In [1]:

```
# IE 678 Deep Learning, University of Mannheim
# Author: Rainer Gemulla
```

In [136]:

```
# Student: Timur Michael Carstensen
# Student ID: 1722194
# Date: 27.03.2022
```

In [2]:

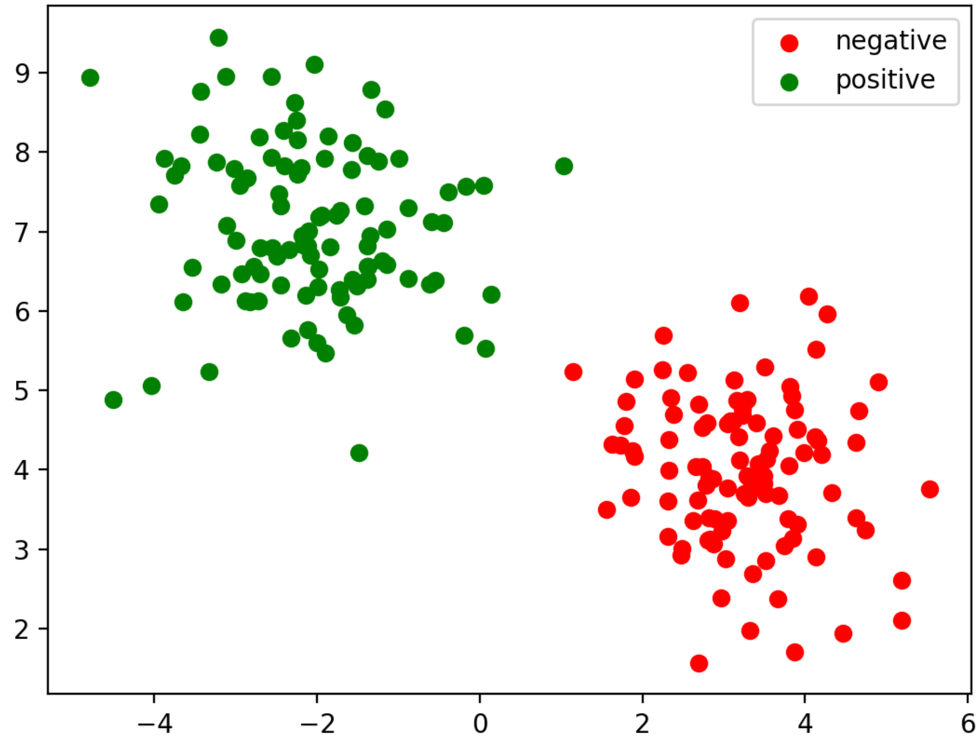
```
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F

from IPython import get_ipython
from util import nextplot
#%matplotlib inline
%matplotlib notebook
get_ipython().magic('run -i "a01-fnn-helper.py"')
```

```
/usr/local/Caskroom/miniconda/base/envs/mtp-ai-turing-tumble/lib/pytho
n3.9/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress not found.
Please update jupyter and ipywidgets. See https://ipywidgets.readthedo
cs.io/en/stable/user_install.html (https://ipywidgets.readthedocs.io/e
n/stable/user_install.html)
  from .autonotebook import tqdm as notebook_tqdm
/Users/timurcarstensen/Library/CloudStorage/OneDrive-bwedu/1. Modules/
1. Master/1. MMDS/2. Semester/IE 678 - Deep Learning/4-Assignments/ie-
678-deep-learning/dl22-a01/a01-fnn-helper.py:26: DeprecationWarning: `
np.int` is a deprecated alias for the builtin `int`. To silence this w
arning, use `int` by itself. Doing this will not modify any behavior a
nd is safe. When replacing `np.int`, you may wish to use e.g. `np.int6
4` or `np.int32` to specify the precision. If you wish to review your
 current use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://nump
y.org/devdocs/release/1.20.0-notes.html#deprecations (https://numpy.or
g/devdocs/release/1.20.0-notes.html#deprecations)
  y = np.concatenate([np.zeros(n, dtype=np.int), np.ones(n, dtype=np.i
nt)])
```

# 1 Perceptrons

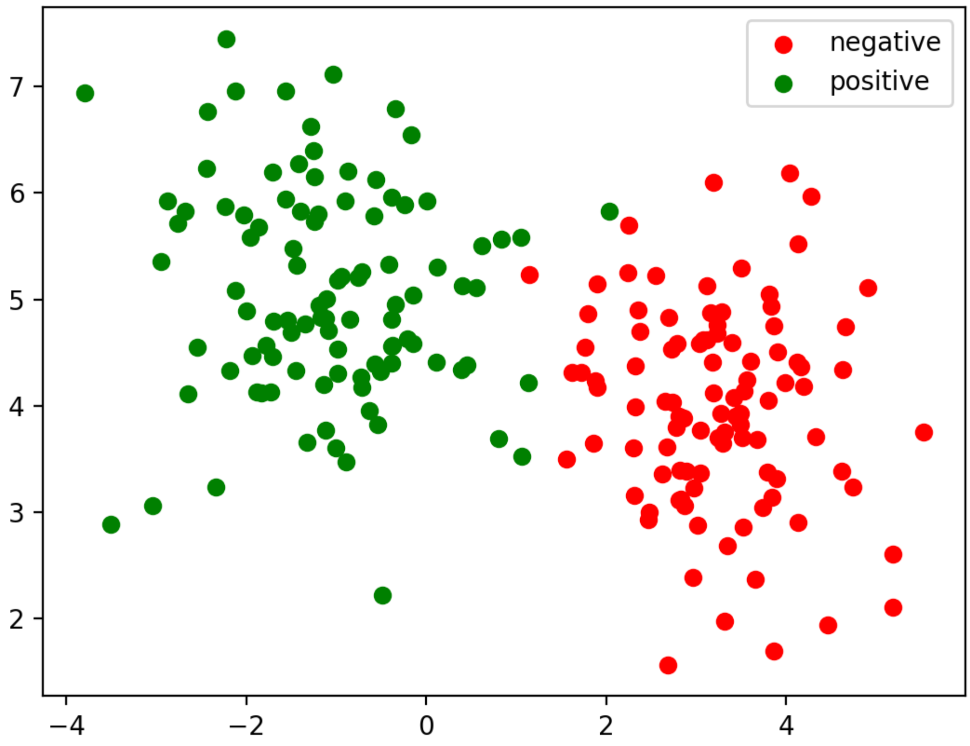
In [3]:

```
# plot X1 (separable)
nextplot()
plot2(X1, y1)
```



In [4]:

```
# plot X2 (not separable)
nextplot()
plot2(X2, y2)
```



In [48]:

```
def pt_classify(X, w):
    """Classify using a perceptron.

    Parameters
    ----------
    X : torch array of shape (N,D) or shape (D,)
        Design matrix of test examples
    w : torch array of shape (D,)
        Weight vector

    Returns
    -------
    torch array of shape (N,)
        Predicted binary labels (either 0 or 1)"""
    if X.dim() == 1:
        X = X.view(1, -1)
    return (X @ w >= 0).int()
```

# 1a+c Learning

In [44]:

```
def pt_train(X, y, maxepochs=100, pocket=False, w0=None):
    """Train a perceptron.

    Parameters
    ----------
    X : torch array of shape (N,D)
        Design matrix
    y : torch array of shape (N,)
        Binary labels (either 0 or 1)
    maxepochs : int
        Maximum number of passes through the training set before the algorithm
        returns
    pocket : bool
       Whether to use the pocket algorithm (True) or the perceptron learning algorit
       (False)
    w0 : torch array of shape (D,)
        Initial weight vector

    Returns
    -------
    torch array of shape (D,)
        Fitted weight vector"""

    N, D = X.shape
    if w0 is None:  # initial weight vector
        w0 = torch.zeros(D)
    w = w0  # current weight vector

    train = X.clone()
    train_target = y.clone()
    train_target[train_target==0.0] = -1


    total_weight_updates = 0
    total_tested_examples = 0
    total_correctly_classified_examples = 0

    pocket_weight_vector = w.clone()
    pocket_weight_vector_count = 0

    local_weight_vector = pocket_weight_vector.clone()
    local_weight_vector_count = 0

    for epoch in range(maxepochs):
        no_updates: bool = False

        if not pocket:
            no_updates: bool = True

            for i, x in enumerate(train):

                total_tested_examples += 1

                if torch.sign(w[1:]@x[1:])!= torch.sign(train_target[i]):
                    w[1:] += torch.sign(train_target[i]) * x[1:]
                    total_weight_updates += 1
                    no_updates = False
                else:
                    total_correctly_classified_examples += 1
```

```
        elif pocket:

            for i in range(N):
                r = torch.randint(high=N, size=(1,1)).item()
                rand_sample = train[r]
                rand_sample_target = train_target[r]

                total_tested_examples += 1

                if torch.sign(local_weight_vector[1:]@rand_sample[1:])!= torch.sign(
                    local_weight_vector_count = 0
                    local_weight_vector[1:] += torch.sign(rand_sample_target) * ranc
                    total_weight_updates += 1

                elif not torch.sign(local_weight_vector[1:]@rand_sample[1:])!= torch
                    local_weight_vector_count += 1
                    total_correctly_classified_examples += 1

                if local_weight_vector_count >= pocket_weight_vector_count:
                    pocket_weight_vector_count = local_weight_vector_count
                    pocket_weight_vector = local_weight_vector.clone()

        if no_updates:
                print(f"stopped training after {epoch + 1} epochs")
                break

    print(
        f"weight updates: {total_weight_updates:4}"
        f" / tested examples: {total_tested_examples:6}"
        f" / correctly classified: {total_correctly_classified_examples:6}"
        f" / incorrectly classified: {(total_tested_examples - total_correctly_class
    )


    if pocket:
        print(
            f"the best weight vector for the pocket algorithm classified "
            f"{pocket_weight_vector_count} samples correctly in a row"
        )
        return pocket_weight_vector
    elif not pocket:
        return w
```

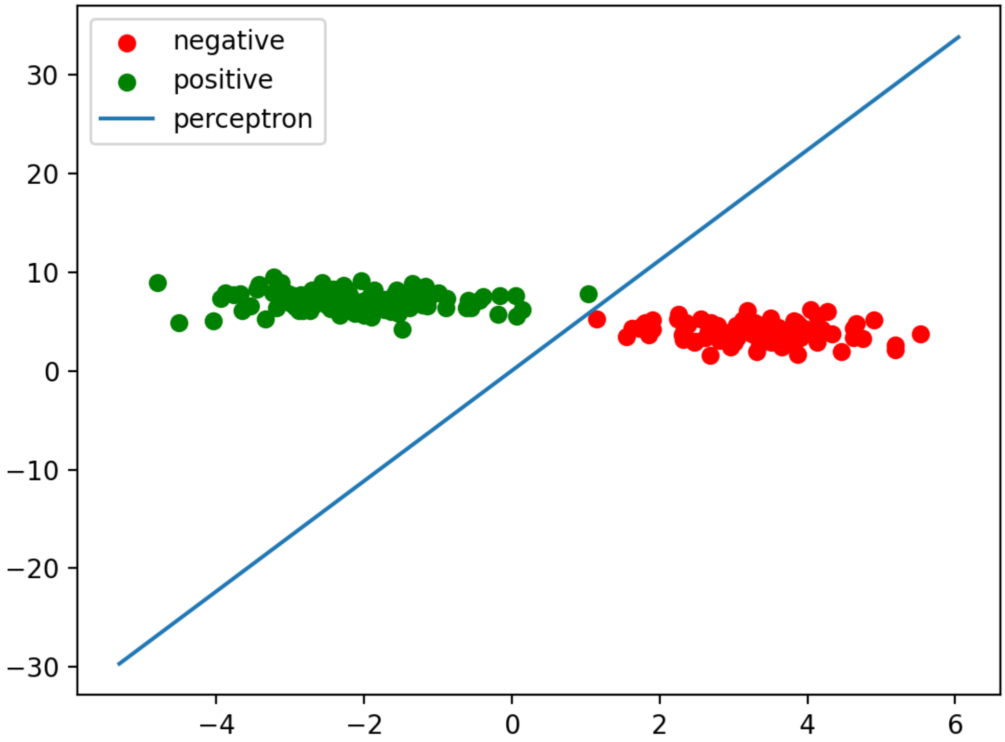
# 1b+d Experimentation

## perceptron learning algorithm on separable data

In [45]:

```
# Train a perceptron using the perceptron learning algorithm and plot decision
# boundary. You should get a perfect classification here. The decision boundary
# should not change if you run this multiple times.
w = pt_train(X1, y1)
nextplot()
plot2(X1, y1)
plot2db(w, label="perceptron")
```

```
stopped training after 4 epochs
weight updates:    5 / tested examples:    800 / correctly classified:
795 / incorrectly classified:    5
```

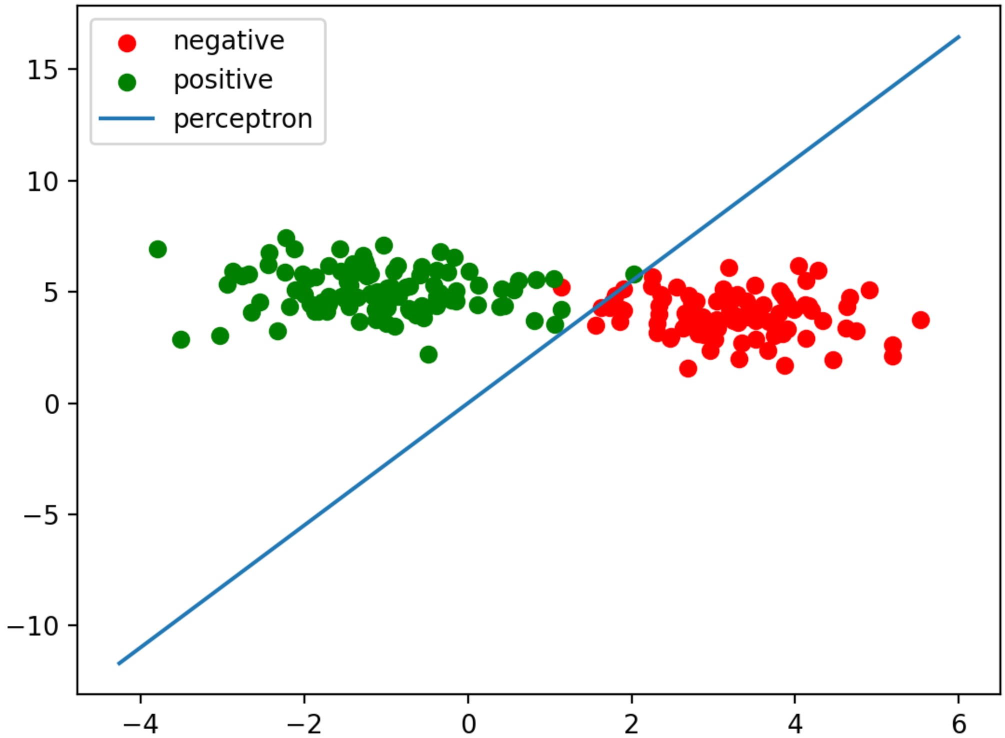



## perceptron learning algorithm on non-separable data

In [46]:

```
w = pt_train(X2, y2)
nextplot()
plot2(X2, y2)
plot2db(w, label="perceptron")
```

