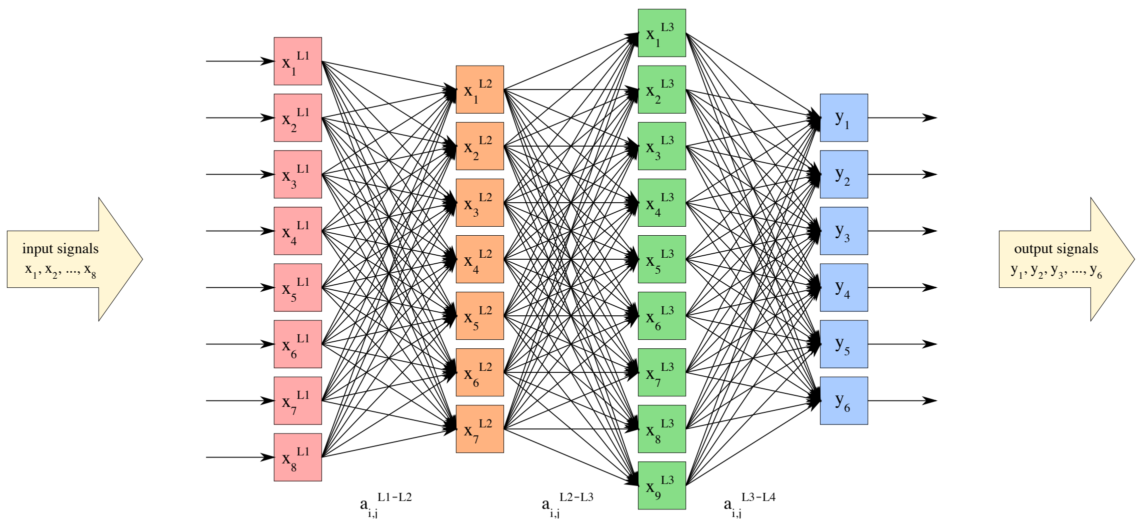
Machine Learning at CoDaS-HEP 2024, Lesson 1 Part 2

Reminders

As a reminder, a (fully-connected, feed-forward, 4-layer) neural network looks like this:



Which is to say, like this:

$$y_i = f\left(a_{i,j}^{\backslash\text{scriptsize L3-L4}} \cdot f\left(a_{i,j}^{\backslash\text{scriptsize L2-L3}} \cdot f\left(a_{i,j}^{\backslash\text{scriptsize L1-L2}} \cdot x_j + b_i^{\backslash}\right.\right.\right.$$

In code, it could be implemented as:

In [1]:
```
import numpy as np
```

In [2]:
```
# take 8-dimensional input layer 1 to 7-dimensional hidden layer 2
a_L1_L2 = np.random.normal(0, 1, (7, 8))
b_L1_L2 = np.random.normal(0, 1, (7,))

# take 7-dimensional hidden layer 2 to 9-dimensional hidden layer 3
a_L2_L3 = np.random.normal(0, 1, (9, 7))
b_L2_L3 = np.random.normal(0, 1, (9,))

# take 9-dimensional hidden layer 3 to 6-dimensional output layer 4
a_L3_L4 = np.random.normal(0, 1, (6, 9))
b_L3_L4 = np.random.normal(0, 1, (6,))

def relu(x):
    return np.maximum(0, x)

def model(x):
    layer1 = x
    layer2 = relu(a_L1_L2 @ layer1 + b_L1_L2)
    layer3 = relu(a_L2_L3 @ layer2 + b_L2_L3)
    layer4 = relu(a_L3_L4 @ layer3 + b_L3_L4)
    y = layer4
    return y
```

Here's the model's output for a sample input:

In [3]:
```
x = np.random.normal(0, 1, (8,))

model(x)
```

Out[3]:
```
array([11.37268123,  1.84277344,  0.        ,  0.        ,  9.58999636,
       10.09085982])
```

Given a large dataset of `x` vectors, an equally large set of expected `y` vectors, and a minimizer, we could train the model by optimizing these parameters:

In [4]:
```
a_L1_L2
```

```
Out[4]:  array([[-1.20024372,  1.13070243, -2.27308424,  0.95658166,  1.05589929,
                 -1.03503505,  1.45313506, -1.99747558],
                [-0.06879236, -2.19061821,  1.49560334, -1.64131217,  0.12834235,
                  1.13556172, -0.55870012,  0.2340648 ],
                [ 0.30976238, -1.45361302,  2.81257223, -0.55650381, -0.51879039,
                  1.57167208,  0.65244283,  0.24918116],
                [ 0.01518746, -1.19849445,  0.25458008,  0.5563235 , -0.04619568,
                  0.21947596, -1.48536336,  0.11960232],
                [ 0.04021875, -0.29178537, -2.17529194,  2.12121177, -0.957797  ,
                 -1.09365074, -1.15753115,  0.89950024],
                [-0.09864693,  0.57027693, -1.33417813, -1.24395602, -0.29489543,
                 -0.67746797,  0.08729895, -0.11241428],
                [-0.34604425,  0.43593279, -0.13020425, -0.07126757,  2.07834104,
                 -0.27691238,  1.16226747,  0.40691324]])
```

```
In [5]:  b_L1_L2
```

```
Out[5]:  array([ 0.04640714,  0.08560652,  0.54643672, -0.01485878,  0.60319144,
                -1.88460984,  0.14962277])
```

```
In [6]:  a_L2_L3
```

```
Out[6]:  array([[-0.44000655,  0.4642198 ,  0.18495921,  0.65838587,  0.73586247,
                 -0.24878258,  0.61473151],
                [-0.72870784, -0.50145962, -0.6313949 , -0.23895861,  1.5215994 ,
                 -1.52033205, -1.92660445],
                [ 0.58796967,  0.60791933,  0.26527056,  0.6486277 , -0.56834011,
                  0.18192105,  0.03820606],
                [-0.38277335, -0.89958337,  0.18394001,  0.4090964 ,  0.17221583,
                  0.39132635, -0.61220214],
                [-0.22748794,  0.88767062,  2.32411227,  1.64227853,  1.81894009,
                  0.75473237,  1.10648144],
                [-0.93941231, -0.0383538 , -0.86185811,  0.38591696,  0.50353627,
                 -0.67372796,  0.25013983],
                [-1.70525281,  1.02122406,  0.30789012, -1.11399033,  0.69304716,
                  0.582213  , -0.27514548],
                [ 0.68445944, -0.43769169,  0.65683234,  0.7431873 , -0.61748191,
                  0.09593288, -0.72541038],
                [-0.64753602,  1.02382729,  0.31368039,  1.05840853,  0.87968991,
                 -0.51943186,  1.35847677]])
```

```
In [7]:  b_L2_L3
```

```
Out[7]:  array([ 0.20407815,  0.74692994, -0.12640493,  0.23627184,  0.05354756,
                 0.79363438,  0.94975748, -0.55036406, -0.14419797])
```

```
In [8]:  a_L3_L4
```

```
Out[8]: array([[ 0.69422692, -0.04905967,  0.17045081,  0.38968497,  1.56146147,
                -0.72930654,  0.96979669,  0.49161953,  0.38056102],
               [-0.39738764,  0.10518261,  0.27614299,  1.20618703, -0.32065139,
                -0.09607092,  1.29380086,  0.4518837 , -1.47284257],
               [ 0.20536729, -0.86109505, -0.49936995, -0.54762952, -0.79508638,
                -0.69552127,  1.00160538,  1.25791473,  2.35391665],
               [-0.29575812,  0.25564885, -1.07649338,  0.03974225, -1.35037206,
                -0.02293308,  0.67931946, -0.28920654,  1.56753518],
               [-0.61698558,  1.1134497 , -0.16285945, -0.58126739,  0.27285873,
                 0.73681187,  0.4807764 ,  2.58603411, -0.4716092 ],
               [ 1.9980568 , -2.57272947,  0.8192133 , -1.13328357,  1.37321109,
                 1.1785371 ,  0.05660426, -0.24396036, -0.85478697]])
```

```
In [9]: b_L3_L4
```

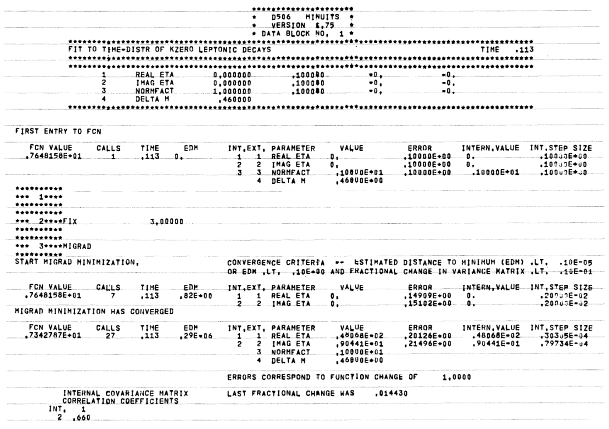
```
Out[9]: array([-0.17136499,  1.49278955, -0.38690603,  2.21870019,  0.8552186 ,
               -0.3993844 ])
```

such that `model(x)` comes as close as possible to `y`.

