scheme element

Ball size

0.50

Create some training data by randomly displacing atoms

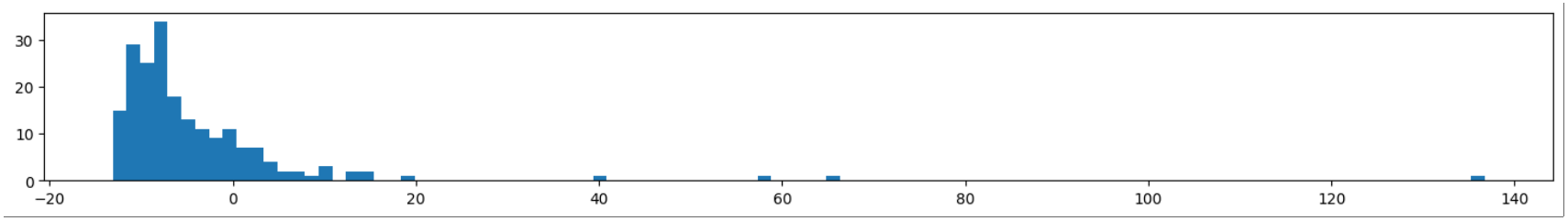
In [10]:

```
train_pos = []
train_energies = []
train_forces = []

for i in range(200):
    atoms = ase.Atoms(positions=relaxed)
    atoms.rattle(stdev=0.05, seed=i)
    pos = atoms.positions.copy()
    energy, grad = nnx.value_and_grad(total_energy)(pos)

    train_pos.append(pos)
    train_energies.append(total_energy(pos))
    train_forces.append(-grad)

train_pos, train_energies, train_forces = tuple(map(jnp.array, (train_pos, train_energies, train_forces)))
```

Distribution of training energies

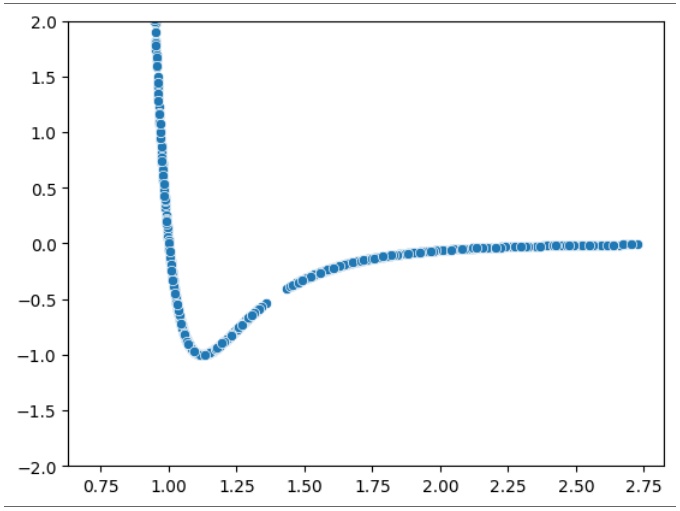
In [11]:

```
plt.figure(figsize=(18, 2))
plt.hist(train_energies, bins=100);
```

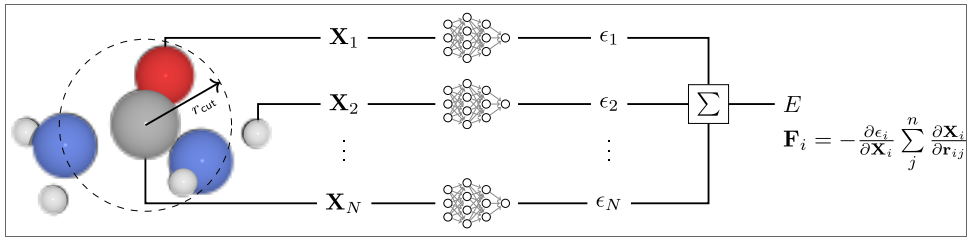


In [12]:

```
dists = jax.vmap(pdist)(train_pos).flatten()
ax = sns.scatterplot(x=dists, y=lennard_jones(dists))
ax.set_ylim(-2, 2);
```



Now, let's create our model based on this architecture:



In [14]:

```
from flax import nnx

rngs = nnx.Rngs(0)
act = nnx.silu  # Our activation function
bias = False
model = nnx.Sequential(
    nnx.Linear(1, 20, rngs=rngs, use_bias=bias),  # Input
    act,
    nnx.Linear(20, 20, rngs=rngs, use_bias=bias),  # Hidden layer 1
    act,
    nnx.Linear(20, 20, rngs=rngs, use_bias=bias),  # Hidden layer 2
    act,
    nnx.Linear(20, 1, rngs=rngs, use_bias=bias),  # Output
)

def predict_energy(model, pos):
    dists = pdist(pos)  # Get the distances

    # Predict an energy per atom
    energies = nnx.vmap(model)(dists.reshape(-1, 1))

    # Sum energies in each input structure to get global energy
    return energies.sum()

predict_energies = nnx.vmap(predict_energy, in_axes=(None, 0), state_axes={None: 0})
```

Now, let's create a training loop and adjust the parametrers to minimise the loss

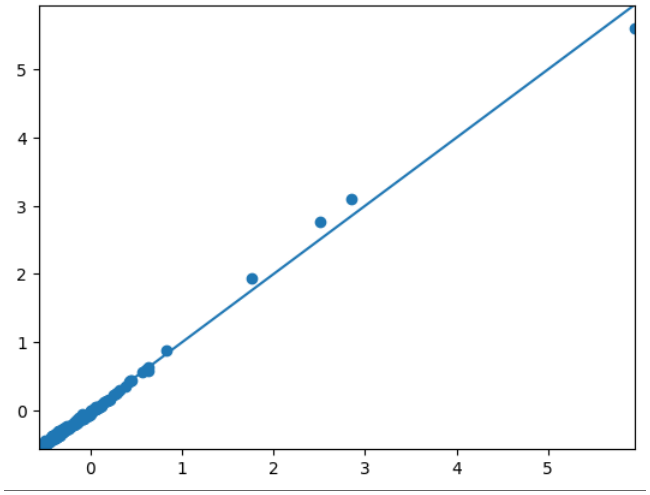
In [15]:

```
@nnx.jit
def train_step(model, optimiser, pos, energy_labels, force_labels):
    def loss(energy_predictions):
        return ((energy_predictions - energy_labels) ** 2).mean()

    def calc_loss(model):
        energy_predictions = predict_energies(model, pos)
        return loss(energy_predictions)

    loss, grads = nnx.value_and_grad(calc_loss)(model)
    optimiser.update(grads)
    return loss

optimiser = nnx.Optimizer(model, optax.adamw(2e-3))
losses = []
prog = tqdm.tqdm(range(20000))
for i in prog:
    loss = train_step(model, optimiser, train_pos, train_energies, train_forces)
    if i % 100 == 0:
        losses.append(loss.item())
        prog.set_description(f"train loss {loss}")
```

```
train loss 0.0019335217075422406: 100%|██████████| 20000/20000 [02:21<00:00,
141.24it/s]
```