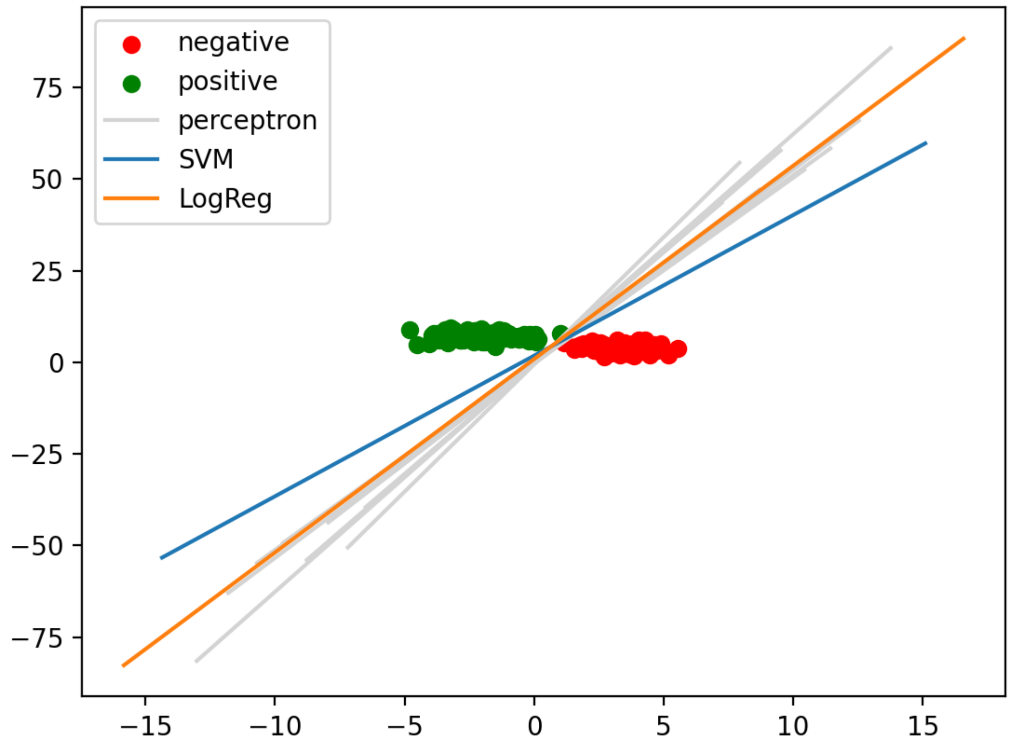
```
weight updates:  398 / tested examples:  20000 / correctly classified:
19602 / incorrectly classified:  398
```




### perceptron learning algorithm on separable data compared with linear SVM and logistic regression

In [49]:

```
nextplot()
plot2dbs(X1, y1, n=10, maxepochs=1000, pocket=False)
```



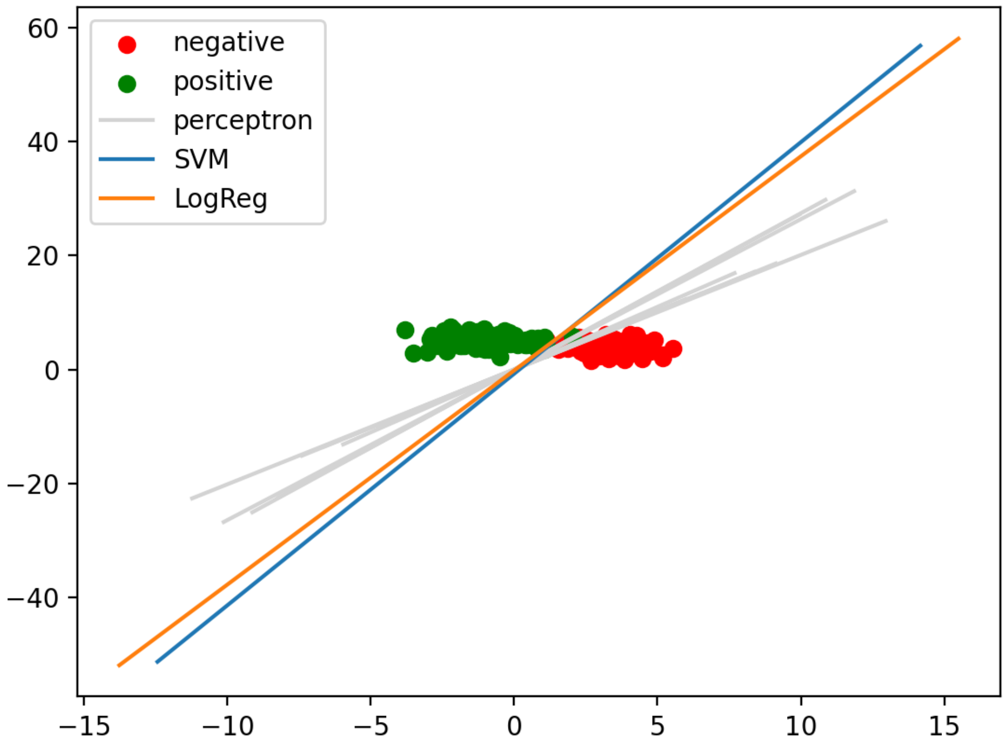

```
stopped training after 4 epochs
weight updates:     5 / tested examples:    800 / correctly classified:
795 / incorrectly classified:     5
stopped training after 3 epochs
weight updates:     3 / tested examples:    600 / correctly classified:
597 / incorrectly classified:     3
stopped training after 5 epochs
weight updates:     8 / tested examples:   1000 / correctly classified:
992 / incorrectly classified:     8
stopped training after 3 epochs
weight updates:     3 / tested examples:    600 / correctly classified:
597 / incorrectly classified:     3
stopped training after 3 epochs
weight updates:     3 / tested examples:    600 / correctly classified:
597 / incorrectly classified:     3
stopped training after 5 epochs
```

```
weight updates:    8 / tested examples:   1000 / correctly classified:
992 / incorrectly classified:    8
stopped training after 6 epochs
weight updates:   10 / tested examples:   1200 / correctly classified:
1190 / incorrectly classified:   10
stopped training after 6 epochs
weight updates:   10 / tested examples:   1200 / correctly classified:
1190 / incorrectly classified:   10
stopped training after 3 epochs
weight updates:    3 / tested examples:    600 / correctly classified:
597 / incorrectly classified:    3
stopped training after 8 epochs
weight updates:   15 / tested examples:   1600 / correctly classified:
1585 / incorrectly classified:   15

Misclassification rates (train)
Perceptron (best result): 0
Linear SVM (C=1)        : 0
Logistic regression     : 0
```

## perceptron learning algorithm on non-separable data and a comparison with linear SVM and logistic regression

In [50]:

```
nextplot()
plot2dbs(X2, y2, n=10, maxepochs=1000, pocket=False)
```

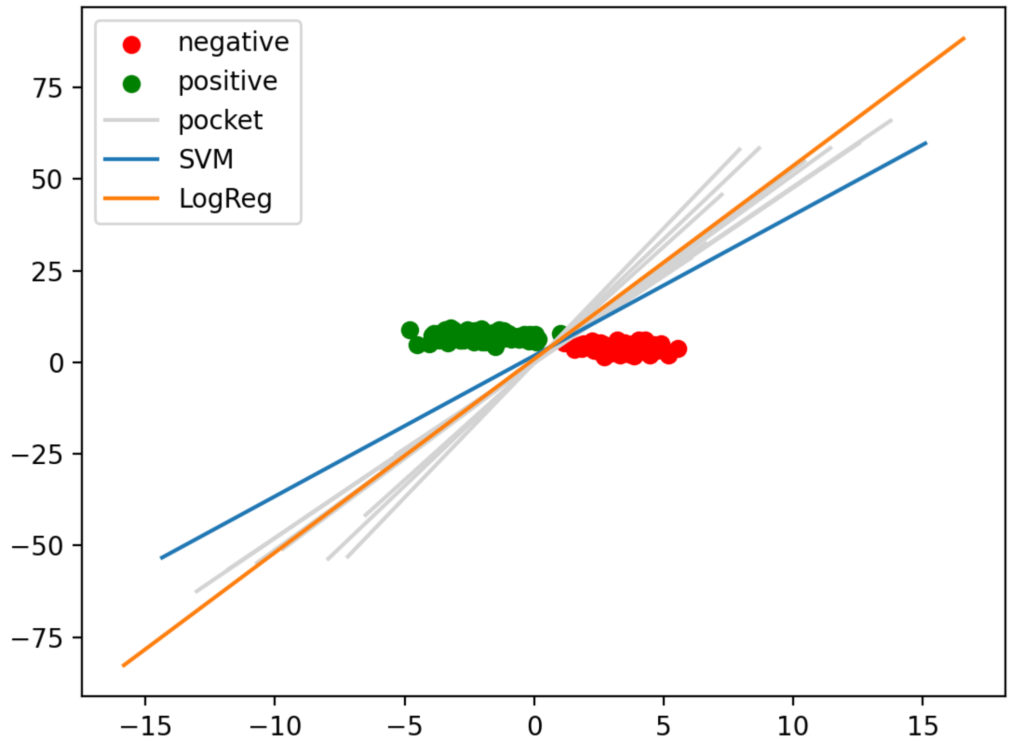



```
weight updates: 4127 / tested examples: 200000 / correctly classified:
195873 / incorrectly classified: 4127
weight updates: 4117 / tested examples: 200000 / correctly classified:
195883 / incorrectly classified: 4117
weight updates: 4128 / tested examples: 200000 / correctly classified:
195872 / incorrectly classified: 4128
weight updates: 4124 / tested examples: 200000 / correctly classified:
195876 / incorrectly classified: 4124
weight updates: 4122 / tested examples: 200000 / correctly classified:
195878 / incorrectly classified: 4122
weight updates: 4122 / tested examples: 200000 / correctly classified:
195878 / incorrectly classified: 4122
weight updates: 4130 / tested examples: 200000 / correctly classified:
195870 / incorrectly classified: 4130
weight updates: 4131 / tested examples: 200000 / correctly classified:
195869 / incorrectly classified: 4131
weight updates: 4132 / tested examples: 200000 / correctly classified:
195868 / incorrectly classified: 4132
weight updates: 4127 / tested examples: 200000 / correctly classified:
195873 / incorrectly classified: 4127

Misclassification rates (train)
Perceptron (best result): 1
Linear SVM (C=1)        : 3
Logistic regression     : 3
```

## pocket algorithm on separable data and a comparison with linear SVM and logistic regression

In [51]:

```
nextplot()
plot2dbs(X1, y1, n=10, maxepochs=1000, pocket=True)
```



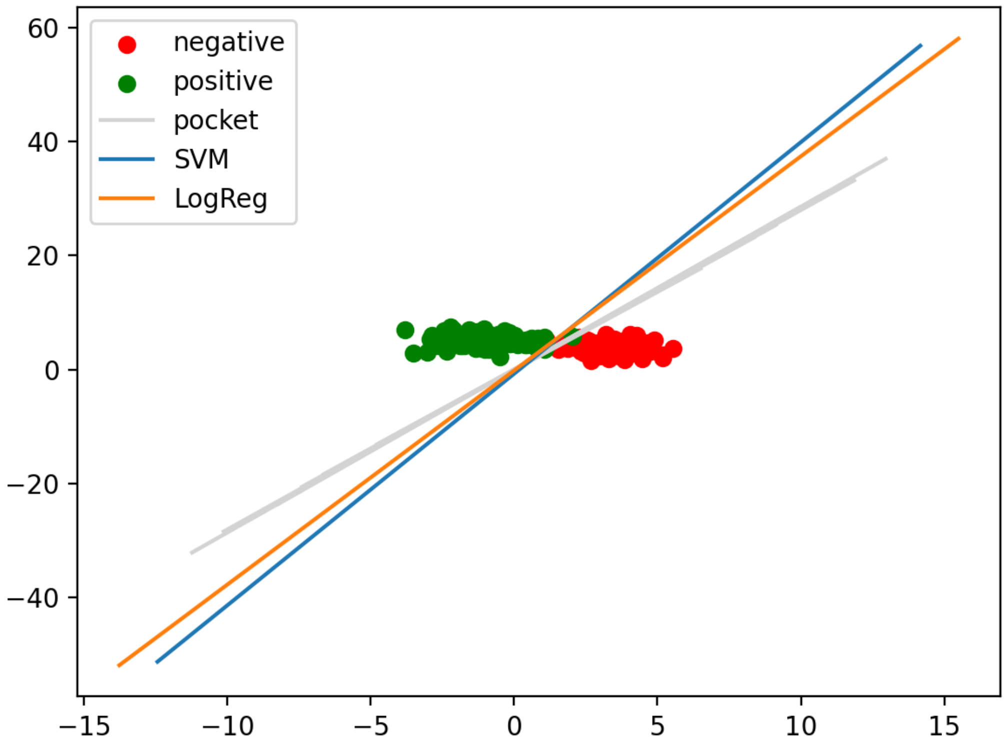

```
weight updates:   20 / tested examples: 200000 / correctly classified:
199980 / incorrectly classified:   20
the best weight vector for the pocket algorithm classified 197949 samp
les correctly in a row
weight updates:    5 / tested examples: 200000 / correctly classified:
199995 / incorrectly classified:    5
the best weight vector for the pocket algorithm classified 199936 samp
les correctly in a row
weight updates:    5 / tested examples: 200000 / correctly classified:
199995 / incorrectly classified:    5
the best weight vector for the pocket algorithm classified 199898 samp
les correctly in a row
weight updates:   24 / tested examples: 200000 / correctly classified:
199976 / incorrectly classified:   24
the best weight vector for the pocket algorithm classified 198814 samp
les correctly in a row
weight updates:   24 / tested examples: 200000 / correctly classified:
199976 / incorrectly classified:   24
the best weight vector for the pocket algorithm classified 197159 samp
les correctly in a row
weight updates:    5 / tested examples: 200000 / correctly classified:
199995 / incorrectly classified:    5
the best weight vector for the pocket algorithm classified 199986 samp
les correctly in a row
weight updates:    9 / tested examples: 200000 / correctly classified:
199991 / incorrectly classified:    9
the best weight vector for the pocket algorithm classified 199176 samp
les correctly in a row
weight updates:   18 / tested examples: 200000 / correctly classified:
```

```
199982 / incorrectly classified:   18
the best weight vector for the pocket algorithm classified 198033 samp
les correctly in a row
weight updates:   27 / tested examples: 200000 / correctly classified:
199973 / incorrectly classified:   27
the best weight vector for the pocket algorithm classified 196524 samp
les correctly in a row
weight updates:   14 / tested examples: 200000 / correctly classified:
199986 / incorrectly classified:   14
the best weight vector for the pocket algorithm classified 199239 samp
les correctly in a row

Misclassification rates (train)
Perceptron (best result): 0
Linear SVM (C=1)        : 0
Logistic regression     : 0
```