Then we could use `model(x_new)` to predict new y values for `x_new`, and the predictions would have (roughly) the same correlations as the training dataset.

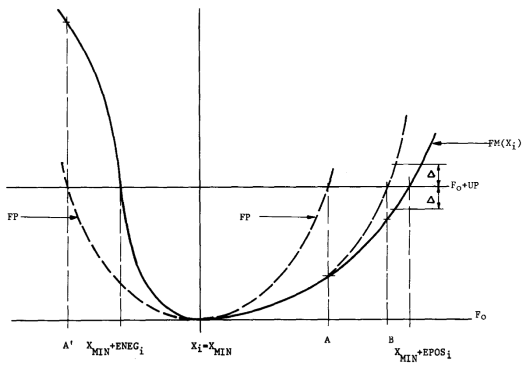
HEP has a favorite minimizer: MINUIT.

Introduced in 1972 by Fred James, MINUIT computes numerical second derivatives of the function, attempts to jump to the minimum, and then recomputes.



```
                                   **********************
                                   *   D506   MINUITS   *
                                   *   VERSION  £,75    *
                                   * DATA BLOCK NO,  1  *
         ****************************************************************************
         FIT TO TIME-DISTR OF KZERO LEPTONIC DECAYS                          TIME   .113
         ****************************************************************************
         ****************************************************************************
              1     REAL ETA     0,000000       ,100000      -0,          -0.
              2     IMAG ETA     0,000000       ,100000      -0,          -0.
              3     NORMFACT     1,000000       ,100000      -0,          -0.
              4     DELTA M       ,460000
         ****************************************************************************

FIRST ENTRY TO FCN

  FCN VALUE     CALLS    TIME     EDM      INT,EXT, PARAMETER     VALUE        ERROR      INTERN,VALUE  INT.STEP SIZE
 ,7648158E+01     1      ,113   0,           1   1  REAL ETA    0,           ,10000E+00   0.            ,10000E+00
                                             2   2  IMAG ETA    0,           ,10000E+00   0.            ,10000E+00
                                             3   3  NORMFACT     ,10000E+01  ,10000E+00    .10000E+01   ,10000E+00
                                                 4  DELTA M      ,46000E+00
**********
***  1****
**********
**********
***  2****FIX          3,00000
**********
**********
***  3****MIGRAD
**********
START MIGRAD MINIMIZATION,                 CONVERGENCE CRITERIA  --  ESTIMATED DISTANCE TO MINIMUM (EDM) .LT,  .10E-05
                                           OR EDM ,LT,  ,10E+00 AND FRACTIONAL CHANGE IN VARIANCE MATRIX .LT,  .10E-01

  FCN VALUE     CALLS    TIME     EDM      INT,EXT, PARAMETER     VALUE        ERROR      INTERN,VALUE  INT,STEP SIZE
 ,7648158E+01     7      ,113   ,82E+00      1   1  REAL ETA    0,           ,14909E+00   0.            ,20000E-02
                                             2   2  IMAG ETA    0,           ,15102E+00   0.            ,20000E-02
MIGRAD MINIMIZATION HAS CONVERGED

  FCN VALUE     CALLS    TIME     EDM      INT,EXT, PARAMETER     VALUE        ERROR      INTERN,VALUE  INT,STEP SIZE
 ,7342787E+01    27      ,113   ,29E-06      1   1  REAL ETA     ,48068E-02  ,20126E+00    .48068E-02   .30305E-04
                                             2   2  IMAG ETA     ,90441E-01  ,21496E+00    .90441E-01   .79734E-04
                                                 3  NORMFACT     ,10000E+01
                                                 4  DELTA M      ,46000E+00

                                           ERRORS CORRESPOND TO FUNCTION CHANGE OF     1,0000

        INTERNAL COVARIANCE MATRIX         LAST FRACTIONAL CHANGE WAS    ,014430
        CORRELATION COEFFICIENTS
      INT,  1
        2  ,660
```

F. James, M. Roos/MINUIT



It doesn't scale well with a large number of parameters to optimize, and we would have

```
In [10]: a_L1_L2.size + b_L1_L2.size + a_L2_L3.size + b_L2_L3.size + a_L3_L4.size + b
```

```
Out[10]: 195
```

parameters to optimize in this simple example.

Nevertheless, I'll use MINUIT for some early examples, through the excellent iminuit package.

```
In [11]: import iminuit
```

As another simplification, note that we don't have to maintain the distinction between matrices of parameters $a_{i,j}$ and vectors of parameters b_i:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,10} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,10} \\ a_{3,1} & a_{3,2} & \cdots & a_{3,10} \\ a_{4,1} & a_{4,2} & \cdots & a_{4,10} \\ a_{5,1} & a_{5,2} & \cdots & a_{5,10} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{10} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}$$

is the same as

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,10} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,10} & b_2 \\ a_{3,1} & a_{3,2} & \cdots & a_{3,10} & b_3 \\ a_{4,1} & a_{4,2} & \cdots & a_{4,10} & b_4 \\ a_{5,1} & a_{5,2} & \cdots & a_{5,10} & b_5 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{10} \\ 1 \end{pmatrix}$$

We can absorb our b_i vectors into a bigger matrix $A_{i,j}$ with the understanding that we concatenate a 1 at the end of the x_j vector.

What's so special about this linear-nonlinear sandwich?

The goal of curve-fitting is to *approximate a function* from noisy samples.

Neural networks are special because they are exceptionally good at approximating functions, a fact that is formally expressed as the universal approximation theorem(s).

As a physicist, I've approximated a few functions in my time. What's "exceptionally good" about this method?

Demonstrate with a sample problem

Suppose x and y are related as

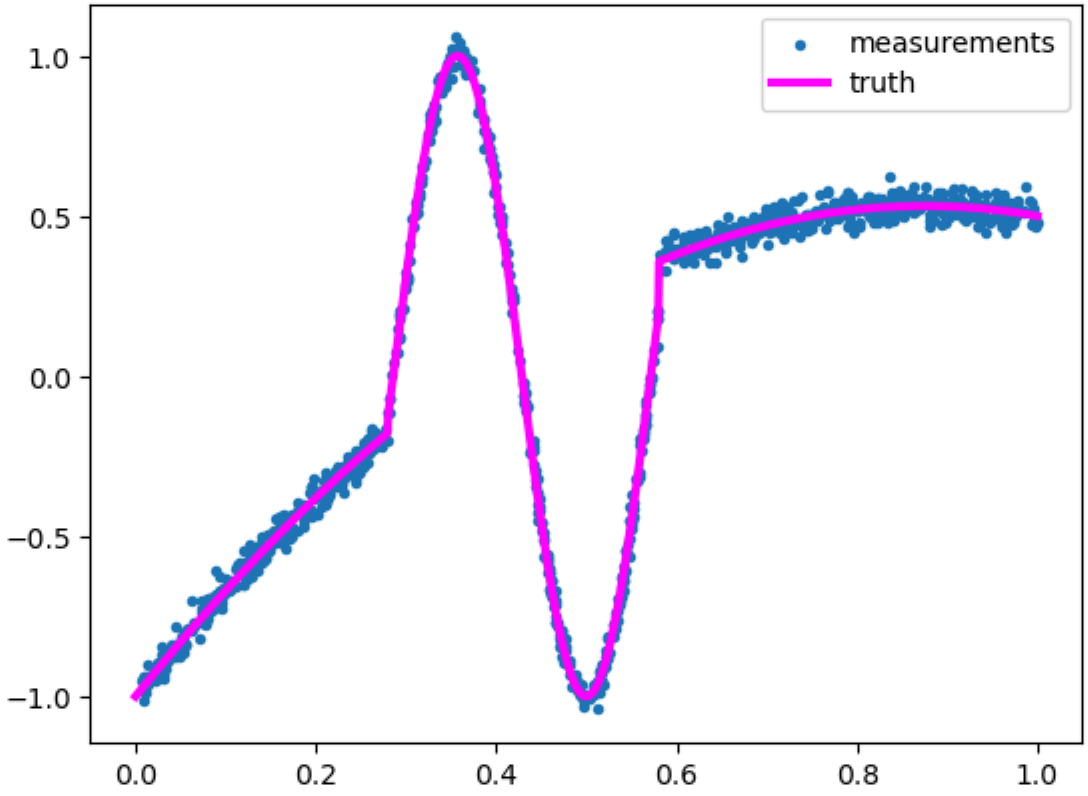
$$y = \begin{cases} \sin(22x) & \text{if } |x - 0.43| < 0.15 \\ -1 + 3.5x - 2x^2 & \text{otherwise} \end{cases}$$