In [16]:

```
plt.scatter(train_energies, predict_energies(model, train_pos))
eminmax = train_energies.min(), train_energies.max()
plt.plot(eminmax, eminmax)
plt.xlim(eminmax)
plt.ylim(eminmax);
```



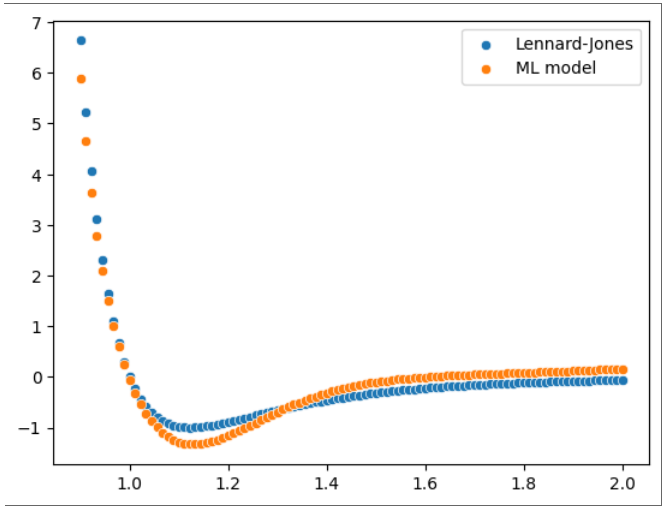
Let's have a look at what the network learned

In [18]:

```
x = jnp.linspace(0.9, 2, 100)
sns.scatterplot(x=x, y=lennard_jones(x), label="Lennard-Jones")
sns.scatterplot(x=x, y=jax.vmap(model)(x.reshape(-1, 1)).flatten() * force_std, label="ML model")
```

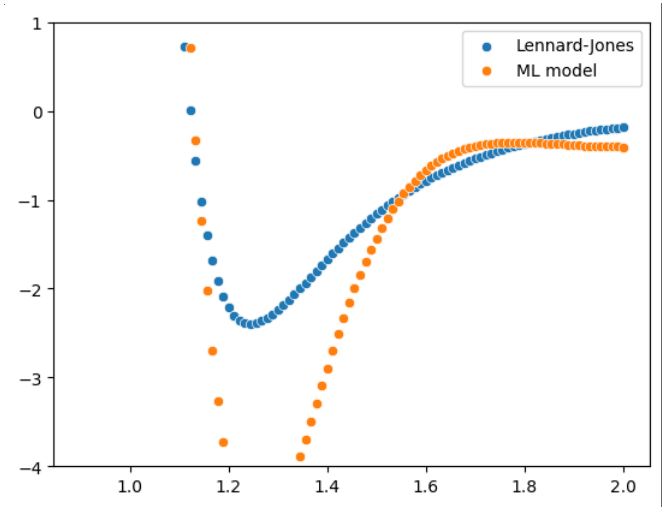
Out[18]:

```
<Axes: >
```



In [19]:

```
x = jnp.linspace(0.9, 2, 100)
sns.scatterplot(x=x, y=-nnx.vmap(nnx.grad(lennard_jones))(x), label="Lennard-Jones")
ax = sns.scatterplot(x=x, y=-nnx.vmap(nnx.grad(lambda x: model(x)[0]))(x.reshape(-1, 1)).flatten() * force_std, label="ML model")
ax.set_ylim((-4, 1));
```



Energies and forces

In [20]:

```
rngs = nnx.Rngs(0)
act = nnx.silu  # Our activation function
bias = False
model = nnx.Sequential(
    nnx.Linear(1, 20, rngs=rngs, use_bias=bias),  # Input
    act,
    nnx.Linear(20, 20, rngs=rngs, use_bias=bias),  # Hidden layer 1
    act,
    nnx.Linear(20, 20, rngs=rngs, use_bias=bias),  # Hidden layer 2
    act,
    nnx.Linear(20, 1, rngs=rngs, use_bias=bias),  # Output
)
```

In [21]:

```
def predict_energy_and_forces(model, pos):
    energy, grads = nnx.value_and_grad(predict_energy, argnums=1)(model, pos)
    return energy, -grads

predict_energies_and_forces = nnx.vmap(predict_energy_and_forces, in_axes=(None, 0), state_axes={None: 0})
```

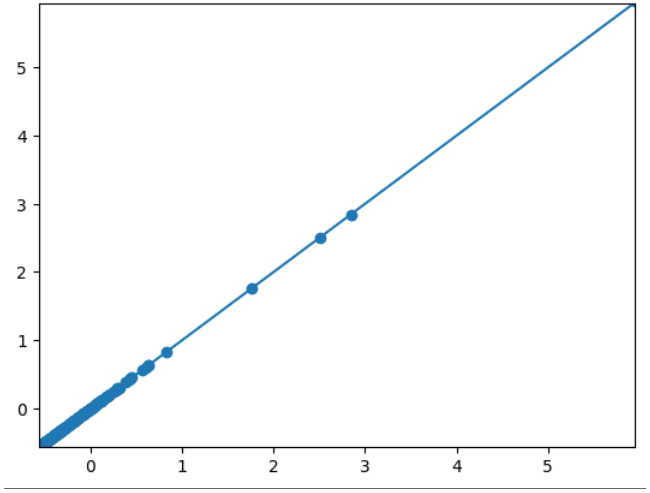
In [24]:

```
@nnx.jit
def train_step(model, optimiser, pos, energy_labels, force_labels):
    def loss(energy_predictions, force_predictions):
        energy_loss = ((energy_predictions - energy_labels) ** 2).mean()
        force_loss = ((force_predictions - force_labels) ** 2).mean()
        return energy_loss + 10. * force_loss

    def calc_loss(model):
        energy_predictions, force_predictions = predict_energies_and_forces(model, pos)
        return loss(energy_predictions, force_predictions)

    loss, grads = nnx.value_and_grad(calc_loss)(model)
    optimiser.update(grads)
    return loss

optimiser = nnx.Optimizer(model, optax.adamw(2e-3))
prog = tqdm.tqdm(range(20000))
for i in prog:
    loss = train_step(model, optimiser, train_pos, train_energies, train_forces)
    if i % 100 == 0:
        losses.append(loss.item())
        prog.set_description(f"train loss {loss}")
```

```
train loss 0.0008542800205759704: 100%|██████████| 20000/20000 [04:12<00:00,
79.30it/s]
```

In [25]:

```
plt.scatter(train_energies, predict_energies(model, train_pos))
eminmax = train_energies.min(), train_energies.max()
plt.plot(eminmax, eminmax)
plt.xlim(eminmax)
plt.ylim(eminmax);
```