### pocket algorithm on non-separable data and a comparison with linear SVM and logistic regression

In [53]:

```
nextplot()
plot2dbs(X2, y2, n=10, maxepochs=1000, pocket=True)
```




```
weight updates: 3235 / tested examples: 200000 / correctly classified:
196765 / incorrectly classified: 3235
the best weight vector for the pocket algorithm classified 1376 sample
s correctly in a row
weight updates: 3154 / tested examples: 200000 / correctly classified:
196846 / incorrectly classified: 3154
the best weight vector for the pocket algorithm classified 1493 sample
s correctly in a row
weight updates: 3232 / tested examples: 200000 / correctly classified:
196768 / incorrectly classified: 3232
the best weight vector for the pocket algorithm classified 1119 sample
s correctly in a row
weight updates: 3170 / tested examples: 200000 / correctly classified:
196830 / incorrectly classified: 3170
the best weight vector for the pocket algorithm classified 1451 sample
s correctly in a row
weight updates: 3177 / tested examples: 200000 / correctly classified:
196823 / incorrectly classified: 3177
the best weight vector for the pocket algorithm classified 1093 sample
s correctly in a row
weight updates: 3129 / tested examples: 200000 / correctly classified:
196871 / incorrectly classified: 3129
the best weight vector for the pocket algorithm classified 1532 sample
s correctly in a row
weight updates: 3172 / tested examples: 200000 / correctly classified:
196828 / incorrectly classified: 3172
the best weight vector for the pocket algorithm classified 920 samples
correctly in a row
weight updates: 2996 / tested examples: 200000 / correctly classified:
```

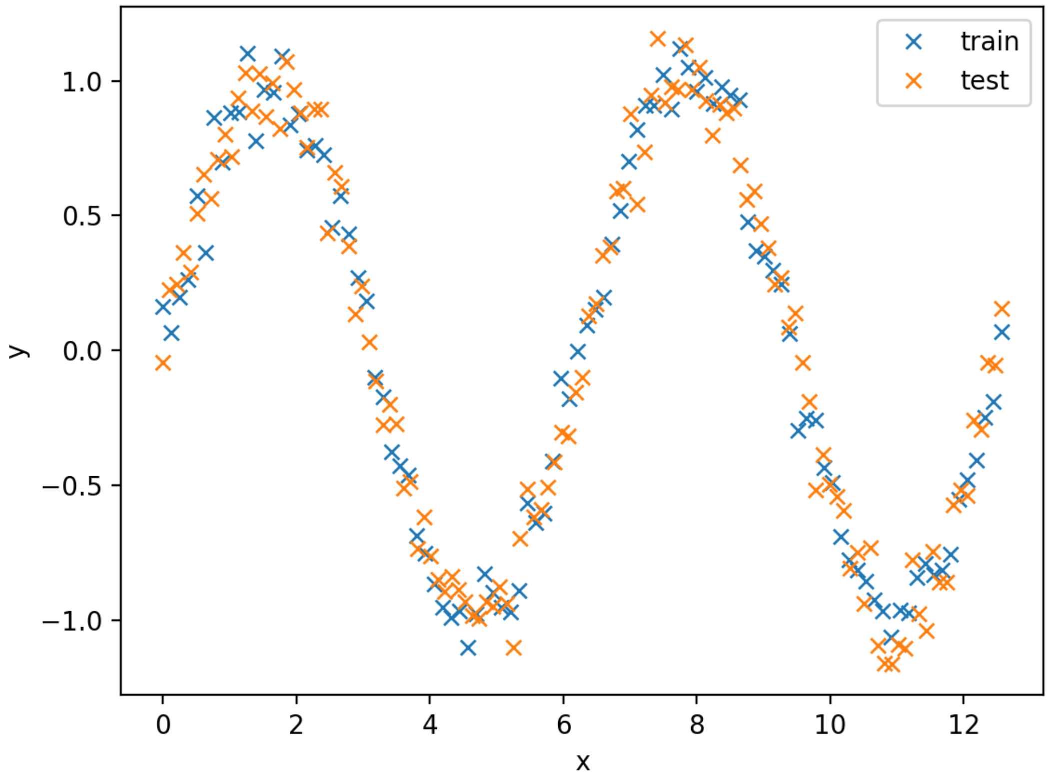
```
197004 / incorrectly classified: 2996
the best weight vector for the pocket algorithm classified 1221 sample
s correctly in a row
weight updates: 3229 / tested examples: 200000 / correctly classified:
196771 / incorrectly classified: 3229
the best weight vector for the pocket algorithm classified 1466 sample
s correctly in a row
weight updates: 3188 / tested examples: 200000 / correctly classified:
196812 / incorrectly classified: 3188
the best weight vector for the pocket algorithm classified 1505 sample
s correctly in a row

Misclassification rates (train)
Perceptron (best result): 1
Linear SVM (C=1)        : 3
Logistic regression     : 3
```

# 2 Multi-Layer Feed-Forward Neural Networks

## 2a Conjecture how an FNN fit will look like

In [5]:

```
# here is the one-dimensional dataset that we will use
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plt.legend()
```




Out[5]:

```
<matplotlib.legend.Legend at 0x7f7d2b3748e0>
```

## 2b Train with 2 hidden units

In [37]:

```
# Training code. You do not need to modify this code.
train_bfgs = lambda model, **kwargs: train_scipy(X3, y3, model, **kwargs)


def train3(
    hidden_sizes, nreps=10, transfer=lambda: nn.Sigmoid(), train=train_bfgs, **kwarg
):
    """Train an FNN.

    hidden_sizes is a (possibly empty) list containing the sizes of the hidden layer
    nreps refers to the number of repetitions.

    """
    best_model = None
    best_cost = math.inf
    for rep in range(nreps):
        model = fnn_model([1] + hidden_sizes + [1], transfer)
        print(f"Repetition {rep: 2d}: ", end="")
        model = train(model, **kwargs)
        mse = F.mse_loss(y3, model(X3)).item()
        if mse < best_cost:
            best_model = model
            best_cost = mse
        print(f"best_cost={best_cost:.3f}")

    return best_model
```

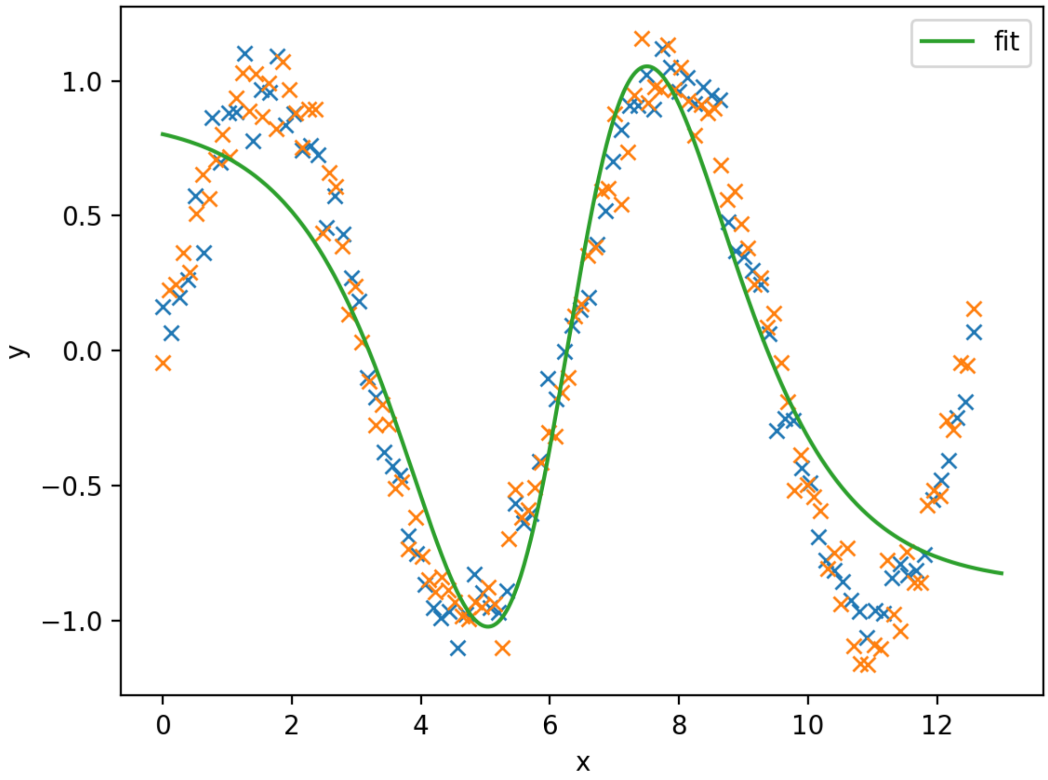
In [39]:

```
# Let's fit the model with one hidden layer consisting of 2 units.
model = train3([2], nreps=1)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079572
         Iterations: 390
         Function evaluations: 610
         Gradient evaluations: 596
best_cost=0.080
Training error: 0.07957000285387039
Test error    : 0.0867156982421875
```

In [40]:

```
# plot the data and the fit
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
# torch.linspace(0, 13, 500).unsqueeze(1): creates a tensor of dim 1 with 500
# elements that are equally spaced from 0 to 13 (i.e. unsqueeze(1) is responsible)
# for the 1d part
```



In [9]:

```
# The weight matrices and bias vectors can be read out as follows. If you want,
# use these parameters to compute the output of the network (on X3) directly and
# compare to model(X3).
for par, value in model.state_dict().items():
    print(f"{par:<15}= {value}")
```

```
linear1.weight = tensor([[-5.0166],
        [ 6.0141]])
linear1.bias   = tensor([13.9789, -3.4538])
output.weight  = tensor([[1.1157, 0.8249]])
output.bias    = tensor([-1.0277])
```

In [10]:

```
# now repeat this multiple times
# YOUR CODE HERE
model = train3([2], nreps=3)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 364
         Function evaluations: 502
         Gradient evaluations: 490
best_cost=0.080
Repetition  1: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 380
         Function evaluations: 575
         Gradient evaluations: 554
best_cost=0.080
Repetition  2: Optimization terminated successfully.
         Current function value: 0.301865
         Iterations: 359
         Function evaluations: 410
         Gradient evaluations: 410
best_cost=0.080
Training error: 0.07957139611244202
Test error    : 0.08671265095472336
```

In [11]:

```
# From now on, always train multiple times (nreps=10 by default) and
# report best model.
model = train3([2], nreps=10)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 382
         Function evaluations: 596
         Gradient evaluations: 587
best_cost=0.080
Repetition  1: Optimization terminated successfully.
         Current function value: 0.277769
         Iterations: 79
         Function evaluations: 83
         Gradient evaluations: 83
best_cost=0.080
Repetition  2: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079572
         Iterations: 416
         Function evaluations: 577
         Gradient evaluations: 566
best_cost=0.080
Repetition  3: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 428
         Function evaluations: 691
         Gradient evaluations: 673
best_cost=0.080
Repetition  4: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 362
         Function evaluations: 490
         Gradient evaluations: 479
best_cost=0.080
Repetition  5: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.286909
         Iterations: 326
         Function evaluations: 444
         Gradient evaluations: 435
best_cost=0.080
Repetition  6: Optimization terminated successfully.
         Current function value: 0.357250
         Iterations: 62
         Function evaluations: 70
         Gradient evaluations: 70
best_cost=0.080
Repetition  7: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 366
         Function evaluations: 583
         Gradient evaluations: 571
```

```
best_cost=0.080
Repetition  8: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 394
         Function evaluations: 605
         Gradient evaluations: 593
best_cost=0.080
Repetition  9: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 391
         Function evaluations: 535
         Gradient evaluations: 525
best_cost=0.080
Training error: 0.07956835627555847
Test error    : 0.08670976012945175
```

# 2c Width

In [12]:

```
# Experiment with different hidden layer sizes. To avoid recomputing
# models, you may want to save your models using torch.save(model, filename) and
# load them again using torch.load(filename).
# model1 = train3([1], nreps=10)
# model2 = train3([2], nreps=10)
# model3 = train3([3], nreps=10)
# model10 = train3([10], nreps=10)
# model50 = train3([50], nreps=10)
# model100 = train3([100], nreps=10)
```

### saving models

In [ ]:

```
torch.save(model1, "models/model1.txt")
torch.save(model2, "models/model2.txt")
torch.save(model3, "models/model3.txt")
torch.save(model10, "models/model10.txt")
torch.save(model50, "models/model50.txt")
torch.save(model100, "models/model100.txt")
```

### loading models

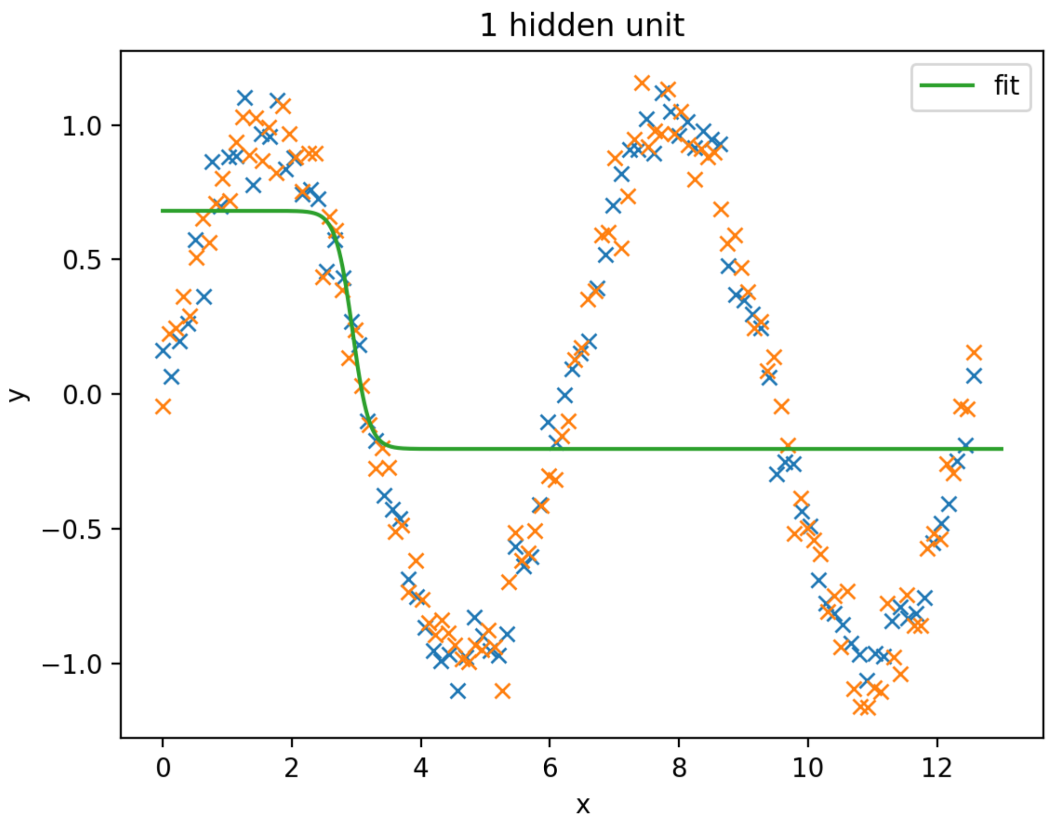
In [14]:

```
model1 = torch.load("models/model1.txt")
model2 = torch.load("models/model2.txt")
model3 = torch.load("models/model3.txt")
model10 = torch.load("models/model10.txt")
model50 = torch.load("models/model50.txt")
model100 = torch.load("models/model100.txt")
```