with small errors.

```
In [12]: def truth(x):
             return np.where(abs(x - 0.43) < 0.15, np.sin(22*x), -1 + 3.5*x - 2*x**2)

         x = np.random.uniform(0, 1, 1000)
         y = truth(x) + np.random.normal(0, 0.03, 1000)
```

In [13]:
```python
import matplotlib.pyplot as plt
```

In [14]:
```python
fig, ax = plt.subplots()

ax.scatter(x, y, marker=".")
ax.plot(np.linspace(0, 1, 1000), truth(np.linspace(0, 1, 1000)), color="mage

ax.legend(["measurements", "truth"])

None
```



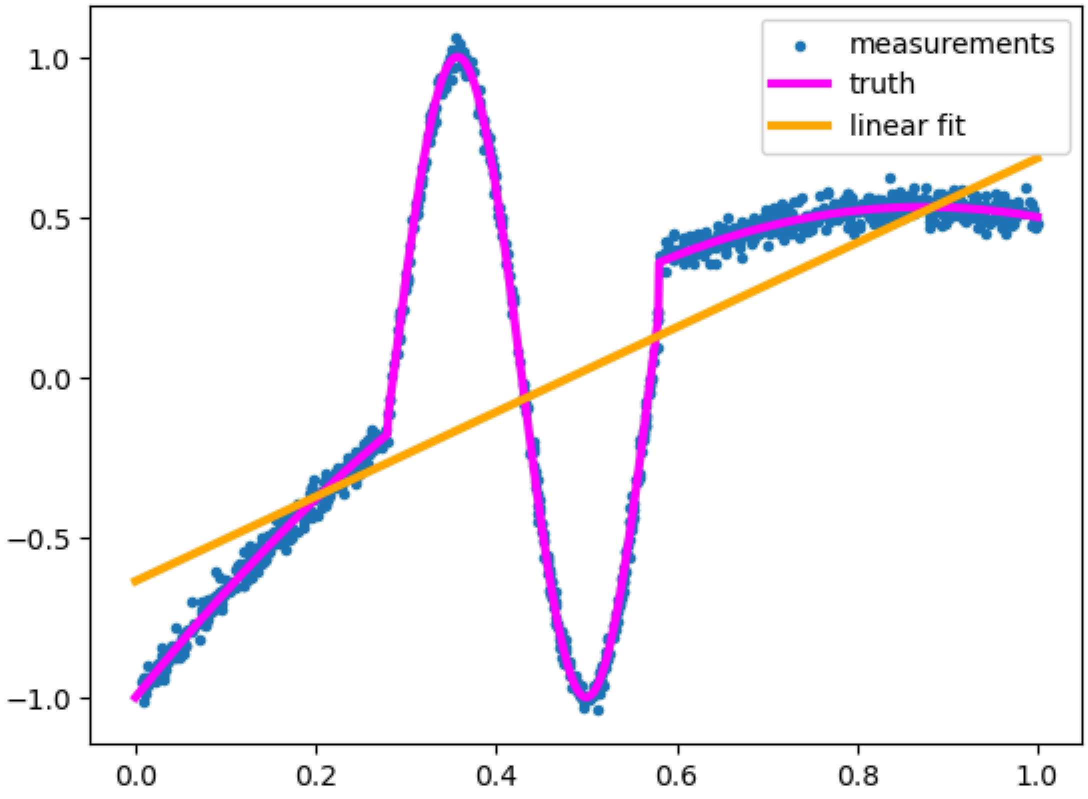
Attempt 1: a linear fit

A linear fit is terrible because the curve isn't close to being linear.

In [15]:
```python
# a linear fit can be computed analytically, which is nice
sum1 = len(x)
```

```
sumx = np.sum(x)
sumy = np.sum(y)
sumxx = np.sum(x**2)
sumxy = np.sum(x * y)
delta = (sum1 * sumxx) - (sumx * sumx)

slope = ((sum1 * sumxy) - (sumx * sumy)) / delta
intercept = ((sumxx * sumy) - (sumx * sumxy)) / delta

model_x = np.linspace(0, 1, 1000)
model_y = slope * model_x + intercept
```

In [16]:
```
fig, ax = plt.subplots()

ax.scatter(x, y, marker=".")
ax.plot(np.linspace(0, 1, 1000), truth(np.linspace(0, 1, 1000)), color="mage
ax.plot(model_x, model_y, color="orange", linewidth=3)

ax.legend(["measurements", "truth", "linear fit"])

None
```



Attempt 2: a theory-driven ansatz

A physicist's usual strategy is to find the underlying theory, with configurable parameters for the unknown values.

This parameterizable function is called an "ansatz".

Suppose we *just know* that the functional form is

$$y = \begin{cases} \sin(Cx) & \text{if } |x - A| < B \\ D + Ex + Fx^2 & \text{otherwise} \end{cases}$$

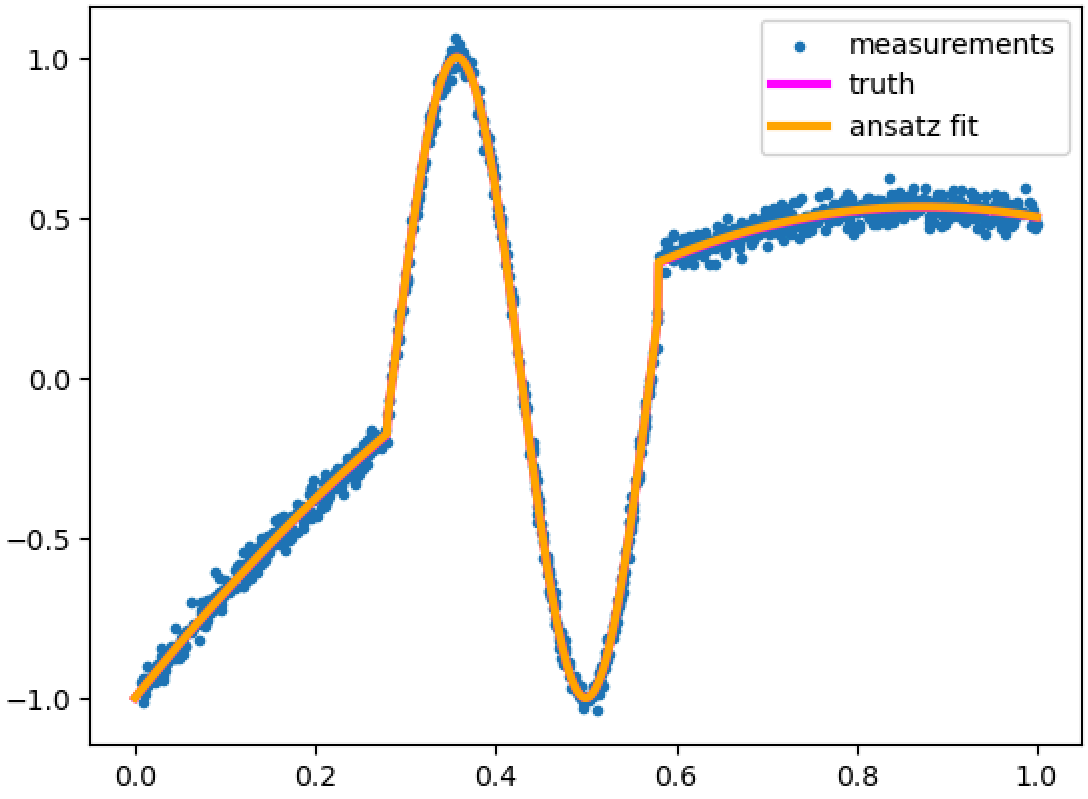
for some A, B, C, D, E, F.