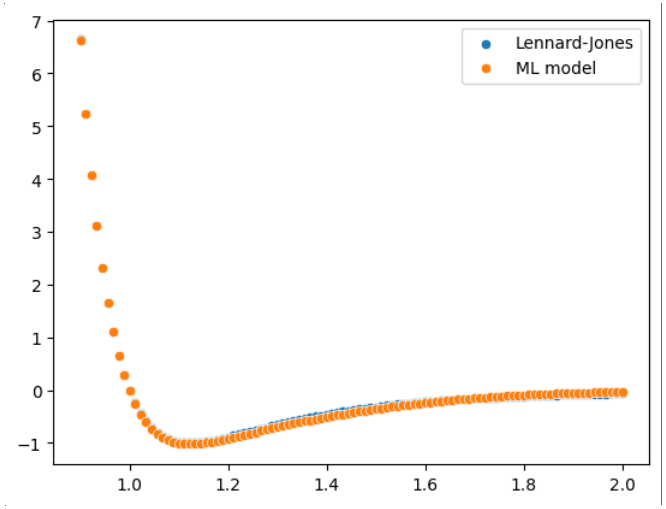


In [27]:

```
x = jnp.linspace(0.9, 2, 100)
sns.scatterplot(x=x, y=lennard_jones(x), label="Lennard-Jones")
sns.scatterplot(x=x, y=jax.vmap(model)(x.reshape(-1, 1)).flatten() * force_std, label="ML model")
```

Out[27]:

```
<Axes: >
```

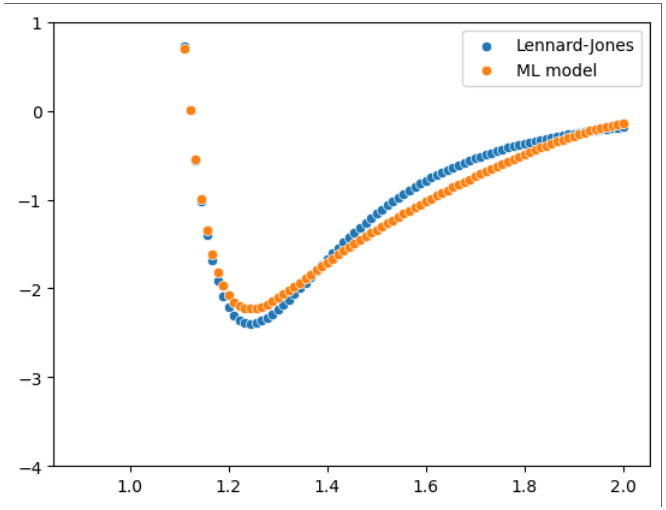


In [28]:

```
x = jnp.linspace(0.9, 2, 100)
sns.scatterplot(x=x, y=-nnx.vmap(nnx.grad(lennard_jones))(x), label="Lennard-Jones")
ax = sns.scatterplot(x=x, y=-nnx.vmap(nnx.grad(lambda x: model(x)[0]))(x.reshape(-1, 1)).flatten() * force_std, label="ML model")
ax.set_ylim((-4, 1))
```

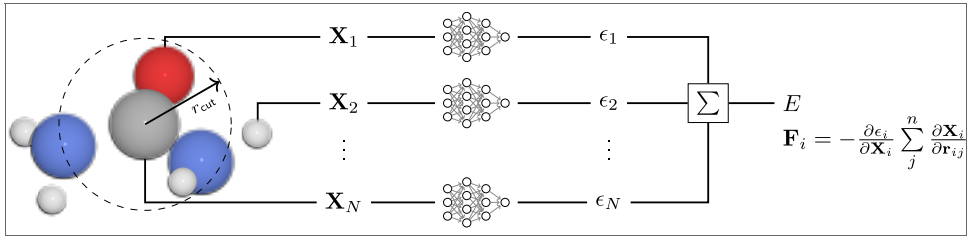
Out[28]:

```
(-4.0, 1.0)
```



Constructing many-body descriptors

Let's come back to our architecture



The actual effective, quantum mechanical, interactions between atoms are *many body*, meaning that the energy (and all observables) depends on the positions of all atoms simultaneously. This means that X_i should also needs to contain information about all the atoms (within the cutoff sphere), and have the usual properties:

1. Permutational invariance
2. Translational invariance
3. Rotational invariance

In [29]:

```
molecule = build.molecule('CH3COCH3')
view(molecule)
```

Out[29]:

Show All

Color scheme element

Ball size

0.50

In [30]:

```
show_array(molecule.positions)
```

xrot

0.00

xtrans

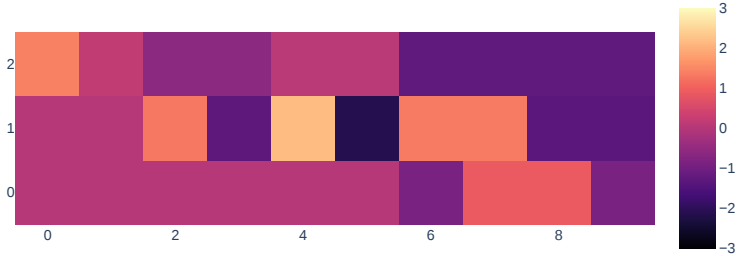
0.00

swap

0

```
[[ 0.     0.     1.406]
 [ 0.     0.     0.179]
 [ 0.     1.285 -0.616]
 [ 0.    -1.285 -0.616]
 [ 0.     2.135  0.067]
 [ 0.    -2.135  0.067]
 [-0.881  1.332 -1.264]
 [ 0.881  1.332 -1.264]
 [ 0.881 -1.332 -1.264]
 [-0.881 -1.332 -1.264]]
```

Out[30]:



1. Permutation

In [31]:

```
def sum_coords(pos):
    # Sum the coordinates along x, y and z
    return pos.sum(axis=0).reshape(3, 1)

show_array(molecule.positions, sum_coords)
```

xrot

0.00

xtrans

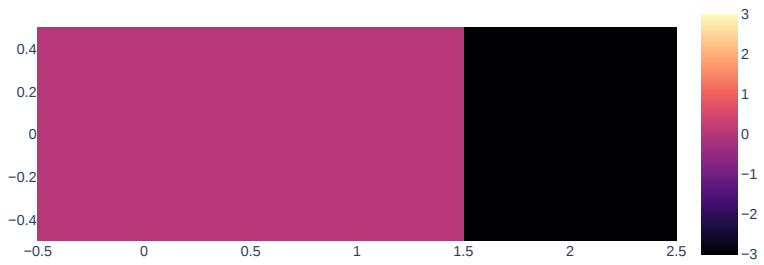
0.00

swap

0

```
[[ 0.     0.     1.406]
 [ 0.     0.     0.179]
 [ 0.     1.285 -0.616]
 [ 0.    -1.285 -0.616]
 [ 0.     2.135  0.067]
 [ 0.    -2.135  0.067]
 [-0.881  1.332 -1.264]
 [ 0.881  1.332 -1.264]
 [ 0.881 -1.332 -1.264]
 [-0.881 -1.332 -1.264]]
```

Out[31]:



2. Translation

In [32]:

```
def center(pos):
    # Subtract centre of mass
    return (pos - pos.sum(axis=0) / len(pos))

show_array(molecule.positions, center)
```

xrot

0.00

xtrans

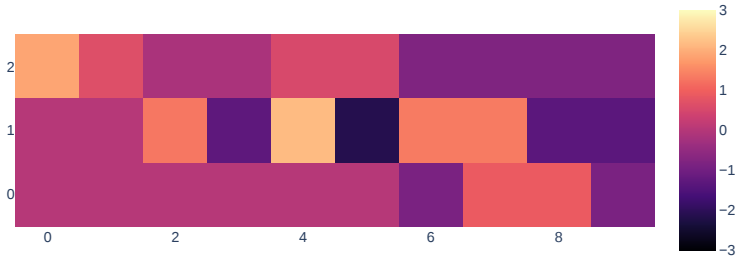
0.00

swap

0

```
[[ 0.     0.     1.406]
 [ 0.     0.     0.179]
 [ 0.     1.285 -0.616]
 [ 0.    -1.285 -0.616]
 [ 0.     2.135  0.067]
 [ 0.    -2.135  0.067]
 [-0.881  1.332 -1.264]
 [ 0.881  1.332 -1.264]
 [ 0.881 -1.332 -1.264]
 [-0.881 -1.332 -1.264]]
```