## 1 hidden unit

In [15]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model1)
plt.title("1 hidden unit")
print("Training error:", F.mse_loss(y3, model1(X3)).item())
print("Test error    :", F.mse_loss(y3test, model1(X3test)).item())
```

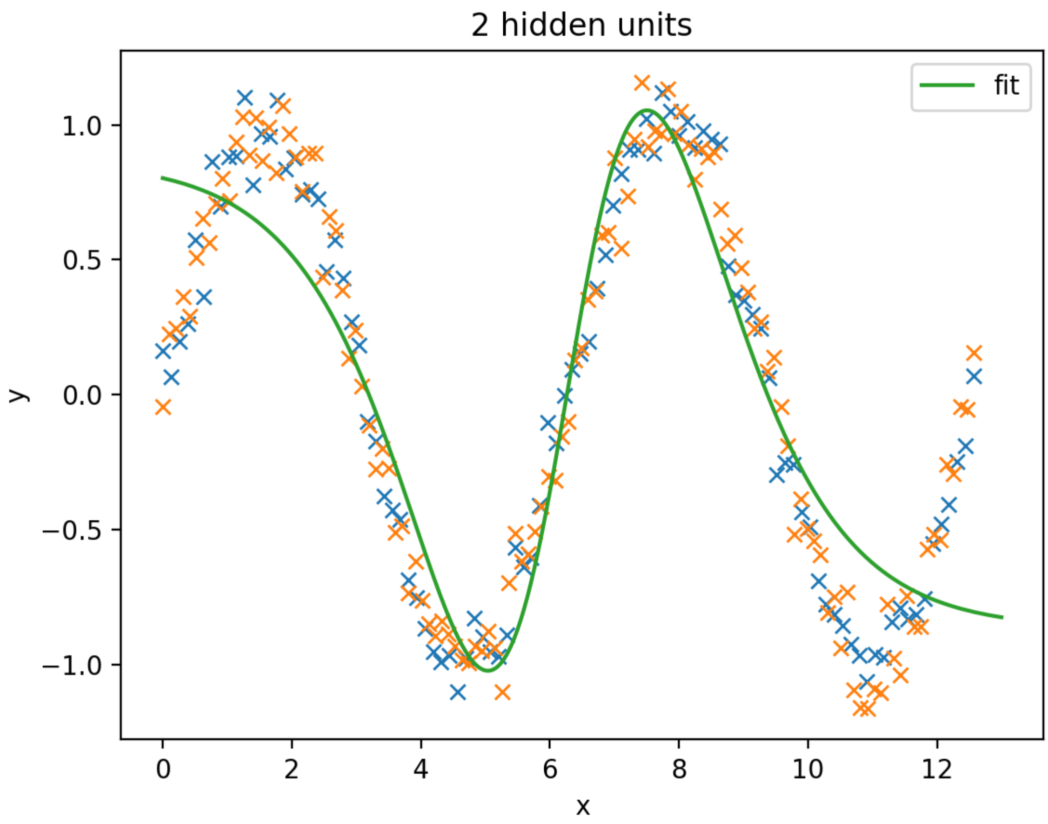



```
Training error: 0.37291884422302246
Test error    : 0.37431666254997253
```

## 2 hidden units

In [16]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model2)
plt.title("2 hidden units")
print("Training error:", F.mse_loss(y3, model2(X3)).item())
print("Test error    :", F.mse_loss(y3test, model2(X3test)).item())
```

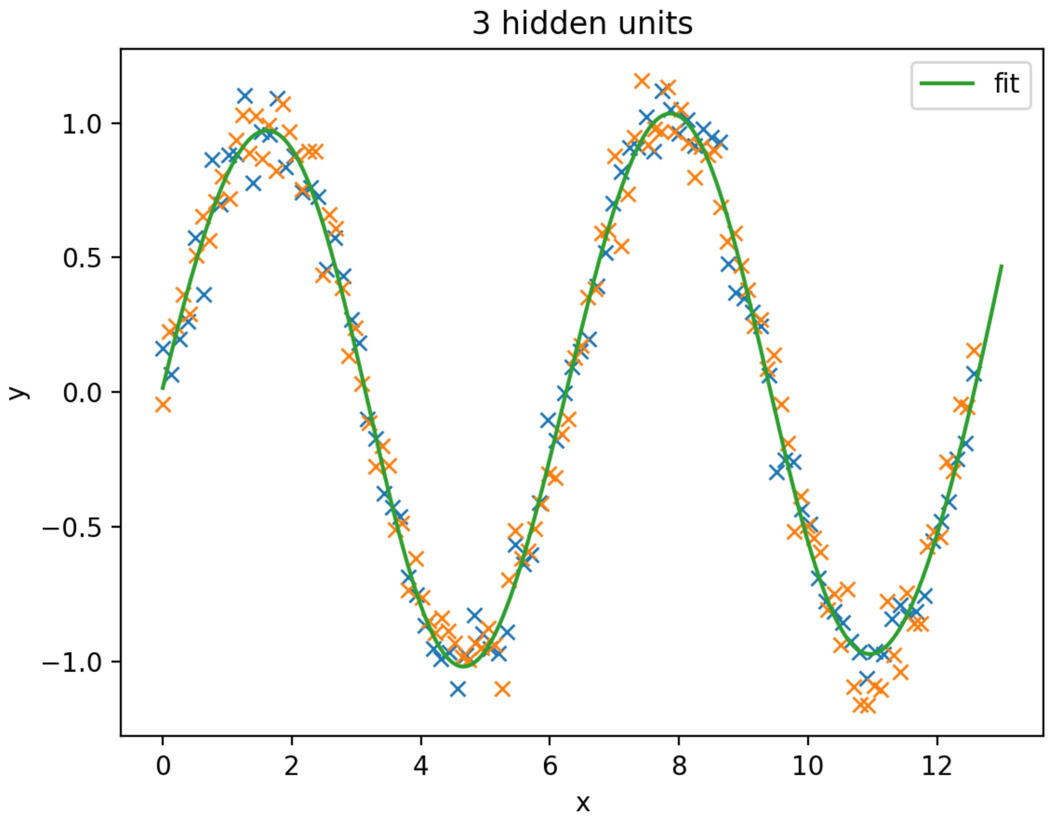



```
Training error: 0.07957205921411514
Test error    : 0.08671297132968903
```

## 3 hidden units

In [17]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model3)
plt.title("3 hidden units")
print("Training error:", F.mse_loss(y3, model3(X3)).item())
print("Test error    :", F.mse_loss(y3test, model3(X3test)).item())
```

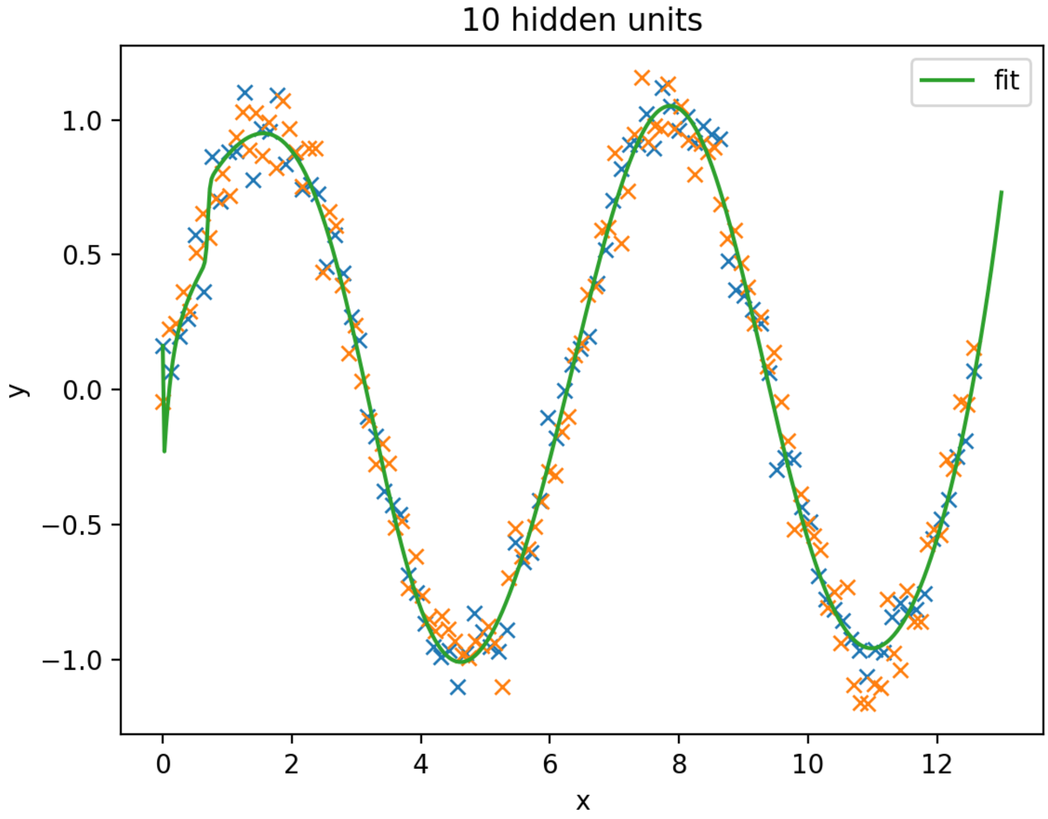



```
Training error: 0.007324173580855131
Test error    : 0.010335267521440983
```

## 10 hidden units

In [25]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model10)
plt.title("10 hidden units")
print("Training error:", F.mse_loss(y3, model10(X3)).item())
print("Test error    :", F.mse_loss(y3test, model10(X3test)).item())
```

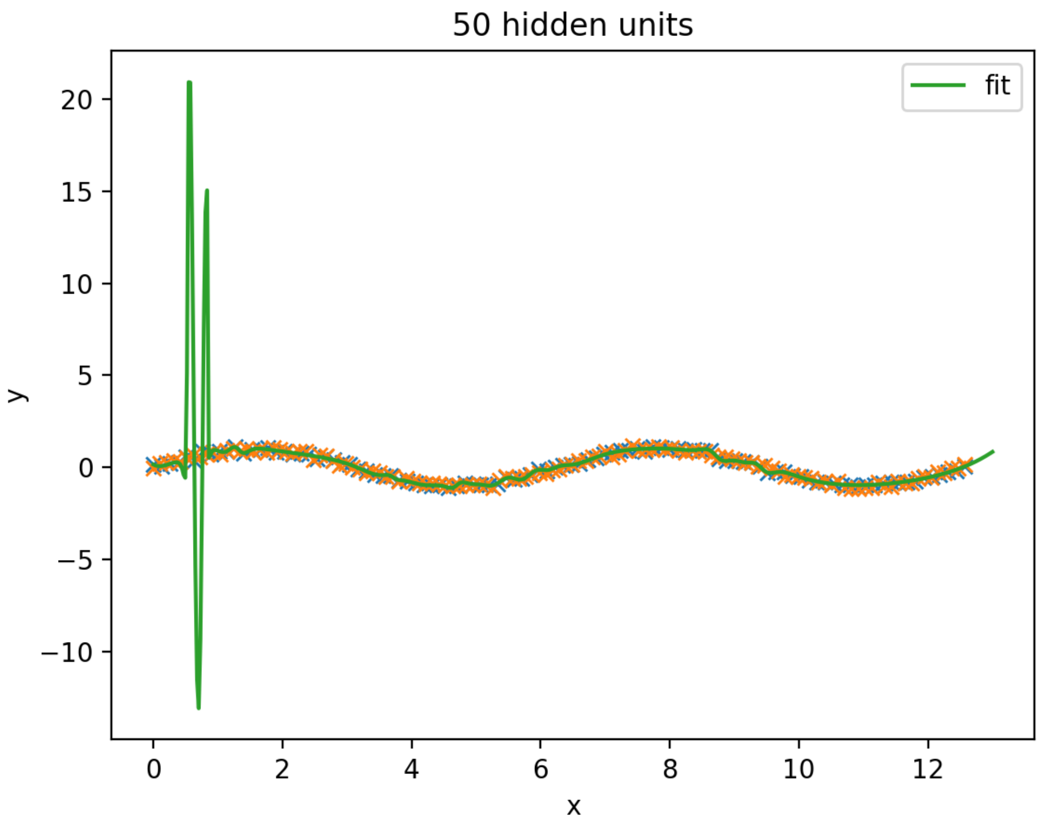



```
Training error: 0.006357043981552124
Test error    : 0.011850706301629543
```

## 50 hidden units

In [19]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model50)
plt.title("50 hidden units")
print("Training error:", F.mse_loss(y3, model50(X3)).item())
print("Test error    :", F.mse_loss(y3test, model50(X3test)).item())
```

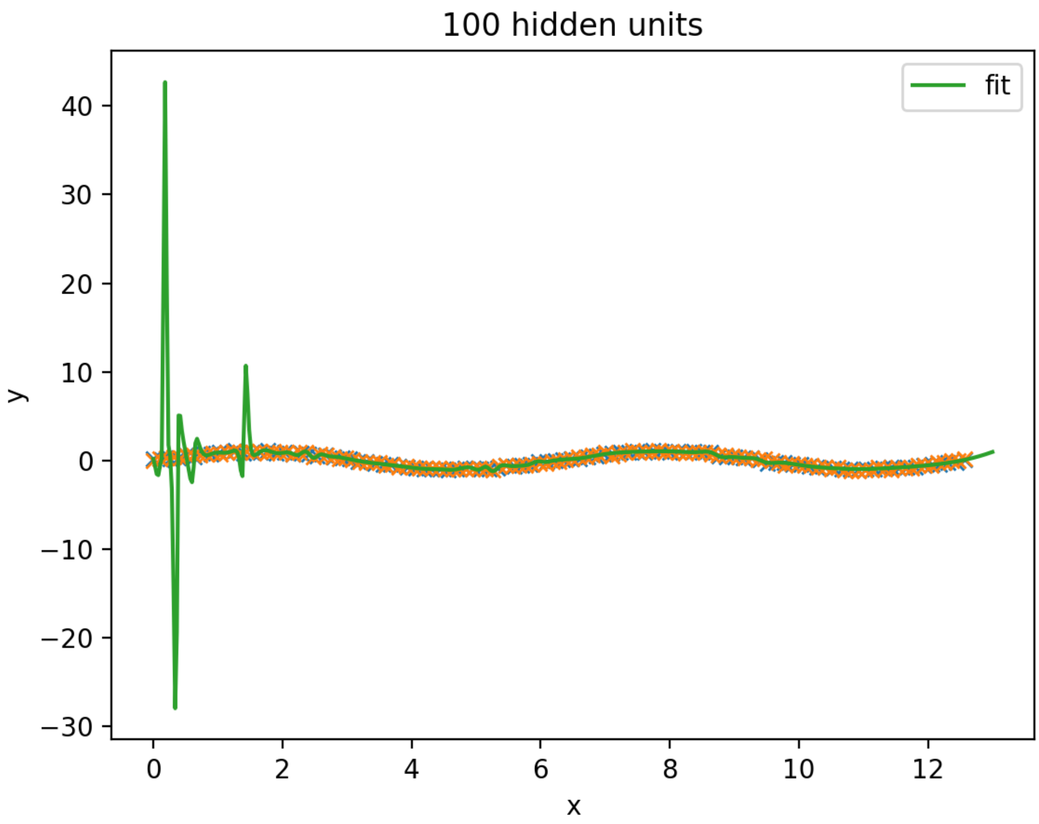