In [17]:
```
def ansatz(x, A, B, C, D, E, F):
    return np.where(abs(x - A) < B, np.sin(C*x), D + E*x + F*x**2)
```

In [18]:
```
from iminuit.cost import LeastSquares
```

In [19]:
```
# define a loss function that is minimized when the parameterized ansatz is
least_squares = LeastSquares(x, y, 0.03, ansatz)

# set initial parameter values
minimizer = iminuit.Minuit(least_squares, A=0.43, B=0.15, C=22, D=-1, E=3.5,
minimizer.migrad()

model_x = np.linspace(0, 1, 1000)
model_y = ansatz(model_x, **{p.name: p.value for p in minimizer.params})
```

In [20]:
```
fig, ax = plt.subplots()

ax.scatter(x, y, marker=".")
ax.plot(np.linspace(0, 1, 1000), truth(np.linspace(0, 1, 1000)), color="mage
ax.plot(model_x, model_y, color="orange", linewidth=3)

ax.legend(["measurements", "truth", "ansatz fit"])

None
```



It's a great fit, but

- what if we don't know the functional form? or if it's super-complicated, like human behavior?
- the fit depends sensitively on the initial parameters and step size (try starting any of the parameters at the wrong value).

Minuit was designed as an interactive program so that users could hand-tweak their fits.

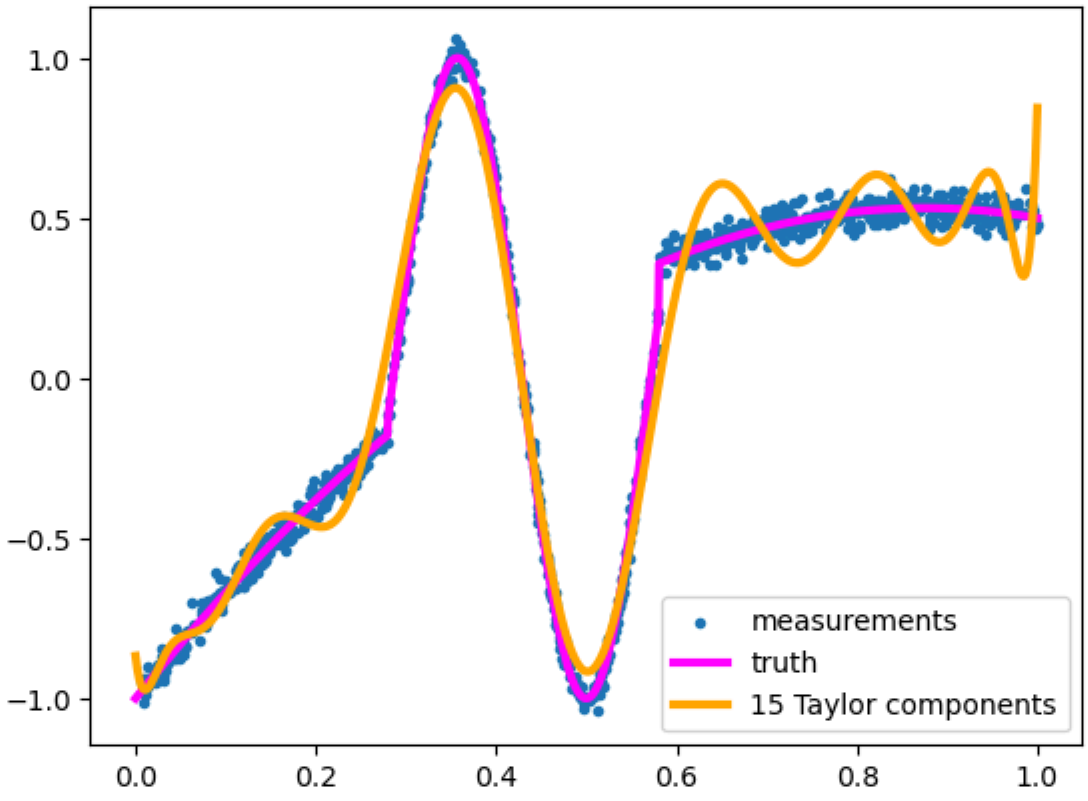
How many of you ever had to tinker with a fit until it converged?

Attempt 3: orthonormal basis functions

As physicists, we would approach a *generic* unknown function with a Taylor series, a Fourier series, or other sum of orthonormal basis functions (Jacobi, Laguerre, Hermite, Chebyshev, ...).

In [21]:
```
NUMBER_OF_POLYNOMIAL_TERMS = 15

# NumPy has a function for polynomial fits (which is analytic because
# it can be rewritten as a linear fit in the polynomial coefficients)
coefficients = np.polyfit(x, y, NUMBER_OF_POLYNOMIAL_TERMS - 1)[::-1]

model_x = np.linspace(0, 1, 1000)
model_y = sum(c * model_x**i for i, c in enumerate(coefficients))
```

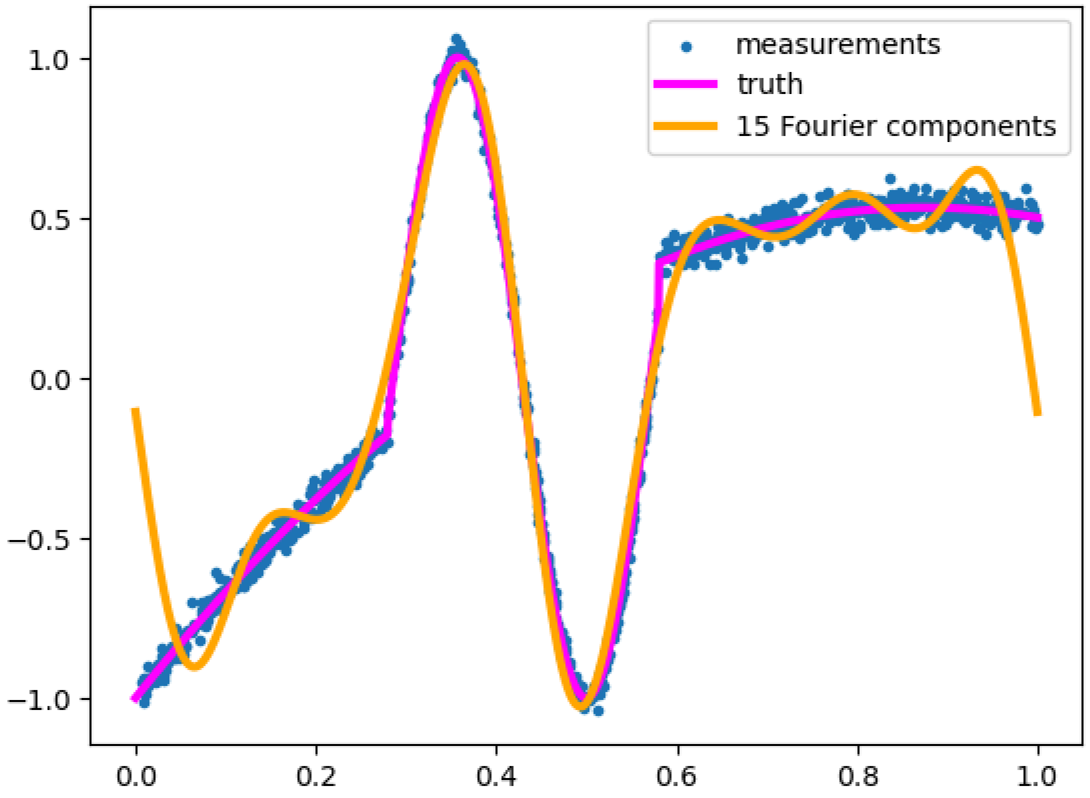
In [22]:
```
fig, ax = plt.subplots()

ax.scatter(x, y, marker=".")
ax.plot(np.linspace(0, 1, 1000), truth(np.linspace(0, 1, 1000)), color="mage
ax.plot(model_x, model_y, color="orange", linewidth=3)

ax.legend(["measurements", "truth", f"{len(coefficients)} Taylor components"

None
```