Out[32]:



3. Rotation

In [33]:

```
def norms(pos):
    # Get norms of pos vectors
    return (pos**2).sum(axis=1).reshape(-1, 1)

show_array(molecule.positions, norms)
```

xrot

0.00

xtrans

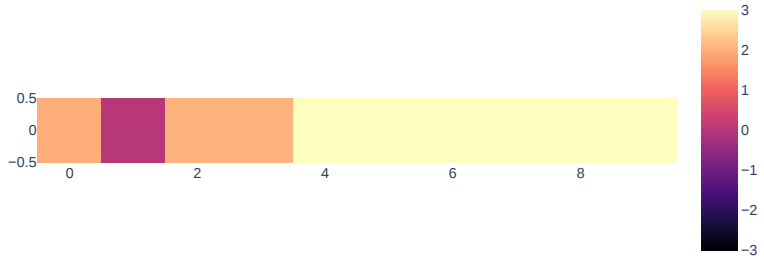
0.00

swap

0

```
[[ 0.     0.     1.406]
 [ 0.     0.     0.179]
 [ 0.     1.285 -0.616]
 [ 0.    -1.285 -0.616]
 [ 0.     2.135  0.067]
 [ 0.    -2.135  0.067]
 [-0.881  1.332 -1.264]
 [ 0.881  1.332 -1.264]
 [ 0.881 -1.332 -1.264]
 [-0.881 -1.332 -1.264]]
```

Out[33]:



Putting it all together

In [34]:

```
def invariance(pos):
    trans = center(pos)
    rots = norms(trans)
    return rots.sum().reshape(-1, 1)
#     return sum_coords(rots)

show_array(molecule.positions, invariance)
```

xrot

0.00

xtrans

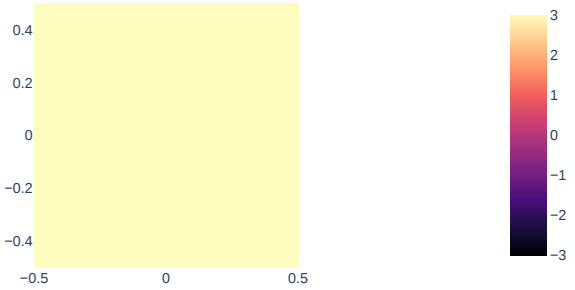
0.00

swap

0

```
[[ 0.     0.     1.406]
 [ 0.     0.     0.179]
 [ 0.     1.285 -0.616]
 [ 0.    -1.285 -0.616]
 [ 0.     2.135  0.067]
 [ 0.    -2.135  0.067]
 [-0.881  1.332 -1.264]
 [ 0.881  1.332 -1.264]
 [ 0.881 -1.332 -1.264]
 [-0.881 -1.332 -1.264]]
```

Out[34]:



Moments to the rescue

The first thing we do with most data sets is calcualte the moments

0^{TH} MOMENT: TOTAL MASS

$$\sum_i 1 = n$$

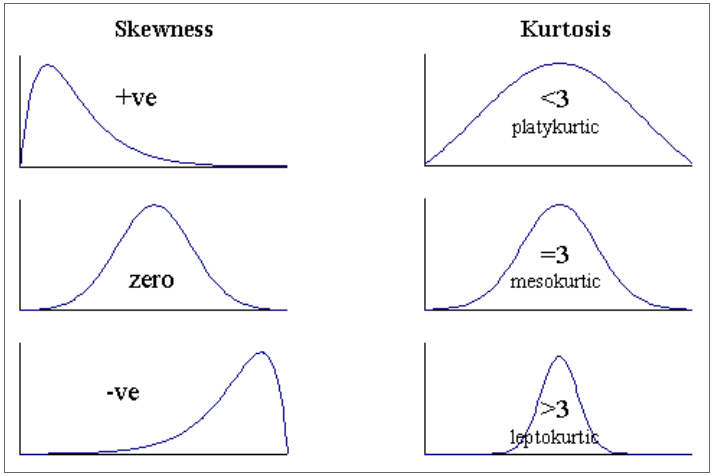
1^{ST} (RAW) MOMENT: MEAN

$$\frac{1}{n} \sum_i x_i = \mu$$

2^{ND} (CENTRAL) MOMENT: VARIANCE

$$\left(\sum_i x_i^2\right) - \mu^2 = \sigma^2$$

3: skewness, 4: kurtosis, 5: hyperskewness, 6: hypertailedness...



In general, k^{th} order (raw) moment of population:

$\frac{1}{n}\sum_i x_i^k$

Completeness

OK, but what does this have to do with atomic descriptors?
"For a distribution of mass or probability on a bounded interval, the collection of all the moments (of all orders, from 0 to ∞) uniquely determines the distribution"
Sounds promising...

Natural extension of statistical moments to atoms is simply
$m_{prq} = \sum_i x_i^p y_i^q z_i^r$,
where the order is $p + q + r$.
1st order moments are $\left(\sum_i x_i, \sum_i y_i, \sum_i z_i\right)$ i.e. the coordinate-wise sum of all position vectors.
2nd order moments are:

$$\begin{bmatrix} \sum_i x_i^2 & \sum_i x_i y_i & \sum_i x_i z_i \\ \sum_i y_i x_i & \sum_i y_i^2 & \sum_i y_i z_i \\ \sum_i z_i x_i & \sum_i z_i y_i & \sum_i z_i^2 \end{bmatrix}$$

...otherwise known as the inertial matrix.

Geometric moments

This can be generalised to any function in Hilbert space, f, expanded in a polynomial basis
$m_{pqr} = \int f(x,y,z) x^p y^q z^r d\Omega$,
or if you prefer bra-ket notation
$m_{pqr} =< f|\Psi_{pqr} >$
(what we saw before is just special case where f is sum of delta functions.)
Typically, use f that is sum of dirac-deltas or Gaussians. This automatically gives permutational invariances provides a flexible description of the atomic positions.