```
Training error: 0.0016042940551415086
Test error    : 3.343168020248413
```

## 100 hidden units

In [26]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model100)
plt.title("100 hidden units")
print("Training error:", F.mse_loss(y3, model100(X3)).item())
print("Test error    :", F.mse_loss(y3test, model100(X3test)).item())
```

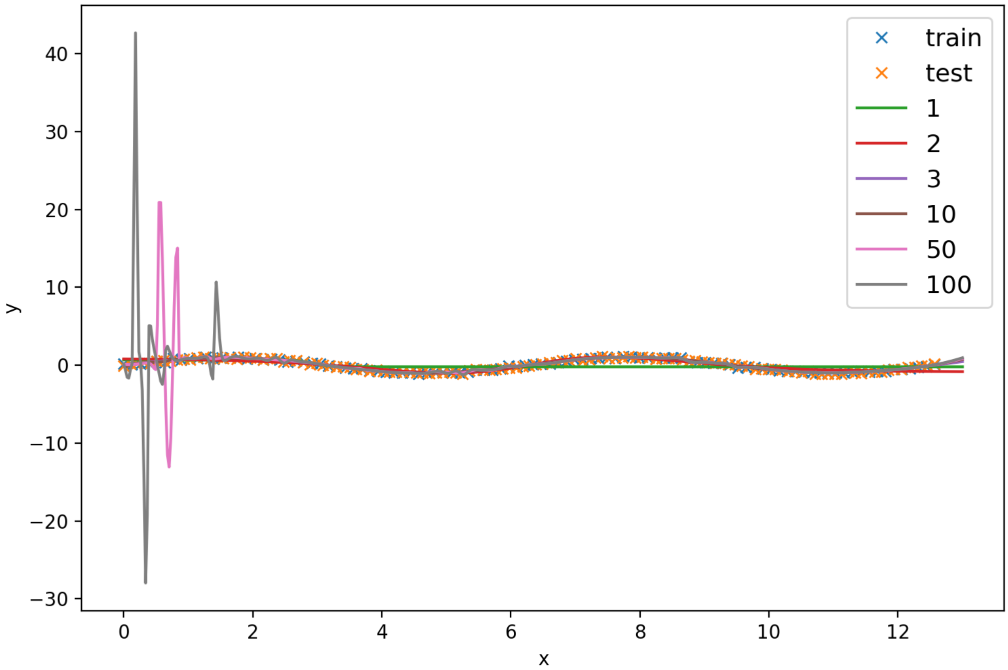



```
Training error: 0.0010693169897422194
Test error    : 6.358546733856201
```

### Then plot the dataset as well as the predictions of each FNN on the test set into a single plot.

In [30]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model1, label="1")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model2, label="2")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model3, label="3")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model10, label="10")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model50, label="50")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model100, label="100")
#plt.title("100 hidden units")

plt.legend(prop={'size': 13})
plt.show()
```



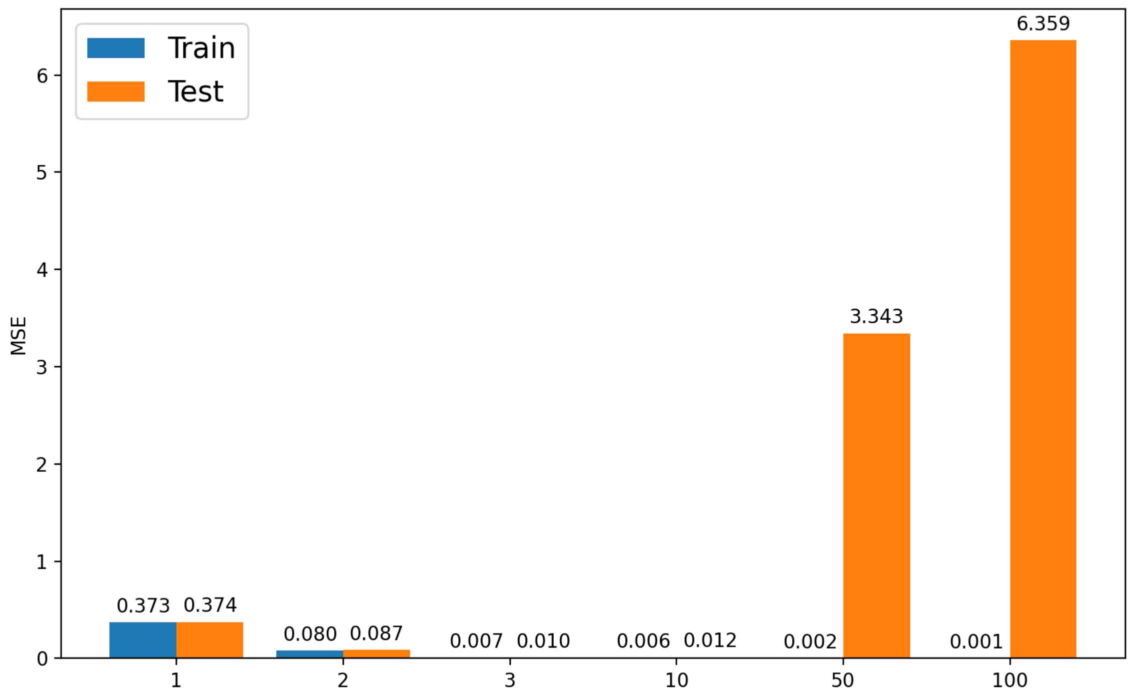

## Training and test error for 1 to 100 hidden neurons

In [22]:

```
train1 = F.mse_loss(y3, model1(X3)).item()
test1 = F.mse_loss(y3test, model1(X3test)).item()

train2 = F.mse_loss(y3, model2(X3)).item()
test2 = F.mse_loss(y3test, model2(X3test)).item()

train3 = F.mse_loss(y3, model3(X3)).item()
test3 = F.mse_loss(y3test, model3(X3test)).item()

train10 = F.mse_loss(y3, model10(X3)).item()
test10 = F.mse_loss(y3test, model10(X3test)).item()

train50 = F.mse_loss(y3, model50(X3)).item()
test50 = F.mse_loss(y3test, model50(X3test)).item()

train100 = F.mse_loss(y3, model100(X3)).item()
test100 = F.mse_loss(y3test, model100(X3test)).item()
```

In [32]:

```
labels = ['1', '2', '3', '10', '50', '100']
train = [train1, train2, train3, train10, train50, train100]
test = [test1, test2, test3, test10, test50, test100]

x = np.arange(len(labels))  # the label locations
width = 0.4  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, train, width, label='Train')
rects2 = ax.bar(x + width/2, test, width, label='Test')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('MSE')
ax.set_xticks(x, labels)
ax.legend(prop={'size': 15})


ax.bar_label(rects1, padding=3, fmt="%1.3f")
ax.bar_label(rects2, padding=3, fmt="%1.3f")

fig.tight_layout()

plt.show()
```




# 2d Distributed representations

## 2 hidden unit

In [63]:

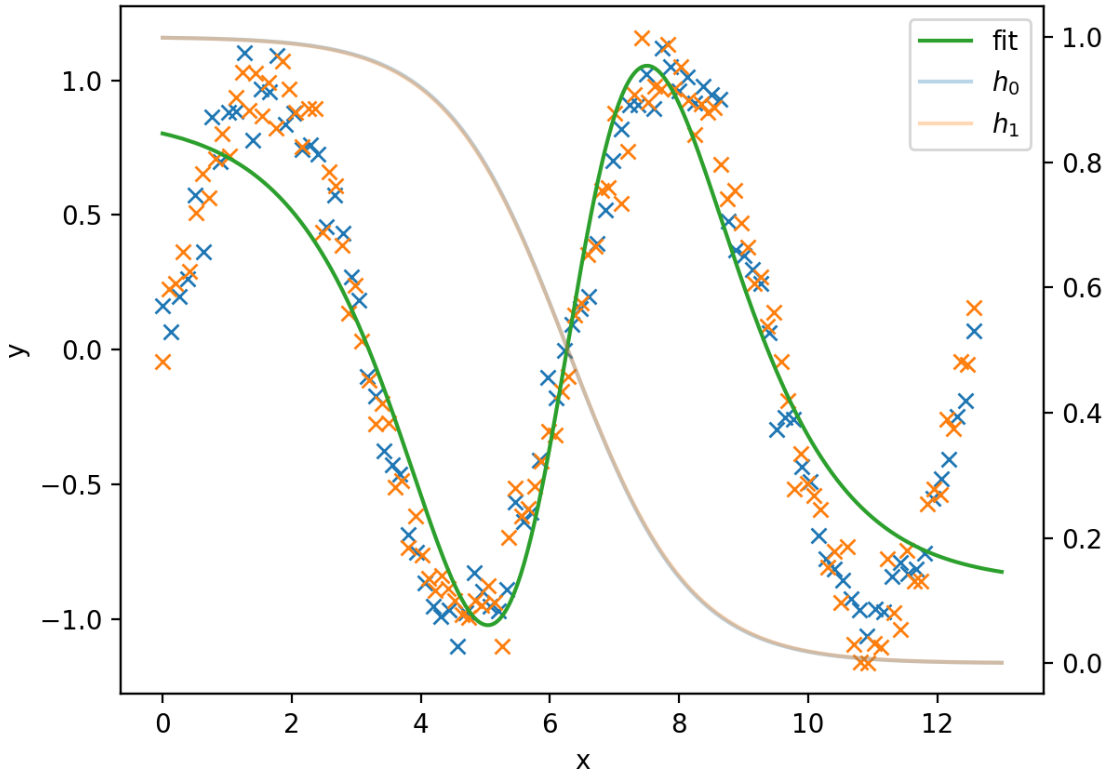
```
# train a model to analyze
model_dist_2 = train3([2])
torch.save(model_dist_2, "models/model_dist_2.txt")
model_dist_2 = torch.load("models/model_dist_2.txt")
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.367556
         Iterations: 54
         Function evaluations: 130
         Gradient evaluations: 118
best_cost=0.368
Repetition  1: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.286909
         Iterations: 379
         Function evaluations: 513
         Gradient evaluations: 501
best_cost=0.287
Repetition  2: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079572
         Iterations: 414
         Function evaluations: 589
         Gradient evaluations: 575
best_cost=0.080
Repetition  3: Optimization terminated successfully.
         Current function value: 0.357250
         Iterations: 86
         Function evaluations: 96
         Gradient evaluations: 96
best_cost=0.080
Repetition  4: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 387
         Function evaluations: 553
         Gradient evaluations: 542
best_cost=0.080
Repetition  5: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 369
         Function evaluations: 523
         Gradient evaluations: 513
best_cost=0.080
Repetition  6: Optimization terminated successfully.
         Current function value: 0.302737
         Iterations: 127
         Function evaluations: 151
         Gradient evaluations: 151
best_cost=0.080
Repetition  7: Optimization terminated successfully.
         Current function value: 0.357250
         Iterations: 86
         Function evaluations: 93
         Gradient evaluations: 93
best_cost=0.080
Repetition  8: Optimization terminated successfully.
```

```
         Current function value: 0.357250
         Iterations: 87
         Function evaluations: 99
         Gradient evaluations: 99
best_cost=0.080
Repetition  9: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.079573
         Iterations: 383
         Function evaluations: 596
         Gradient evaluations: 582
best_cost=0.080
```

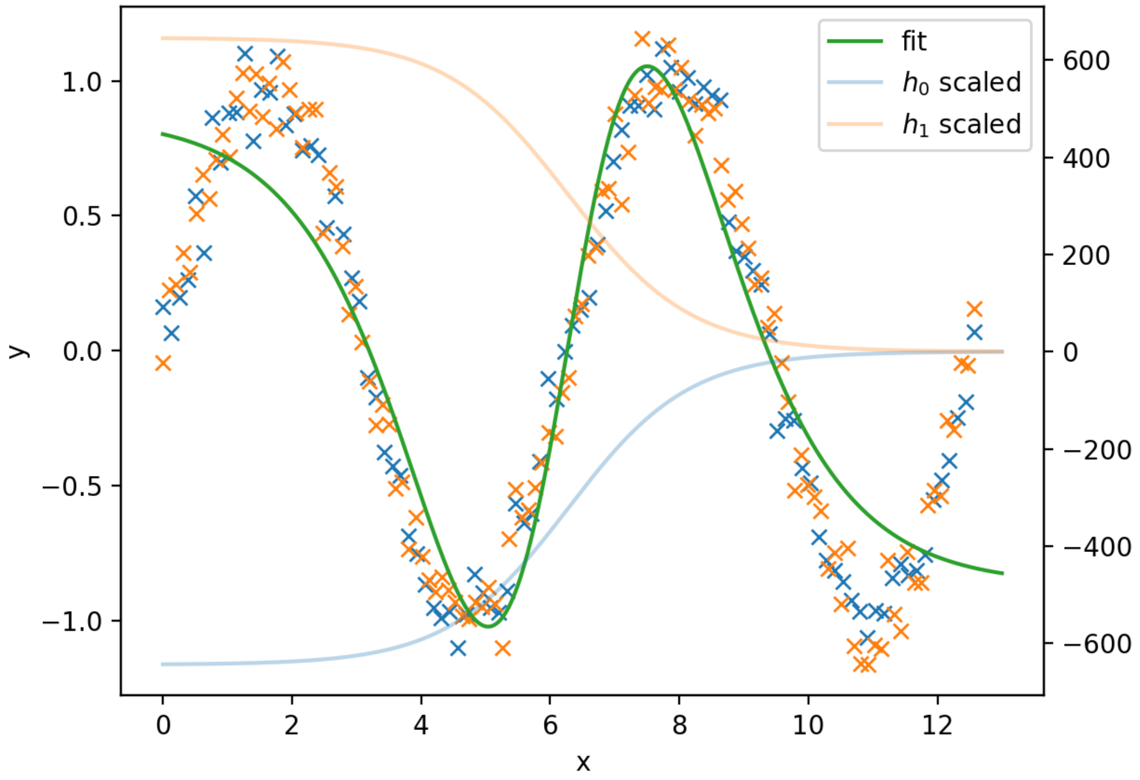
In [87]:

```
# plot the fit as well as the outputs of each neuron in the hidden
# layer (scale for the latter is shown on right y-axis)
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model_dist_2, hidden=True, scale=F
```



In [88]:

```
# plot the fit as well as the outputs of each neuron in the hidden layer, scaled
# by its weight for the output neuron (scale for the latter is shown on right
# y-axis)
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model_dist_2, hidden=True, scale=T
```




**weights of the model with two (2) hidden units**

In [67]:

```
for par, value in model_dist_2.state_dict().items():
    print(f"{par:<15}= {value}")
```

```
linear1.weight = tensor([[-1.0684],
        [-1.0569]])
linear1.bias   = tensor([6.7106, 6.6383])
output.weight  = tensor([[-644.4341,  646.1637]])
output.bias    = tensor([-0.8657])
```

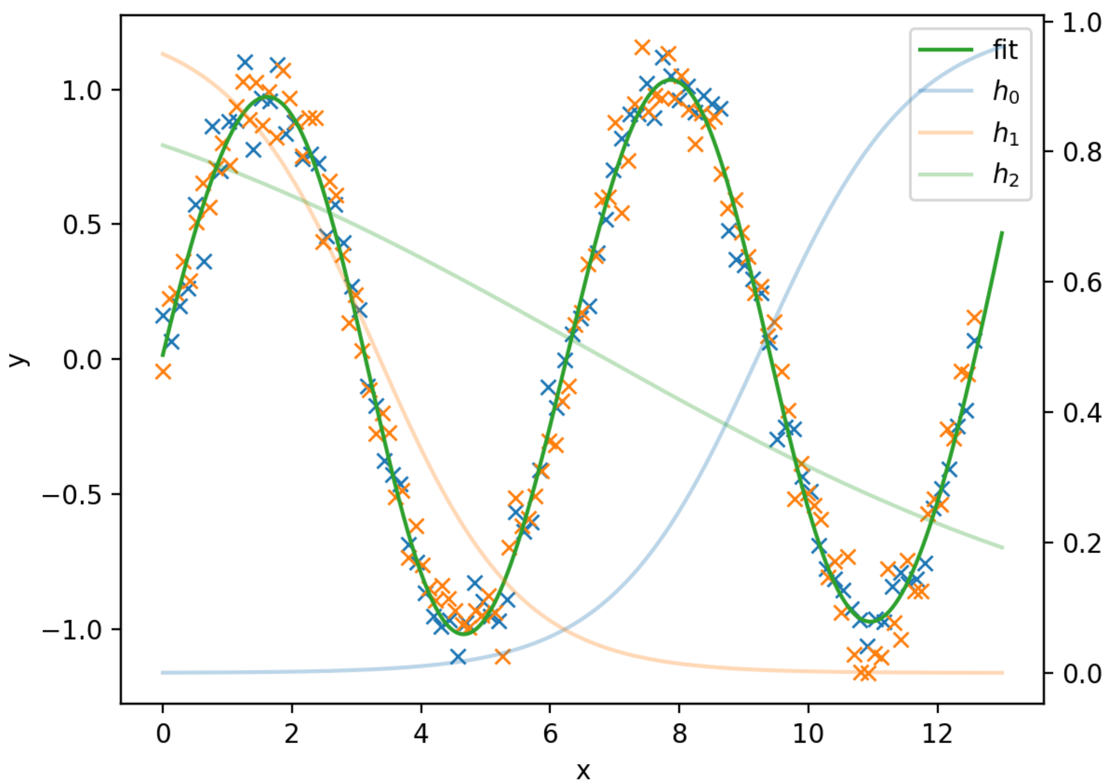
## 3 hidden units

In [57]:

```
# model_dist_3 = train3([3])
# torch.save(model_dist_3, "models/model_dist_3.txt")
model_dist_3 = torch.load("models/model_dist_3.txt")
```

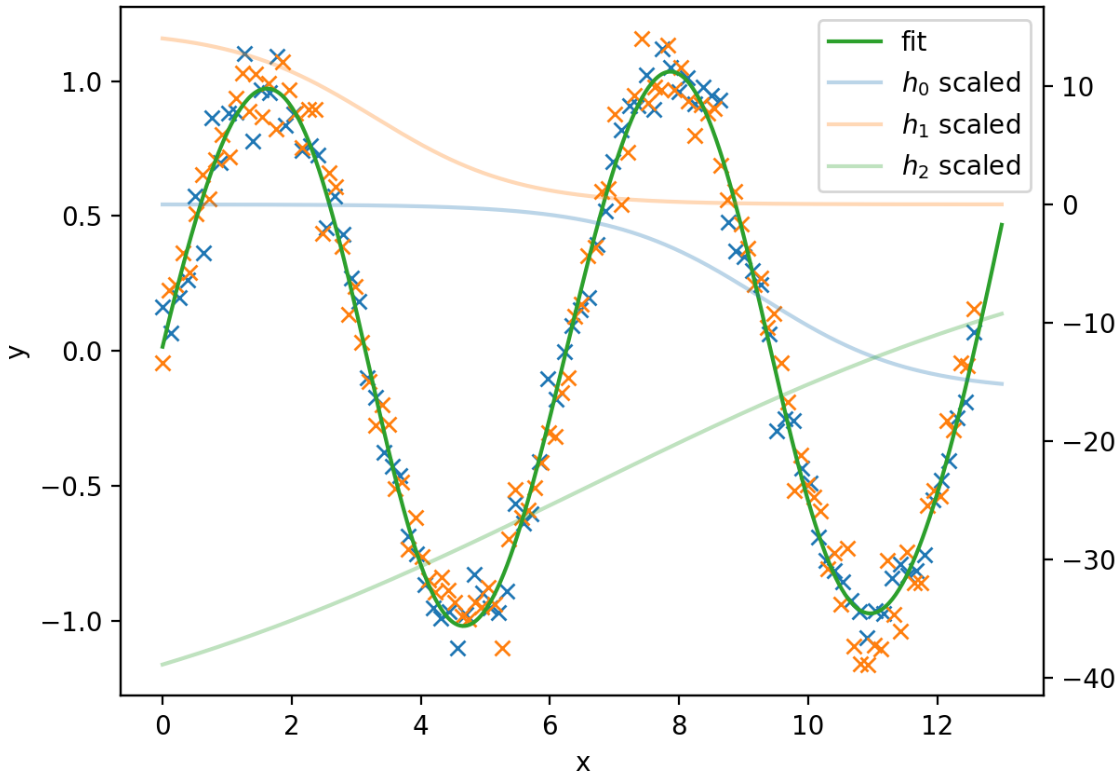
In [89]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model_dist_3, hidden=True, scale=F
```



In [92]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model_dist_3, hidden=True, scale=T
```




**weights of the model with three (3) hidden units**

In [68]:

```
for par, value in model_dist_3.state_dict().items():
    print(f"{par:<15}= {value}")
```

```
linear1.weight = tensor([[ 0.8628],
        [-0.8973],
        [-0.2218]])
linear1.bias   = tensor([-8.0104,  2.9424,  1.4495])
output.weight  = tensor([[-15.7720,  14.7583, -47.9800]])
output.bias    = tensor([24.8621])
```

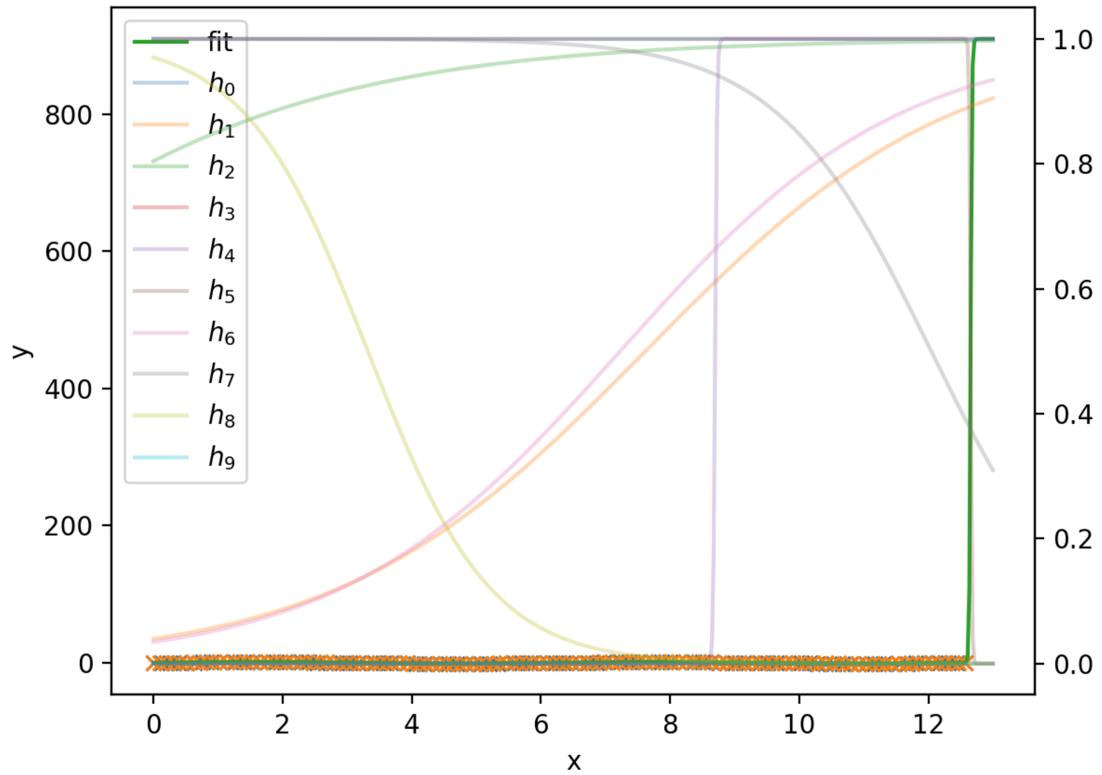
## 10 hidden units

In [60]:

```
# model_dist_10 = train3([10])
# torch.save(model_dist_10, "models/model_dist_10.txt")
model_dist_10 = torch.load("models/model_dist_10.txt")
```

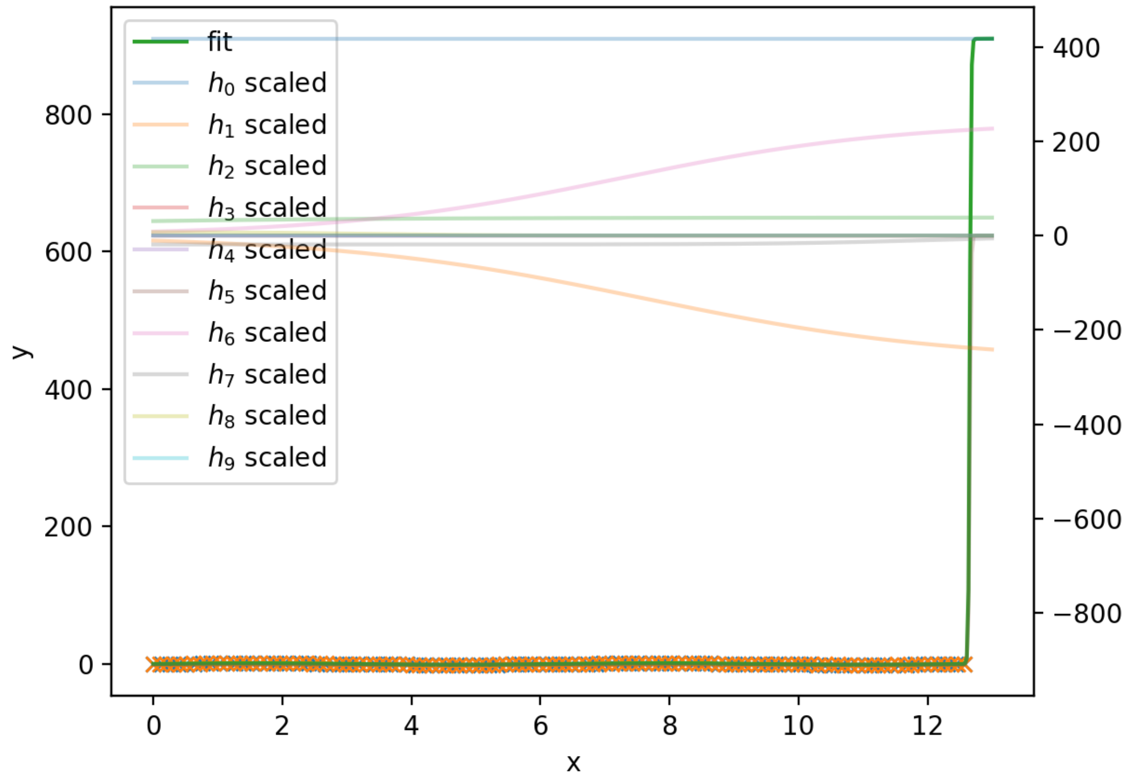
In [94]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model_dist_10, hidden=True, scale=
```



In [95]:

```
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model_dist_10, hidden=True, scale=
```




**weights of the model with ten (10) hidden units**

In [70]:

```
for par, value in model_dist_10.state_dict().items():
    print(f"{par:<15}= {value}")
```

```
linear1.weight = tensor([[ 2.8365e+02],
        [ 4.1877e-01],
        [ 3.3326e-01],
        [-3.7034e+02],
        [ 6.7506e+01],
        [-9.8656e+01],
        [ 4.6061e-01],
        [-8.3900e-01],
        [-1.0473e+00],
        [-7.8167e+01]])
linear1.bias   = tensor([ 365.0792,   -3.1908,    1.4124,   -7.4229, -
587.2411, 1248.5552,
          -3.3290,   10.1059,    3.4771, -219.2267])
output.weight  = tensor([[ 4.1737e+02, -2.6671e+02,  3.8250e+01,  4.37
23e+02, -2.6638e-01,
         -9.0956e+02,  2.4256e+02, -1.8766e+01,  7.4181e+00, -2.7525e+
01]])
output.bias    = tensor([475.0463])
```

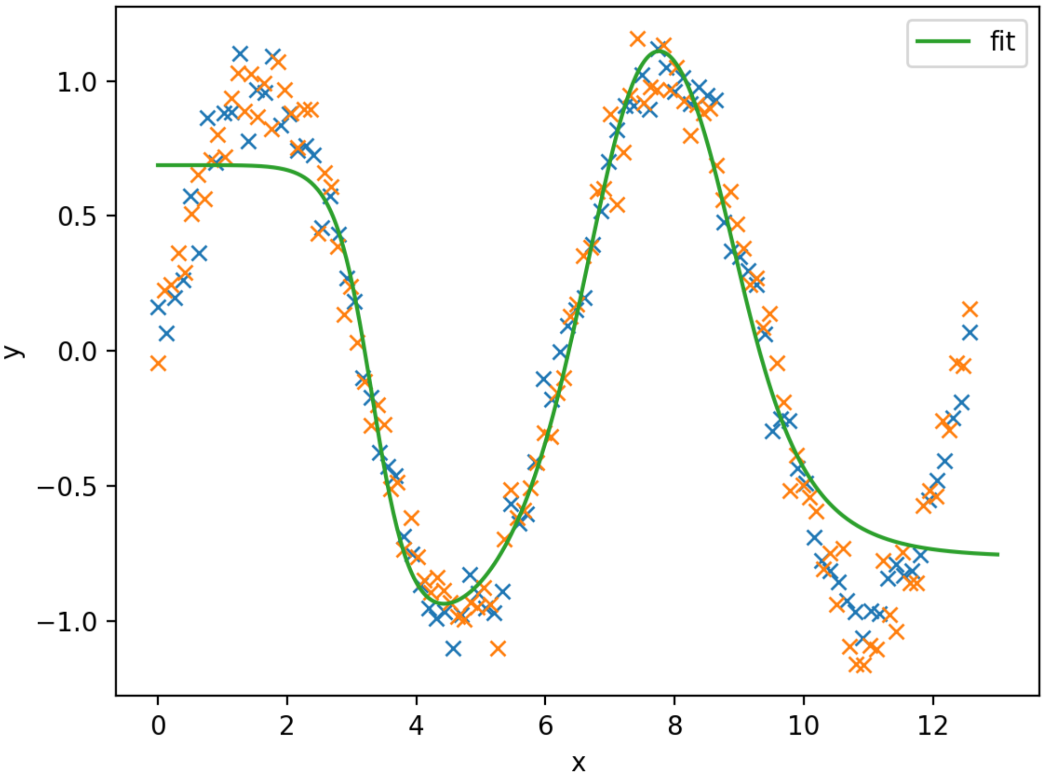
## 2e Experiment with different optimizers (optional)