In [23]:
```
NUMBER_OF_COS_TERMS = 7
NUMBER_OF_SIN_TERMS = 7

# NumPy's FFT doesn't apply because the data aren't uniformly spaced, but we
# Like the linear fit and the Taylor series, this is an analytic fit.
sort_index = np.argsort(x)
x_sorted = x[sort_index]
y_sorted = y[sort_index]

constant_term = np.trapz(y_sorted, x_sorted)
cos_terms = [2*np.trapz(y_sorted * np.cos(2*np.pi * (i + 1) * x_sorted), x_s
sin_terms = [2*np.trapz(y_sorted * np.sin(2*np.pi * (i + 1) * x_sorted), x_s

model_x = np.linspace(0, 1, 1000)
model_y = (
    constant_term +
    sum(coefficient * np.cos(2*np.pi * (i + 1) * model_x) for i, coefficient
    sum(coefficient * np.sin(2*np.pi * (i + 1) * model_x) for i, coefficient
)
```

```
/var/folders/38/rnl4wm1930j6m17jk5x5xfym0000gn/T/ipykernel_31989/3882078031.
py:10: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, o
r one of the numerical integration functions in `scipy.integrate`.
  constant_term = np.trapz(y_sorted, x_sorted)
/var/folders/38/rnl4wm1930j6m17jk5x5xfym0000gn/T/ipykernel_31989/3882078031.
py:11: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, o
r one of the numerical integration functions in `scipy.integrate`.
  cos_terms = [2*np.trapz(y_sorted * np.cos(2*np.pi * (i + 1) * x_sorted), x
_sorted) for i in range(NUMBER_OF_COS_TERMS)]
/var/folders/38/rnl4wm1930j6m17jk5x5xfym0000gn/T/ipykernel_31989/3882078031.
py:12: DeprecationWarning: `trapz` is deprecated. Use `trapezoid` instead, o
r one of the numerical integration functions in `scipy.integrate`.
  sin_terms = [2*np.trapz(y_sorted * np.sin(2*np.pi * (i + 1) * x_sorted), x
_sorted) for i in range(NUMBER_OF_SIN_TERMS)]
```

In [24]:
```
fig, ax = plt.subplots()

ax.scatter(x, y, marker=".")
ax.plot(np.linspace(0, 1, 1000), truth(np.linspace(0, 1, 1000)), color="mage
ax.plot(model_x, model_y, color="orange", linewidth=3)

ax.legend(["measurements", "truth", f"{1 + len(cos_terms) + len(sin_terms)}

None
```



Since the true function is neither polynomial nor sinusoidal, many terms are needed for convergence.

When only a few terms are used, the fit has irrelevant artifacts (wiggles).

Attempt 4: adaptive basis functions

The classical methods (Taylor, Fourier, etc.) have one thing in common: they all use a fixed function ψ_i for each i:

$$f(x) = \sum_i^N c_i \psi_i(x)$$

and are only allowed to optimize the coefficients c_i in front of each function.

Suppose, instead, we had a set of functions that could also *change shape*:

$$f(x) = \sum_i^N c_i \psi(x; \alpha_i, \beta_i)$$

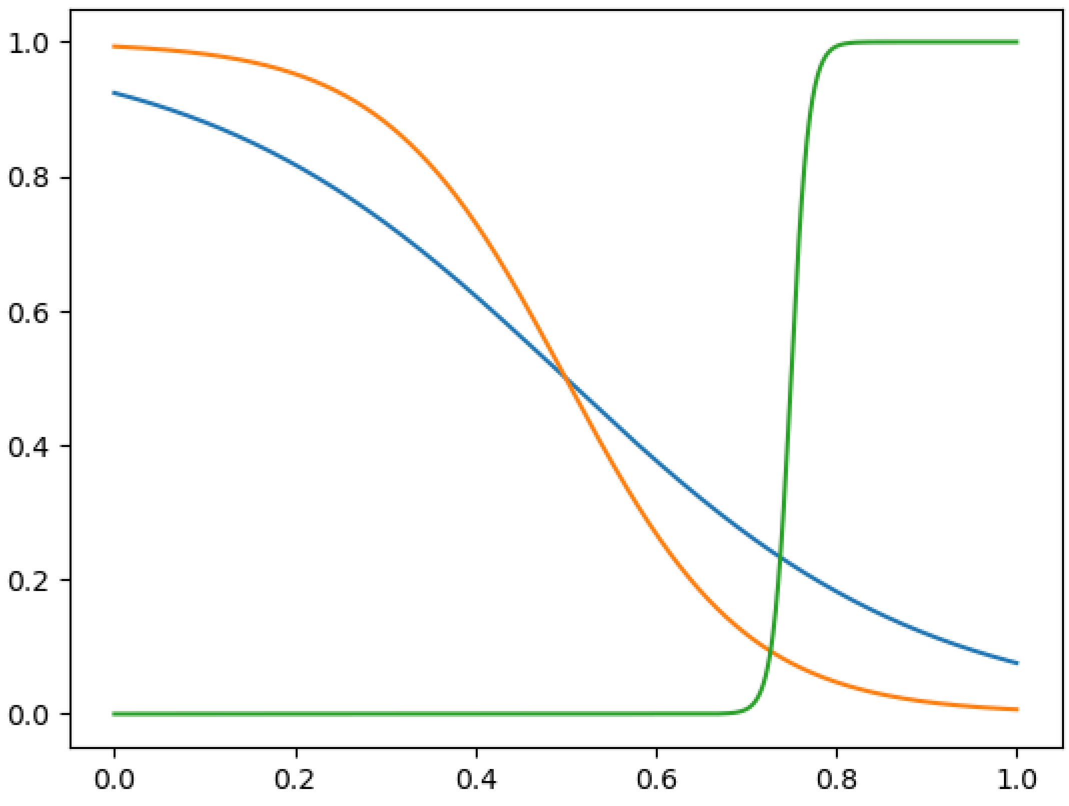
For instance, the functions are sigmoids whose center α and width β are adjustable:

$$\psi(x; \alpha, \beta) = \frac{1}{1 + \exp\big((x - \alpha)/\beta\big)}$$

In [25]:
```
def sigmoid_component(x, center, width):
    # ignore NumPy errors when Minuit explores extreme values
    with np.errstate(over="ignore", divide="ignore"):
        return 1 / (1 + np.exp((x - center) / width))
```

A few examples:

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

model_x = np.linspace(0, 1, 1000)

ax.plot(model_x, sigmoid_component(model_x, 0.5, 0.2))
ax.plot(model_x, sigmoid_component(model_x, 0.5, 0.1))
ax.plot(model_x, sigmoid_component(model_x, 0.75, -0.01))