$$\rho(\{r_i,\vec{w_i}\}) = \sum_i w_i \delta(\vec{r} - \vec{r}_i)$$

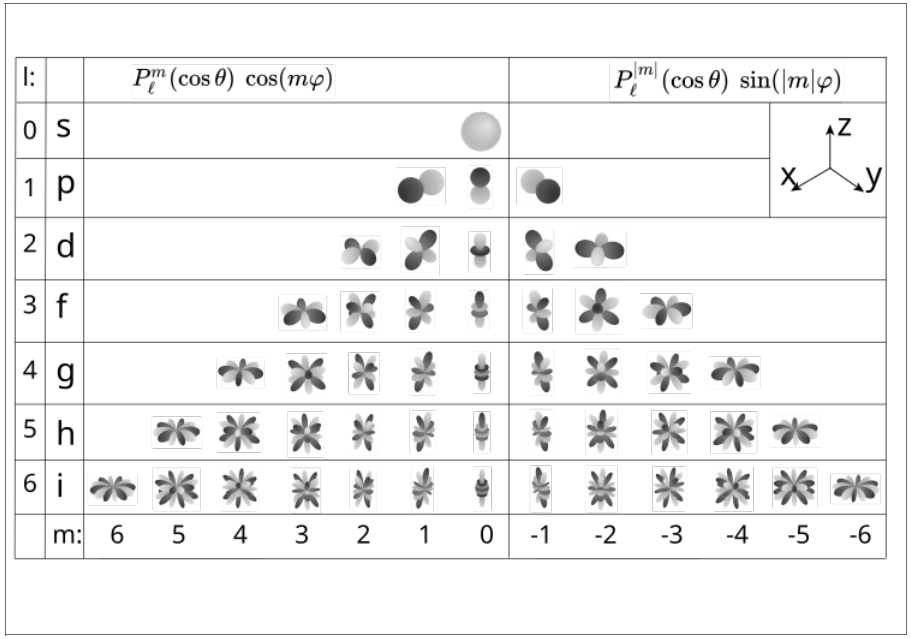
or

$$\rho(\{r_i,\vec{w_i}\}) = \sum_i w_i \mathcal{N}(\vec{r}_i, \sigma^2)$$

Spherical harmonics

While geometric moments are easy to work with and fast to calculate, their lack of orthogonality makes them awkward for use in reconstruction.

Instead, we choose to expand in spherical harmonics.



These *are* orthogonal (+bounded) and, if we add an orthogonal radial basis, ideal for reconstruction.

Reconstructions from spherical harmonics

We can instead, calculate moments (or expansion coefficients) in spherical harmonic basis with some radial function $R_{nl}(r)$:
$c_{nl}^m = \int_0^{2\pi} \int_0^{\pi} \int_0^{\infty} R_{nl}(r) \overline{Y_l^m(\theta, \varphi)} f(r, \theta, \varphi) r^2 \sin\theta dr d\theta d\varphi,$
in which case, orthogonality allows us to perform synthesis,
$\tilde{f}\left(r, \theta, \phi\right) = \sum_n^N \sum_l^n \sum_{m=-l}^l c_{nl}^m R_{nl}(r) Y_l^m(\theta, \phi),$
which gives us back (an approximation to) our original function.

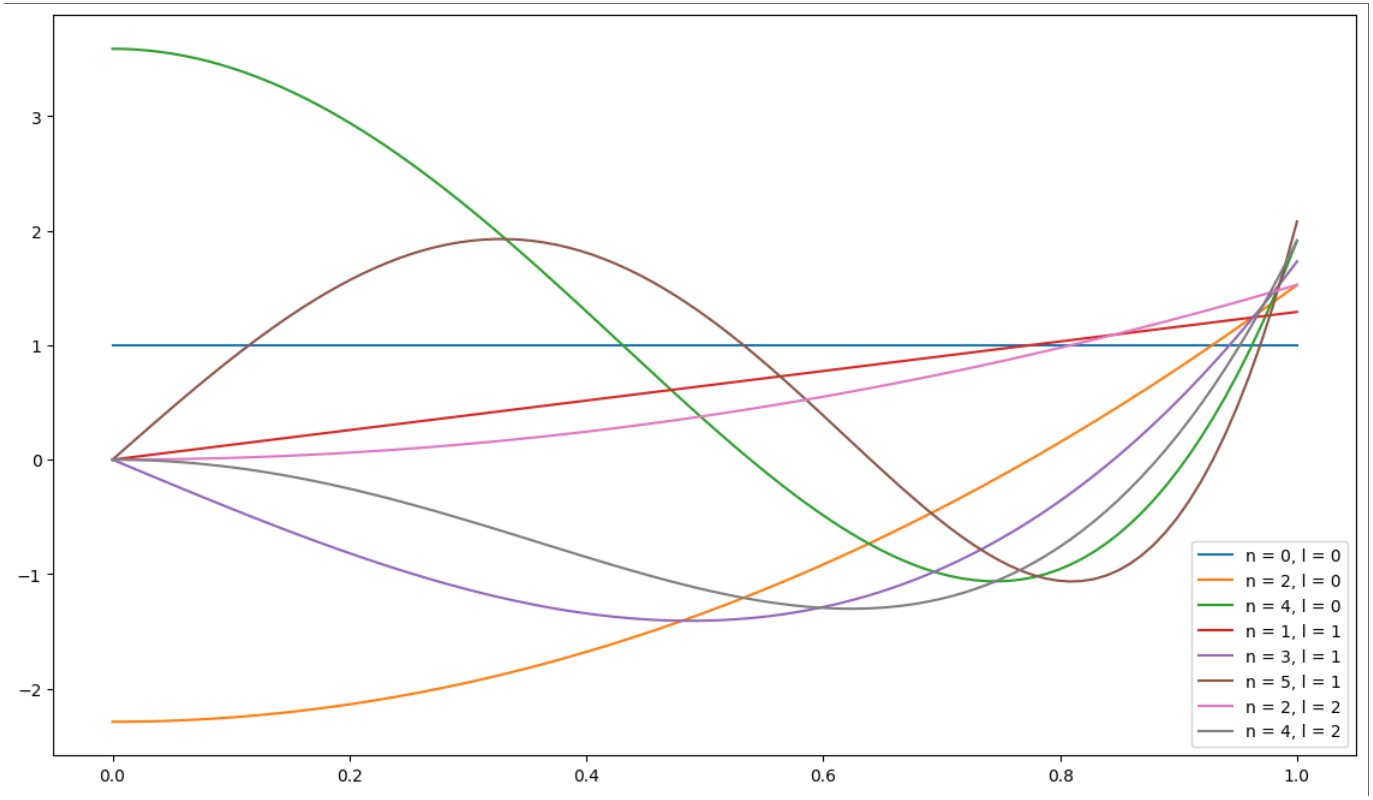
Many choicese of radial function possible, but I will use primarily Zernike:

In [35]:

```
plt.figure(figsize=(14, 8))
r = np.linspace(0, 1, 100)
for l in range(3):
    for n in range(l, 6, 2):
        plt.plot(r, zernike.r_nl(n, l, r), label=f'n = {n}, l = {l}')
plt.legend()
```