### Varying layers

In [129]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adam(model.parameters(), lr=0.01), **kwargs
)
model = train3([2, 2], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, verbos
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

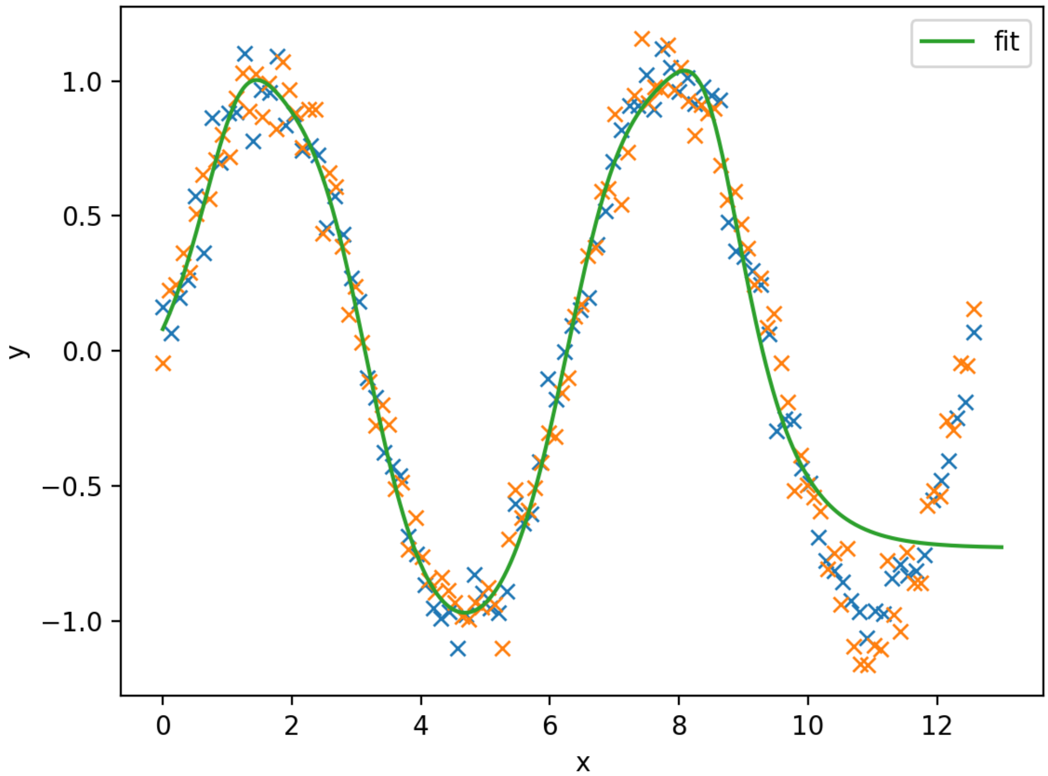
```
Repetition  0: best_cost=0.048
Repetition  1: best_cost=0.048
Repetition  2: best_cost=0.048
Repetition  3: best_cost=0.048
Repetition  4: best_cost=0.048
Repetition  5: best_cost=0.048
Repetition  6: best_cost=0.048
Repetition  7: best_cost=0.048
Repetition  8: best_cost=0.048
Repetition  9: best_cost=0.048
```



In [130]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adam(model.parameters(), lr=0.01), **kwargs
)
model = train3([2, 2, 2], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, ver
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

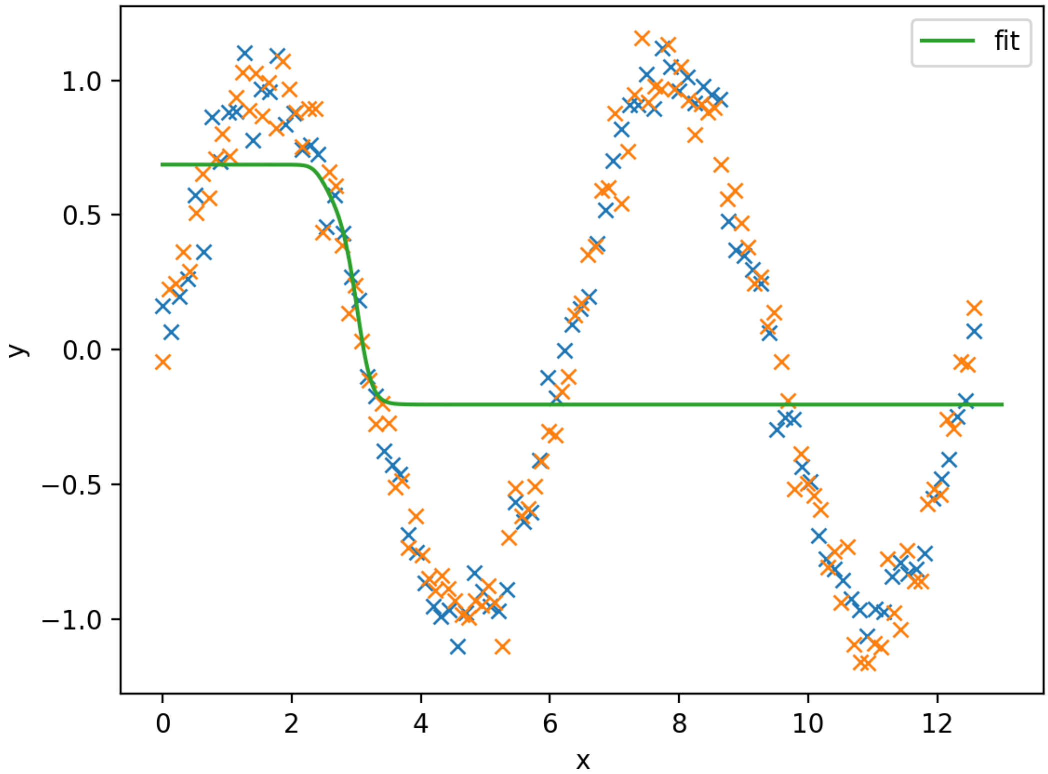
```
Repetition  0: best_cost=0.373
Repetition  1: best_cost=0.277
Repetition  2: best_cost=0.052
Repetition  3: best_cost=0.052
Repetition  4: best_cost=0.052
Repetition  5: best_cost=0.052
Repetition  6: best_cost=0.028
Repetition  7: best_cost=0.028
Repetition  8: best_cost=0.028
Repetition  9: best_cost=0.028
```



In [131]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adam(model.parameters(), lr=0.01), **kwargs
)
model = train3([3, 1, 3], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, ver
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

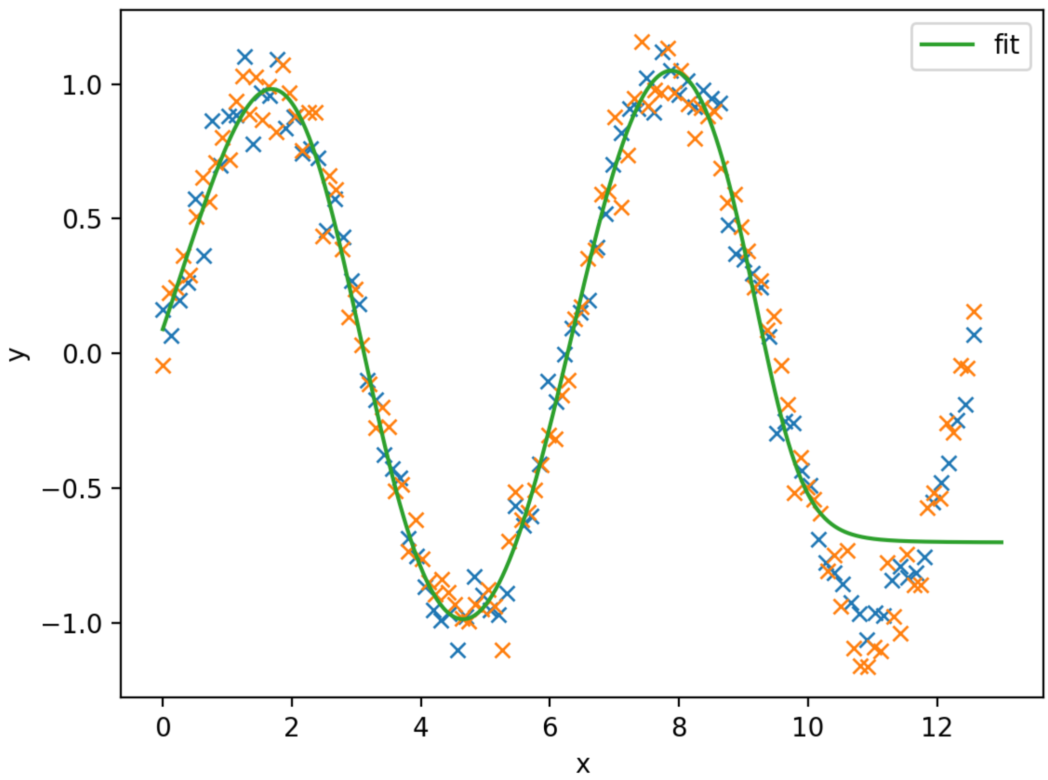
```
Repetition  0: best_cost=0.373
Repetition  1: best_cost=0.373
Repetition  2: best_cost=0.373
Repetition  3: best_cost=0.373
Repetition  4: best_cost=0.373
Repetition  5: best_cost=0.373
Repetition  6: best_cost=0.373
Repetition  7: best_cost=0.373
Repetition  8: best_cost=0.373
Repetition  9: best_cost=0.373
```



In [135]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adam(model.parameters(), lr=0.01), **kwargs
)
model = train3([3, 2, 2], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, ver
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

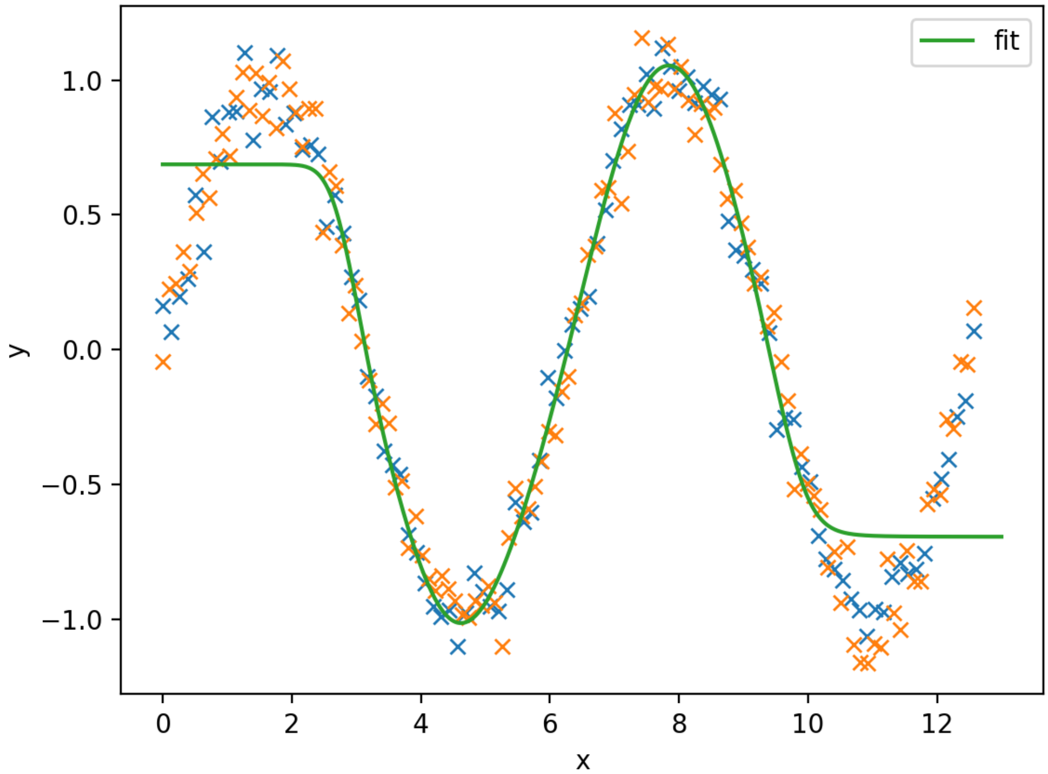
```
Repetition  0: best_cost=0.260
Repetition  1: best_cost=0.260
Repetition  2: best_cost=0.260
Repetition  3: best_cost=0.260
Repetition  4: best_cost=0.260
Repetition  5: best_cost=0.025
Repetition  6: best_cost=0.025
Repetition  7: best_cost=0.025
Repetition  8: best_cost=0.025
Repetition  9: best_cost=0.025
```



In [133]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adam(model.parameters(), lr=0.01), **kwargs
)
model = train3([3, 3, 3, 3, 3], nreps=10, train=train_adam, max_epochs=5000, tol=1e-
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

```
Repetition  0: best_cost=0.373
Repetition  1: best_cost=0.041
Repetition  2: best_cost=0.041
Repetition  3: best_cost=0.041
Repetition  4: best_cost=0.041
Repetition  5: best_cost=0.041
Repetition  6: best_cost=0.041
Repetition  7: best_cost=0.041
Repetition  8: best_cost=0.041
Repetition  9: best_cost=0.041
```