None
```

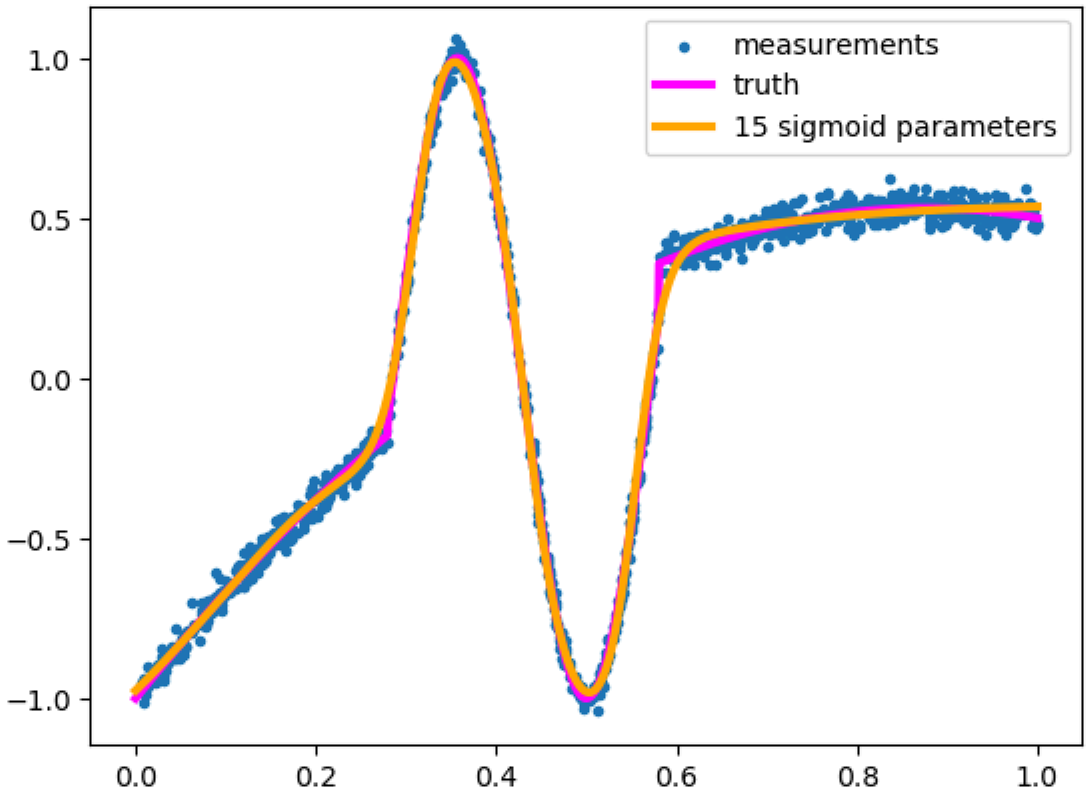


Now use the adaptive sigmoids in the fit:

In [27]:
```
NUMBER_OF_SIGMOIDS = 5

def sigmoid_sum(x, parameters):
    out = np.zeros_like(x)
    for coefficient, center, width in parameters.reshape(-1, 3):
        out += coefficient * sigmoid_component(x, center, width)
    return out

# using Minuit again
least_squares = LeastSquares(x, y, 0.03, sigmoid_sum)

# do best of 15 optimizations because this space has a lot more local minima
best_minimizer = None
for iteration in range(15):

    initial_parameters = np.zeros(5 * 3)
    initial_parameters[0::3] = np.random.normal(0, 1, NUMBER_OF_SIGMOIDS)
    initial_parameters[1::3] = np.random.uniform(0, 1, NUMBER_OF_SIGMOIDS)
    initial_parameters[2::3] = np.random.normal(0, 0.1, NUMBER_OF_SIGMOIDS)

    minimizer = iminuit.Minuit(least_squares, initial_parameters)
    minimizer.migrad()

    if best_minimizer is None or minimizer.fval < best_minimizer.fval:
        best_minimizer = minimizer

model_x = np.linspace(0, 1, 1000)
model_y = sigmoid_sum(model_x, np.array(best_minimizer.values))
```

In [28]:

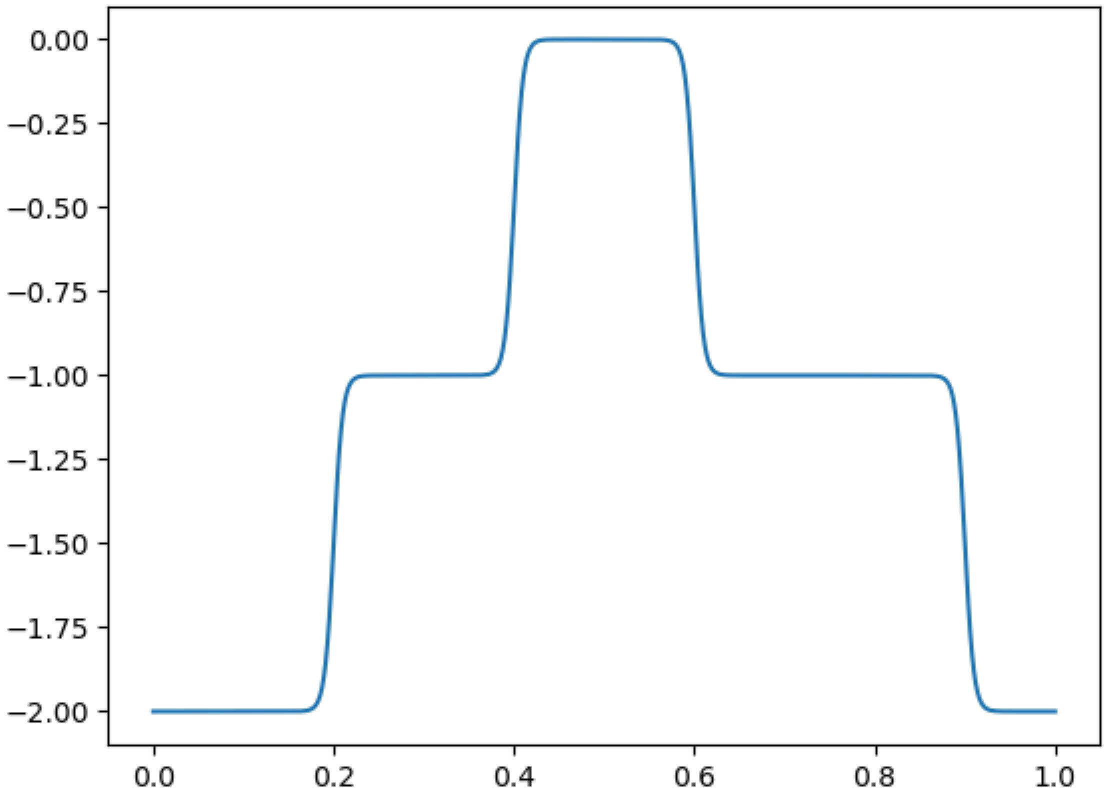
```
fig, ax = plt.subplots()

ax.scatter(x, y, marker=".")
ax.plot(np.linspace(0, 1, 1000), truth(np.linspace(0, 1, 1000)), color="mage
ax.plot(model_x, model_y, color="orange", linewidth=3)

ax.legend(["measurements", "truth", f"{len(minimizer.parameters)} sigmoid pa

None
```



The fitter doesn't need very many sigmoids because it can position each one and stretch it arbitrarily.

It can even stack them to build piecewise shapes.

In [29]:
```
fig, ax = plt.subplots()

model_x = np.linspace(0, 1, 1000)

wide_plateau_left = sigmoid_component(model_x, 0.2, 0.005)
wide_plateau_right = sigmoid_component(model_x, 0.9, -0.005)

narrow_peak_left = sigmoid_component(model_x, 0.4, 0.005)
narrow_peak_right = sigmoid_component(model_x, 0.6, -0.005)