Out[35]:

```
<matplotlib.legend.Legend at 0x72fd8c7a6b00>
```



Let's see what our molecule looks like expanded in a spherical harmonic basis

In [36]:

```
max_radius = asetools.prepare_molecule(molecule)
calc = milad.ZernikeMomentsCalculator(15)

# Calculate the moments
moments = calc(geometric.from_deltas(15, molecule.positions / (1.25 * max_radius)))
moments.visualise('plotly_widget_volume')
```

opacity

0.05

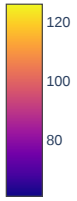
isomin

60.60

isomax

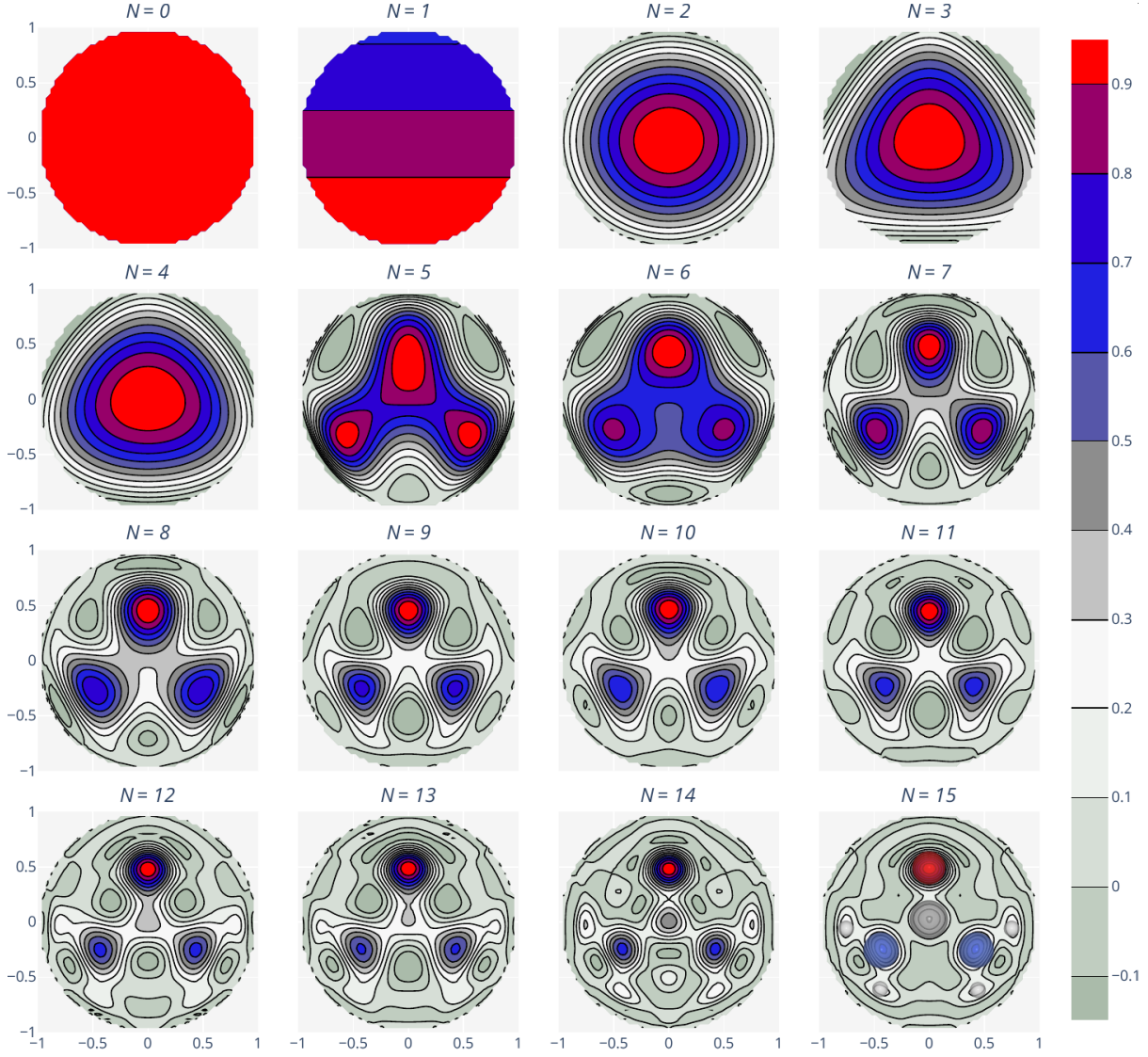
126.20

Out[36]:



Reconstructing urea

Species-weighted dirac-deltas expanded in SPH basis with Zernike radial functions



Invariants from spherical harmonics

In [37]:

```
def calc_zernike_moments(pos):
    order = 7
    calc = milad.ZernikeMomentsCalculator(order)
    moments = calc(milad.geometric.from_deltas(order, pos / (1.25 * max_radius)))
    return moments.array.real[order,:,tuple(utils.inclusive(-order, order))]

show_array(molecule.positions, calc_zernike_moments)
```

xrot

0.00

xtrans

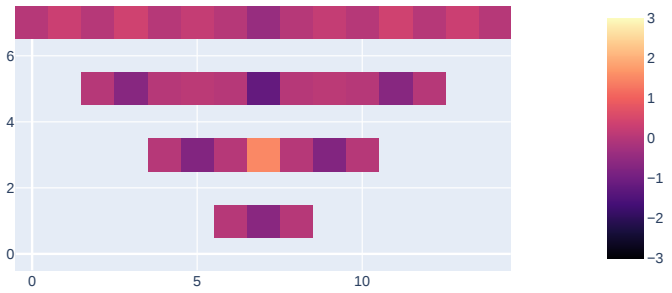
0.00

swap

0

```
[[ 0.     0.     1.339]
 [ 0.     0.     0.113]
 [ 0.     1.285 -0.683]
 [ 0.    -1.285 -0.683]
 [ 0.     2.135  0.   ]
 [ 0.    -2.135  0.   ]
 [-0.881  1.332 -1.331]
 [ 0.881  1.332 -1.331]
 [ 0.881 -1.332 -1.331]
 [-0.881 -1.332 -1.331]]
```

Out[37]:



Invariants from tensor products

In [38]:

```
def calc_rot_mtx(pos):
    pos = torch.tensor(pos, dtype=torch.float32)
    x = torch.tensor([1., 0, 0]); y = torch.tensor([0, 1., 0]);  z = torch.tensor([0., 0, 1.])
    irreps = e3nn.io.SphericalTensor(5, 1, -1)
    # Calculate Wigner-D from Euler angles
    D = irreps.D_from_angles(torch.dot(pos[0], x), torch.dot(pos[0], y), torch.dot(pos[0], z))
    return D.detach().numpy()

show_array(molecule.positions, calc_rot_mtx)
```

xrot

73.00

xtrans

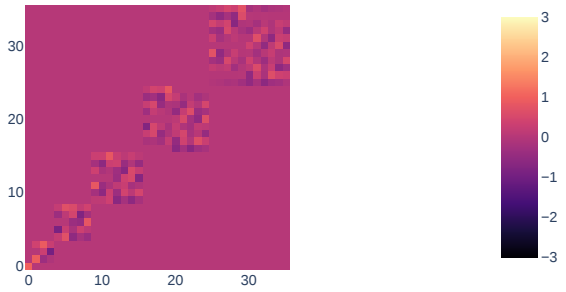
0.00

swap

0

```
[[ 0.    -1.281  0.392]
 [ 0.    -0.108  0.033]
 [ 0.     1.029  1.03 ]
 [ 0.     0.277 -1.429]
 [ 0.     0.624  2.042]
 [ 0.    -0.624 -2.042]
 [-0.881  1.662  0.884]
 [ 0.881  1.662  0.884]
 [ 0.881  0.883 -1.662]
 [-0.881  0.883 -1.662]]
```

Out[38]:



Invariants from tensor products of irreducible representations

Degree-1 invariants

c_{n0}^0 are invariants corresponding to $\mathbf{D}^0$, where n is even.