ax.plot(model_x, -wide_plateau_left - wide_plateau_right - narrow_peak_left

None
```



The important point: adaptive basis functions *are* a neural network layer

Instead of a parameterized sigmoid,

$$\psi(x; \alpha, \beta) = \frac{1}{1 + \exp\left((x - \alpha)/\beta\right)}$$

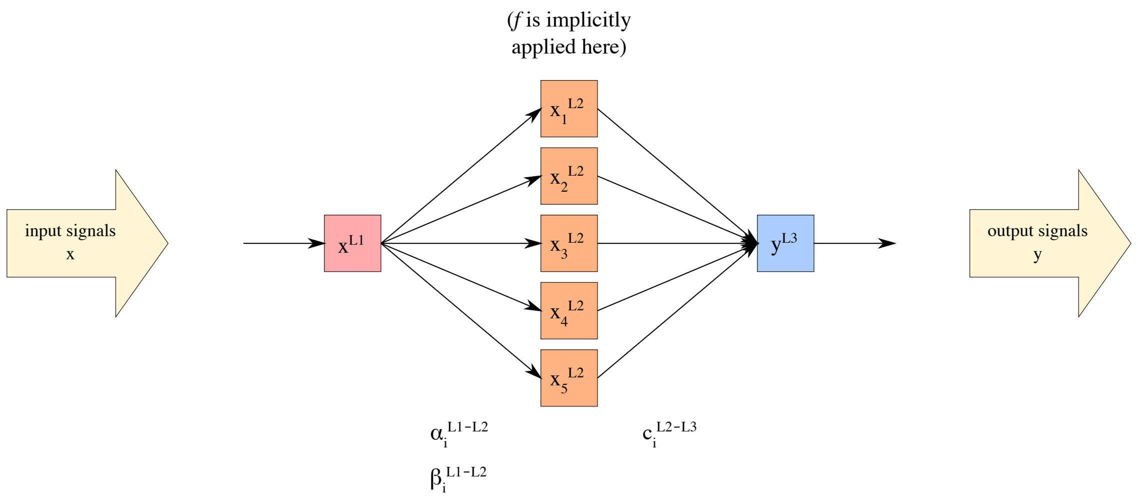
consider applying a linear transformation to the input of a sigmoid:

$$x^{\text{\scriptsize layer 2}} = \frac{x^{\text{\scriptsize layer 1}} - \alpha}{\beta} \text{\hspace\{1 cm\}and\hspace\{1 cm\}} f($$

5 independently scaled sigmoids are a single hidden layer with 5 nodes:

$$y = c_i^{\text{\scriptsize L2--L3}} \cdot f\left(\frac{x - \alpha_i^{\text{\scriptsize L1-L2}}}{\beta_i^{\text{\scriptsize L1-L2}}}\right)$$

The 5 α and 5 β parameters are the linear transformation from the input layer 1 to the hidden layer 2, the sigmoid f is the activation function applied at this layer, and the coefficients in front of each sigmoid c are the linear transformation from layer 2 to the output layer 3.

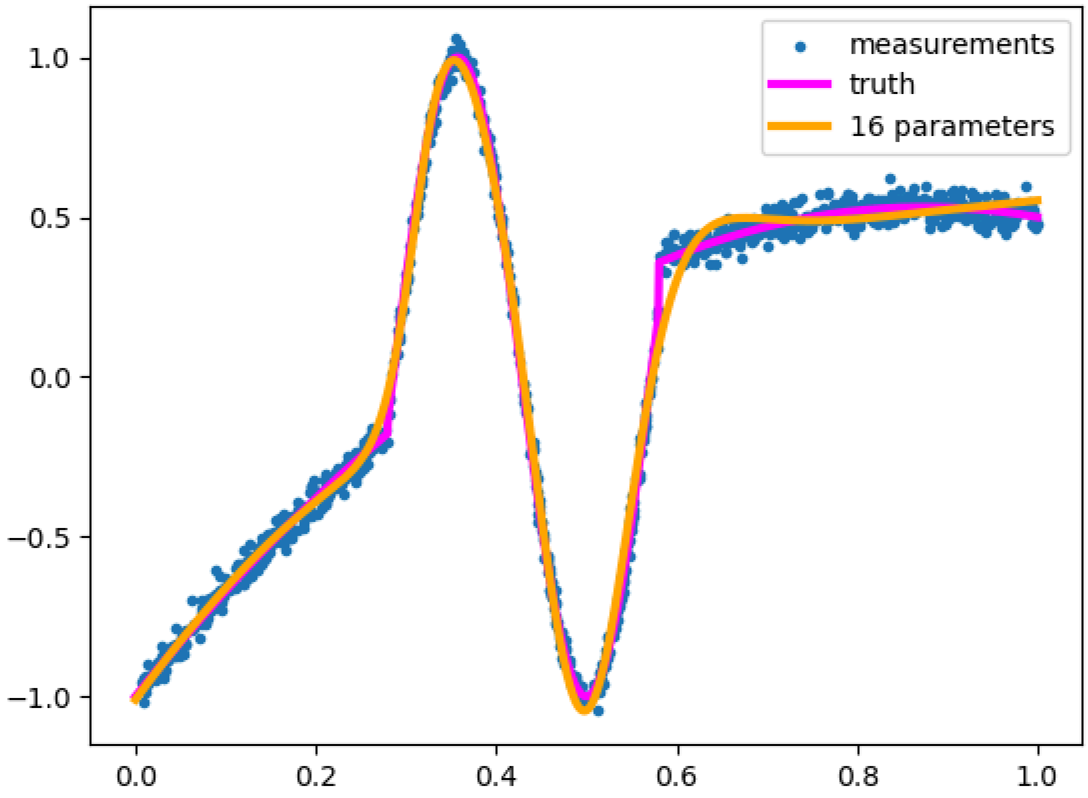


To further demonstrate this, let's use a neural network implementation from Scikit-Learn to fit the same data.

In [30]:
```
import sklearn.neural_network
```

In [31]:
```
# do best of 15 optimizations because this space has a lot of local minima
best_neural_network = None
for iteration in range(15):

    # Scikit-Learn's MLPRegressor uses ordinary least squares as a loss func
    # the "logistic" activation function is our sigmoid
    neural_network = sklearn.neural_network.MLPRegressor(
        activation="logistic", hidden_layer_sizes=(5,),
        solver="lbfgs", max_iter=10000, alpha=0,
    )

    neural_network.fit(x[:, np.newaxis], y)

    if best_neural_network is None or neural_network.loss_ < best_neural_net
        best_neural_network = neural_network

model_x = np.linspace(0, 1, 1000)
model_y = best_neural_network.predict(model_x[:, np.newaxis])
```

In [32]:
```
fig, ax = plt.subplots()

ax.scatter(x, y, marker=".")
ax.plot(np.linspace(0, 1, 1000), truth(np.linspace(0, 1, 1000)), color="mage
ax.plot(model_x, model_y, color="orange", linewidth=3)

num_params = sum(x.size for x in neural_network.coefs_) + sum(x.size for x i
ax.legend(["measurements", "truth", f"{num_params} parameters"])

None
```



(A real neural network has one more bias term per output variable than our analogy, so 16 parameters, not 15.)

Thus, a neural network with a hidden layer is a function approximator like Taylor and Fourier series, but with a special property: *the basis functions are adaptive*.

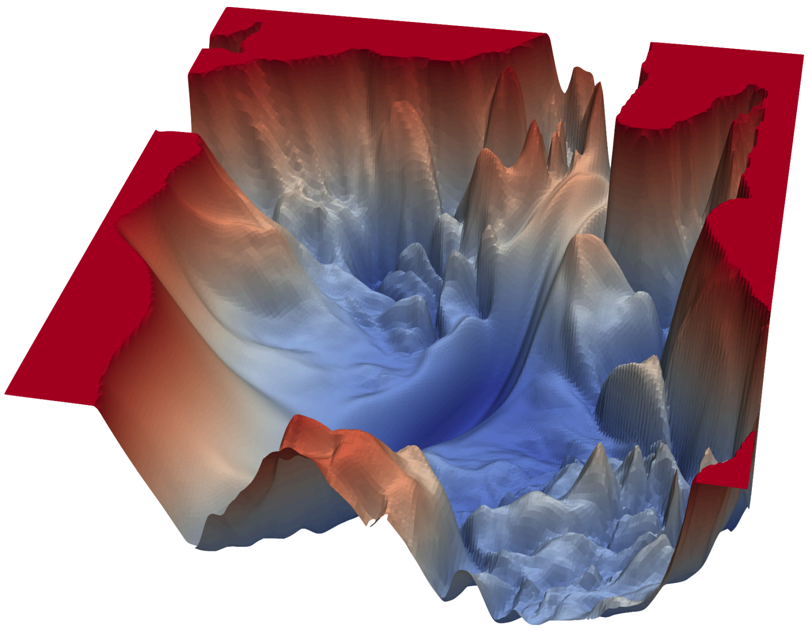
One consequence of this is that the basis functions are not orthogonal, like Taylor and Fourier series.

- Since Taylor and Fourier basis functions are orthogonal, each coefficient can be determined independently.
- Since a neural network's adaptive basis functions are not, they *must* be determined by a combined fit.

In fact, many parameters can be swapped: if $\{\alpha_i, \beta_i, c_i\} \longleftrightarrow \{\alpha_j, \beta_j, c_j\}$ for sigmoids i and j, the function output is unchanged.

- Each minimum in the optimizer's objective function has $n!$ identical minima, for each hidden layer of size n.
- Two neural networks that return the same output for all possible input could have very different internal parameters.

It's a bumpy objective function!



This is what I mean by "farming": ML doesn't eliminate all difficulties, it replaces one set of problems with another.

No description has been provided for this image No description has been provided for this image

Optimizing a neural network requires far less detailed knowledge of the function than the ansatz fit.

But it requires more attention to the network architecture and the minimizer.

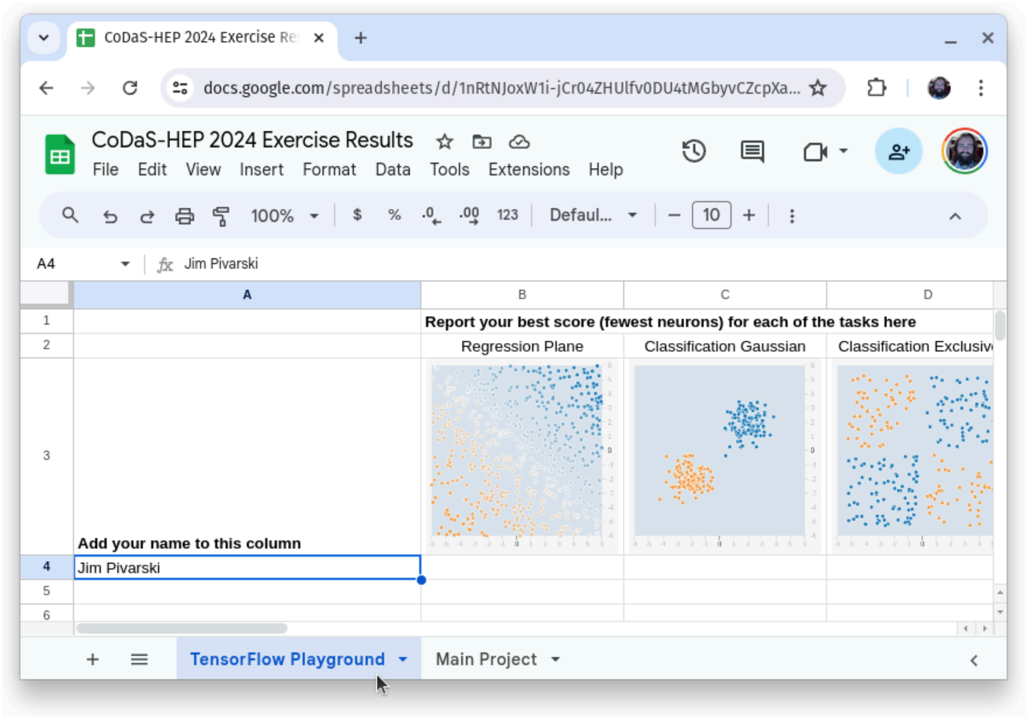
- This is just another tool; sometimes one tool is better for a particular problem, sometimes another.
- However, these "farming" problems are more generic: what computer scientists learn about minimization algorithms in general may be directly applicable to your task.

Today, the best practice for minimization is to use Adam or AdamW and try different choices of learning rate and regularization.

15 minute exercise

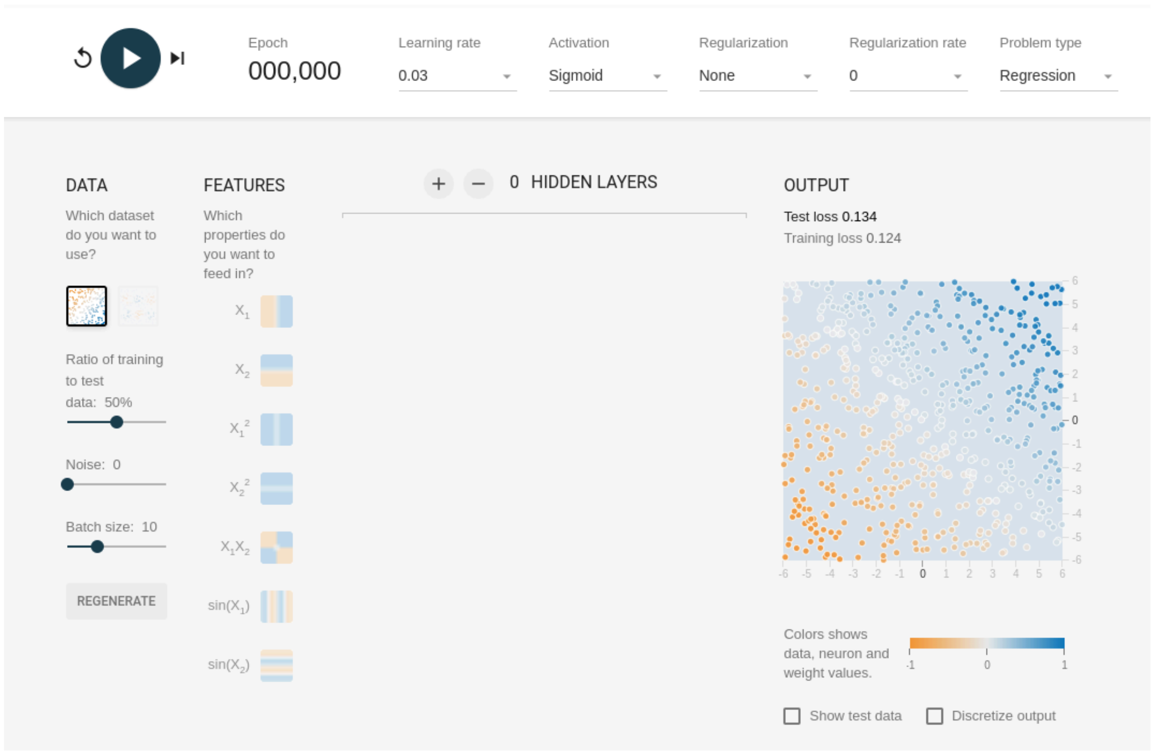
Before we start coding, let's get familiar with training neural networks in a graphical interface.

First, go to the Google Spreadsheet for this course and add your name to the *first* sheet:


CoDaS-HEP 2024 Exercise Re
docs.google.com/spreadsheets/d/1nRtNJoxW1i-jCr04ZHUlfv0DU4tMGbyvCZcpXa...
CoDaS-HEP 2024 Exercise Results
File Edit View Insert Format Data Tools Extensions Help
100%
Defaul...
10
A4
Jim Pivarski
TensorFlow Playground
Main Project

	A	B	C	D
1		Report your best score (fewest neurons) for each of the tasks here		
2		Regression Plane	Classification Gaussian	Classification Exclusiv
3	Add your name to this column			
4	Jim Pivarski			
5				
6				

Next, click on the image below to go to the TensorFlow Playground in a new tab/window.



Epoch 000,000

Learning rate 0.03

Activation Sigmoid

Regularization None

Regularization rate 0

Problem type Regression

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

Test loss 0.134

Training loss 0.124

Your job is to fit each of the training datasets with as few neurons as possible (counting each input feature as a neuron, just as hidden layers are neurons).

Be sure to press the "play" button (top left) to start optimizing.

You can switch between regression and classification problems with the "Problem type" (top right).

The "REGENERATE" button (bottom left) only regenerates the sample data, not the network weights. To reinitialize the network weights, reload the page or add and remove a layer or neuron.

If this is easy for you, can you do it with the "Noise" slider maximized (at 50)?

Outline of the rest of this mini-course

- **Introduction (1.5 hours)**
  - Craftsmanship versus farming
  - History of HEP + ML
  - Universal approximation theorem(s)
  - 15 minute playground.tensorflow.org exercise
- **Issues in Practice (2 hours)**
  - Which library?
  - Regression versus classification, loss functions
  - Optimizers: learning rate, epochs, mini-batches
  - Feature selection and the "kernel trick"
  - Under & overfitting
  - Regularization: L1, L2, dropout
  - Parameters versus hyperparameters
  - Partitioning data into training, validation, and test samples
  - Goodness of fit metrics
- **Main Project (2 hours)**
  - Classify jets into 5 categories
- **Survey of Architectures (1.5 hours)**
  - What are the building blocks?
  - Multilayer perceptron
  - Autoencoder and variational autoencoder
  - Convolutional Neural Networks (CNNs)
  - DeepSet and Graph Neural Networks (GNNs)
  - Transformers