Degree-2 invariants

Invariants from a single tensor product $(\mathbf{D}^l \otimes \mathbf{D}^l)^0$

Degree-3 invariants

Invariants from two tensor products $((\mathbf{D}^{l_1} \otimes \mathbf{D}^{l_2})^l \otimes \mathbf{D}^l)^0$

Degree-4 invariants

Invariants from three tensor products $((\mathbf{D}^{l_1} \otimes \mathbf{D}^{l_2})^l \otimes (\mathbf{D}^{l_3} \otimes \mathbf{D}^{l_4})^l)^0$

In [39]:

```
def calc_invariants(pos):
    moms = milad.zernike.from_deltas(invs.max_order, pos / (1.25 * max_radius))
    return invs[:50](moms).reshape(-1, 1)

show_array(molecule.positions, calc_invariants)
```

xrot

0.00

xtrans

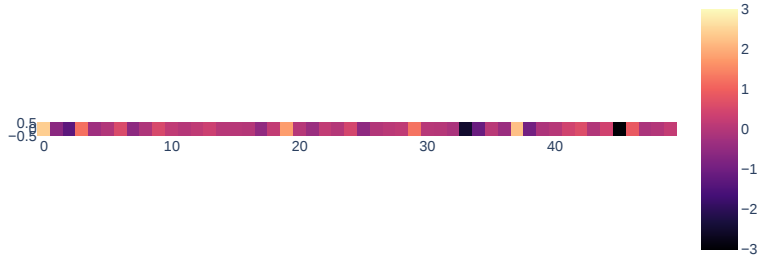
0.00

swap

0

```
[[ 0.     0.     1.339]
 [ 0.     0.     0.113]
 [ 0.     1.285 -0.683]
 [ 0.    -1.285 -0.683]
 [ 0.     2.135  0.   ]
 [ 0.    -2.135  0.   ]
 [-0.881  1.332 -1.331]
 [ 0.881  1.332 -1.331]
 [ 0.881 -1.332 -1.331]
 [-0.881 -1.332 -1.331]]
```

Out[39]:

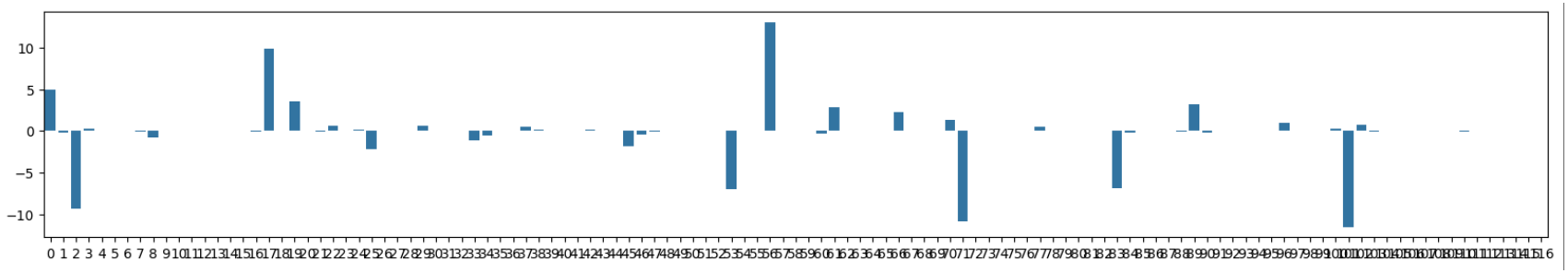


In [40]:

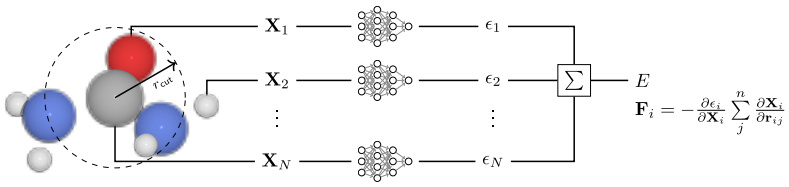
```
cutoff = 5.
descriptor = milad.descriptor(
        species={'map': {'numbers': tuple(species), 'range': (1., 6.)}},
        features={'type': functions.WeightedDelta, 'map_species_to': 'WEIGHT'},
        cutoff=cutoff,
        invs=invs,
        apply_cutoff=False,
)
fingerprint = descriptor(asetools.ase2milad(molecule))
plt.figure(figsize=(20, 3))
sns.barplot(x=jnp.arange(len(fingerprint)), y=fingerprint)
```

Out[40]:

```
<Axes: >
```



Predicting energies



Train neural network to minimise predicted RMSD of energies and forces with respect to training data.

	Training point	Test points	r_{cut} (Å)
Ni	263	31	3.8
Cu	262	31	3.9
Li	241	29	4.8
Mo	194	23	5.0
Si	214	25	4.7
Ge	228	25	5.1

Y. Zuo et al., "Performance and Cost Assessment of Machine Learning Interatomic Potentials", Journal of Physical Chemistry A **124**, 731–745 (2020)

	Energy RMSD (meV/atom)					
	Ni	Cu	Li	Mo	Si	Ge
GAP	0.62	0.56	0.63	3.55	4.18	4.47
MTP	0.74	0.52	0.66	3.89	3.02	3.68
NNP	2.25	1.68	0.98	5.67	9.95	10.95
SNAP	1.17	0.87	1.13	9.06	8.06	10.96
qSNAP	1.04	1.16	0.85	3.96	6.28	10.55
3-body ACE	1.74	1.19	1.23	4.00	5.16	11.62
MILAD	1.39	0.96	0.64	5.79	5.65	5.47
Empirical potentials						
EAM	8.51	7.46	368.64	67.98	-	-
MEAM	23.04	10.49	-	36.42	111.67	-
Tersoff	-	-	-	-	202.37	550.72

Equivariant machine learning models

In [41]:

```
def calc_rot_mtx(pos):
    pos = torch.tensor(pos, dtype=torch.float32)
    x = torch.tensor([1., 0, 0]); y = torch.tensor([0, 1., 0]);  z = torch.tensor([0., 0, 1.])
    irreps = e3nn.io.SphericalTensor(5, 1, -1)
    # Calculate Wigner-D from Euler angles
    D = irreps.D_from_angles(torch.dot(pos[0], x), torch.dot(pos[0], y), torch.dot(pos[0], z))
    return D.detach().numpy()

show_array(molecule.positions, calc_rot_mtx)
```

xrot

0.00

xtrans

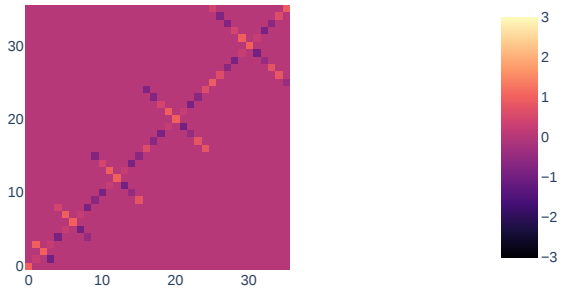
0.00

swap

0

```
[[ 0.     0.     1.339]
 [ 0.     0.     0.113]
 [ 0.     1.285 -0.683]
 [ 0.    -1.285 -0.683]
 [ 0.     2.135  0.   ]
 [ 0.    -2.135  0.   ]
 [-0.881  1.332 -1.331]
 [ 0.881  1.332 -1.331]
 [ 0.881 -1.332 -1.331]
 [-0.881 -1.332 -1.331]]
```

Out[41]:



Is there not something we can do with all the rest of the data?

Yes! That's where equivariant models come in.
So, what is equivariant?

Invariance

$$f_\theta(D(g)x) = f_\theta(x), \forall g \in G$$

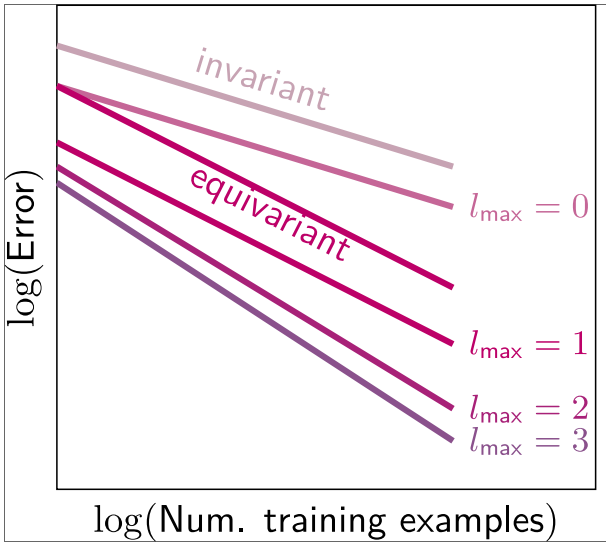
where g is a group element in group G. For rotations/reflections $G = O(3)$.

Equivariance

$$f_\theta(D_X(g)x) = D_Y(g)f_\theta(x), \forall g \in G$$

e.g. when we rotate x the output of f should be correspondly rotated.

- Natural way of extendin ML models to non-scalar inputs and outputs
- Also give **state-of-the-art** data efficiency



System		NequIP$^{a)}$	NequIP$^{b)}$	NequIP$^{c)}$	DeepMD
Liquid Water	Energy	-	1.6	1.7	1.0
	Forces	11.9	49.4	11.6	40.4
Ice Ih (b)	Energy	-	2.5	4.3	0.7
	Forces	10.2	55.8	9.9	43.3
Ice Ih (c)	Energy	-	3.9	10.2	0.7
	Forces	12.0	27.7	11.7	26.8
Ice Ih (d)	Energy	-	2.6	12.7	0.8
	Forces	9.8	23.2	9.5	25.4

NequIP model trained on $< 0.1\%$ of DeepMD model

S. Batzner et al., Nature Communications **13**, 2453 (2022)

Equivariant layer

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \otimes \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1x_2 & x_1y_2 & x_1y_2 \\ x_2x_2 & x_2y_2 & x_2y_2 \\ x_3x_2 & x_3y_2 & x_3y_2 \end{bmatrix}$$

Irreducible form:

$l = 0$ scalar - $w_1(x_1x_2 + y_1y_2 + z_1z_2)$

$l = 1$ vector - $w_2 \begin{bmatrix} y_1z_2 - z_1x_2 \\ z_1x_2 - x_1z_2 \\ x_1y_2 - x_2y_1 \end{bmatrix}$

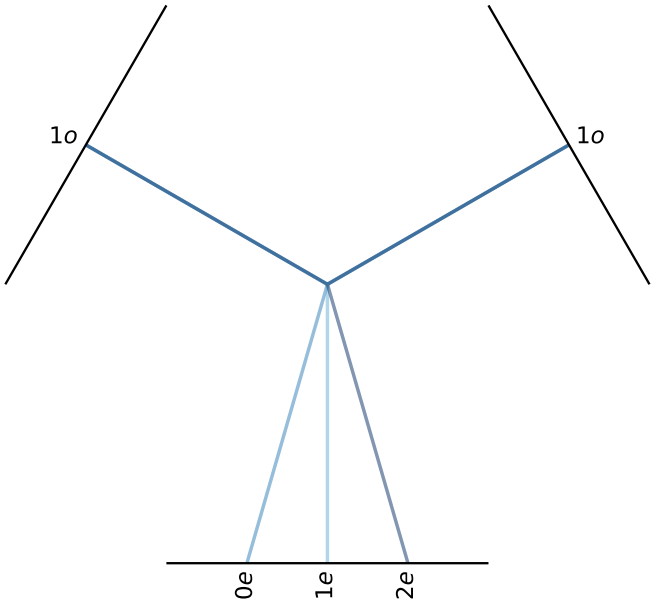
$l = 2$ vector - $w_3 \begin{bmatrix} x_1z_2 + z_1x_2 \\ x_1y_2 + y_1x_2 \\ 2y_1y_2 - x_1x_2 - z_1z_2 \\ y_1z_2 - z_1y_2 \\ z_1z_2 - x_1x_2 \end{bmatrix}$

Fully-connected layer

$$\phi\left(\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ x_2 \\ y_2 \\ z_2 \end{bmatrix}^\top \begin{bmatrix} w_1 & \\ w_2 & \\ w_3 & \cdots \\ w_4 & \\ w_5 & \\ w_6 & \end{bmatrix}\right) = \begin{bmatrix} x_3 & \cdots \end{bmatrix}$$

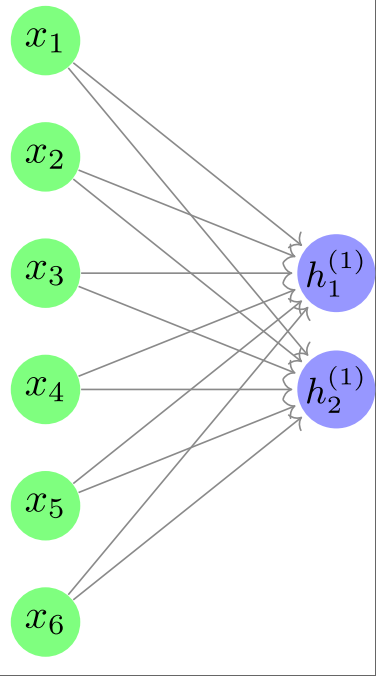
$\phi(x)$ - activation function

Equivariant layer



Outputs: $i \times 0e \oplus j \times 1e \oplus k \times 2e$

Fully-connected layer



New library for equivariant learning, `tensorial`:

- Uses `e3nn-jax` for equivariant operations
- Has graph convolution layers
- Complete flexibility over input and output types:
  - per-node, per-edge or global
  - arbitrary node attributes (scalars, tensors)
- Implemented in JAX: 4-20x speedup over pytorch

```
def f(x):
  return ** 2

# Evaluate gradient
jax.grad(f)(5.)

# Multiple inputs, scalar output
jax.grad(dft_energy(atom_positions))

# Multiple inputs, multiple outputs
jax jacfwd(get_bands(atom_positions))
```