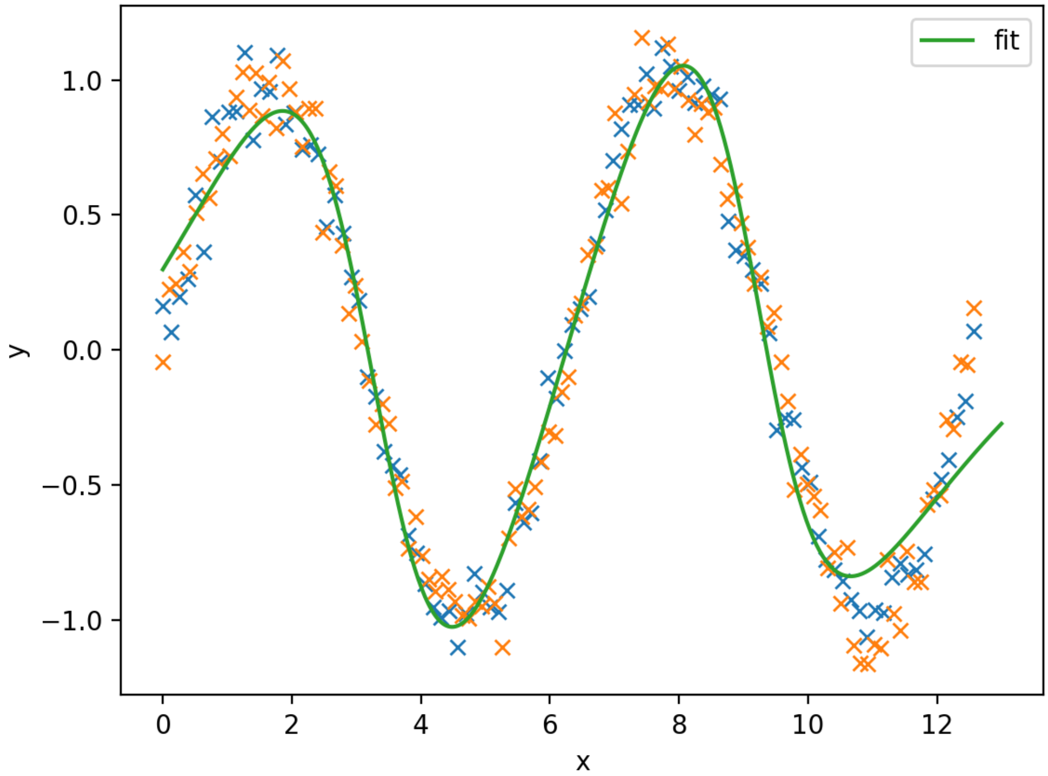



## Adam

In [123]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adam(model.parameters(), lr=0.01), **kwargs
)
model = train3([3], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, verbose=F
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

```
Repetition  0: best_cost=0.018
Repetition  1: best_cost=0.018
Repetition  2: best_cost=0.018
Repetition  3: best_cost=0.018
Repetition  4: best_cost=0.018
Repetition  5: best_cost=0.018
Repetition  6: best_cost=0.018
Repetition  7: best_cost=0.018
Repetition  8: best_cost=0.018
Repetition  9: best_cost=0.018
```

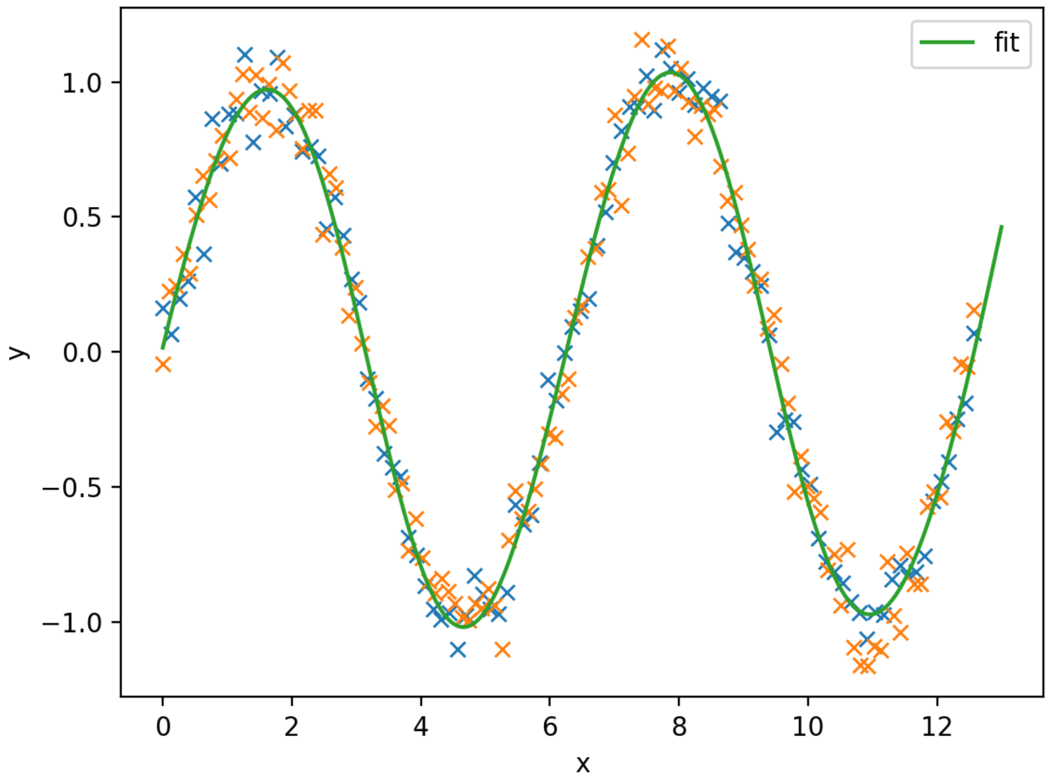



## LBFGS

In [124]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.LBFGS(model.parameters(), lr=0.01), **kwarg
)
model = train3([3], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, verbose=F
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

```
Repetition  0: best_cost=0.357
Repetition  1: best_cost=0.007
Repetition  2: best_cost=0.007
Repetition  3: best_cost=0.007
Repetition  4: best_cost=0.007
Repetition  5: best_cost=0.007
Repetition  6: best_cost=0.007
Repetition  7: best_cost=0.007
Repetition  8: best_cost=0.007
Repetition  9: best_cost=0.007
```

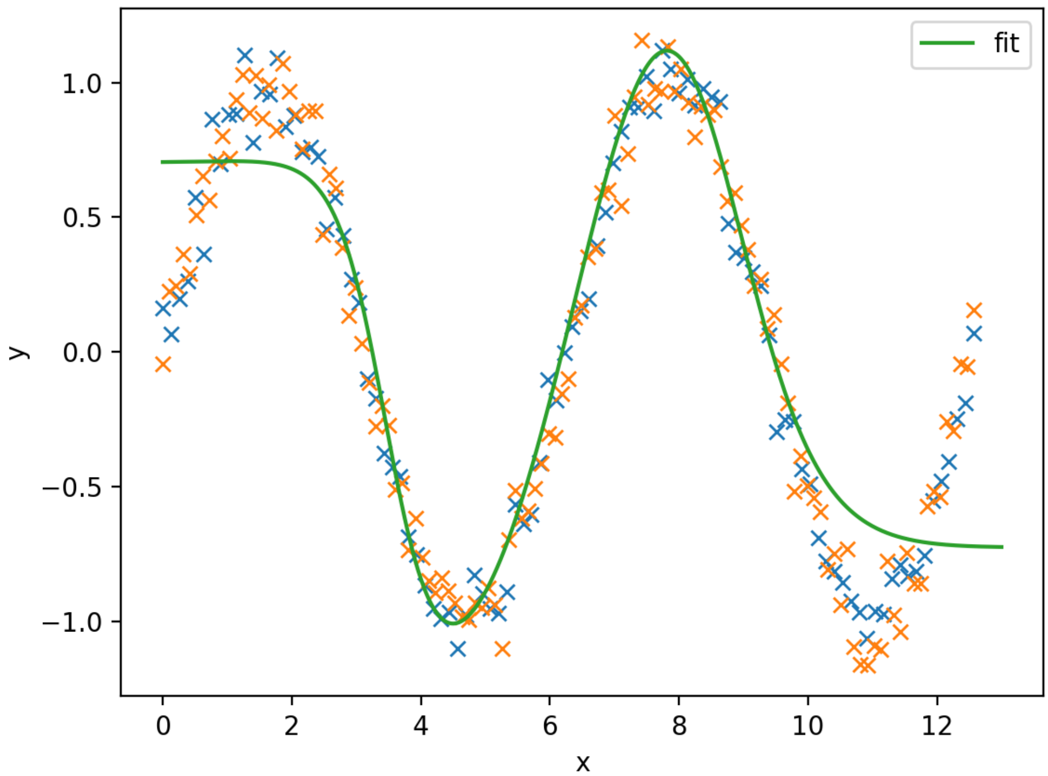



## RMSprop

In [125]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.RMSprop(model.parameters(), lr=0.01), **kwa
)
model = train3([3], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, verbose=F
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

```
Repetition  0: best_cost=0.065
Repetition  1: best_cost=0.065
Repetition  2: best_cost=0.050
Repetition  3: best_cost=0.049
Repetition  4: best_cost=0.049
Repetition  5: best_cost=0.049
Repetition  6: best_cost=0.049
Repetition  7: best_cost=0.049
Repetition  8: best_cost=0.049
Repetition  9: best_cost=0.049
```

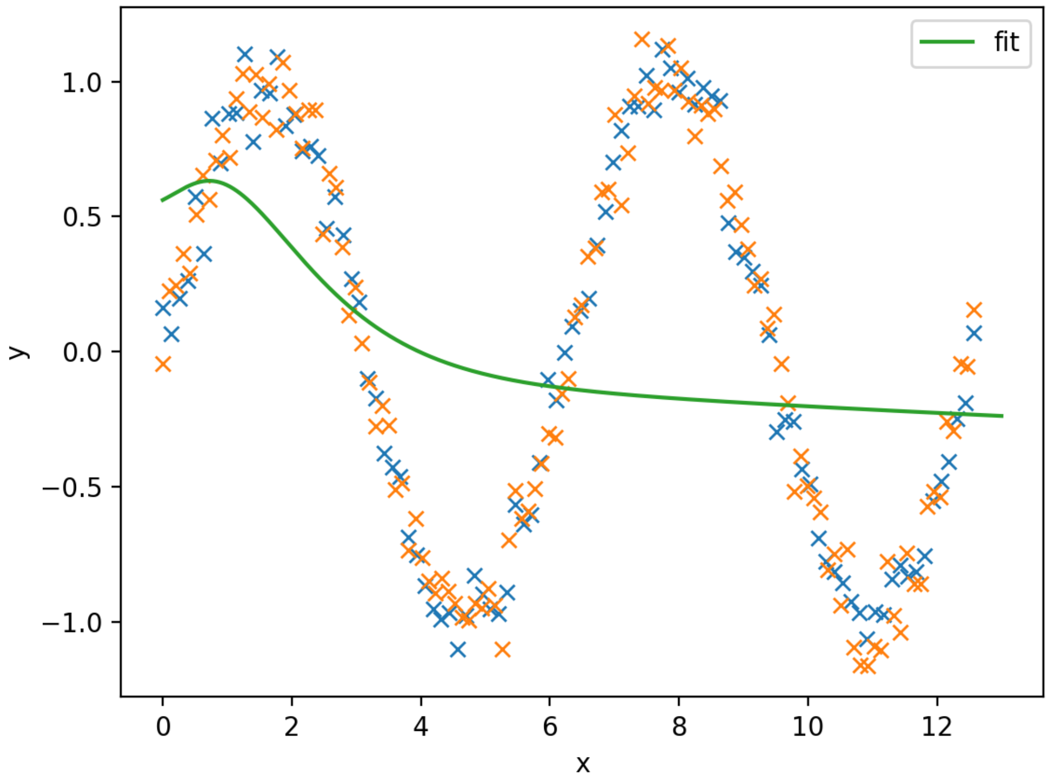



## Adagrad

In [126]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adagrad(model.parameters(), lr=0.01), **kwa
)
model = train3([3], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, verbose=F
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

```
Repetition  0: best_cost=0.440
Repetition  1: best_cost=0.426
Repetition  2: best_cost=0.426
Repetition  3: best_cost=0.419
Repetition  4: best_cost=0.412
Repetition  5: best_cost=0.412
Repetition  6: best_cost=0.412
Repetition  7: best_cost=0.412
Repetition  8: best_cost=0.412
Repetition  9: best_cost=0.412
```

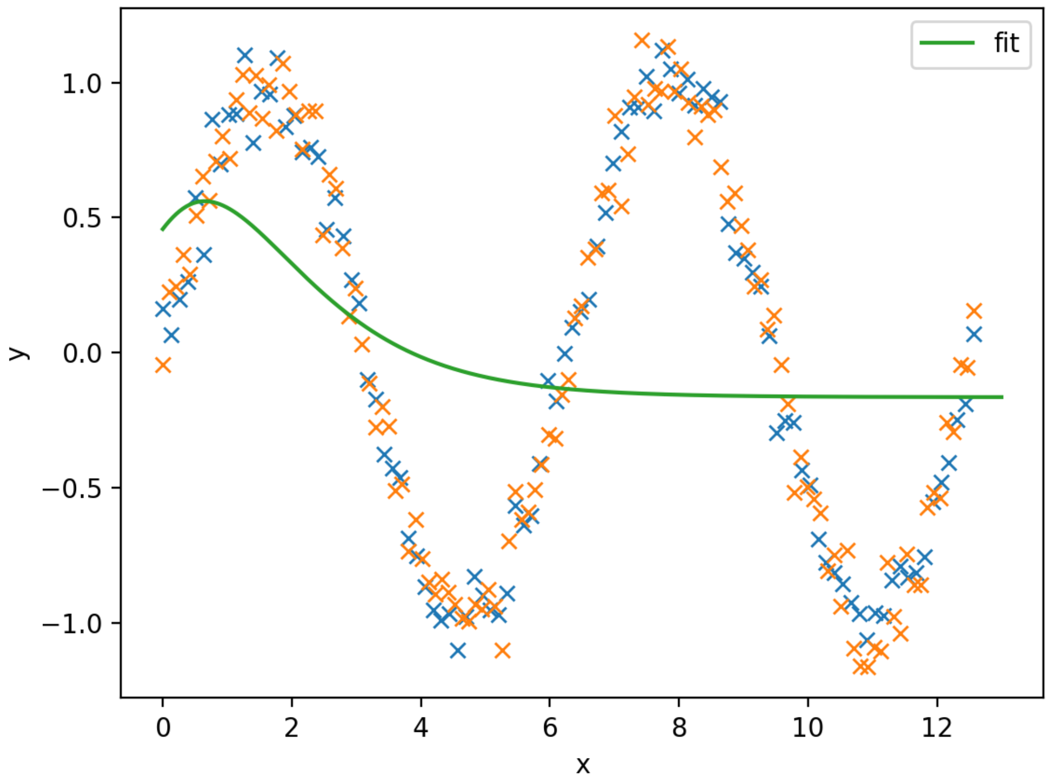



## Adadelta

In [127]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.Adadelta(model.parameters(), lr=0.01), **kw
)
model = train3([3], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, verbose=F
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

```
Repetition  0: best_cost=0.418
Repetition  1: best_cost=0.418
Repetition  2: best_cost=0.418
Repetition  3: best_cost=0.418
Repetition  4: best_cost=0.418
Repetition  5: best_cost=0.418
Repetition  6: best_cost=0.418
Repetition  7: best_cost=0.418
Repetition  8: best_cost=0.418
Repetition  9: best_cost=0.418
```

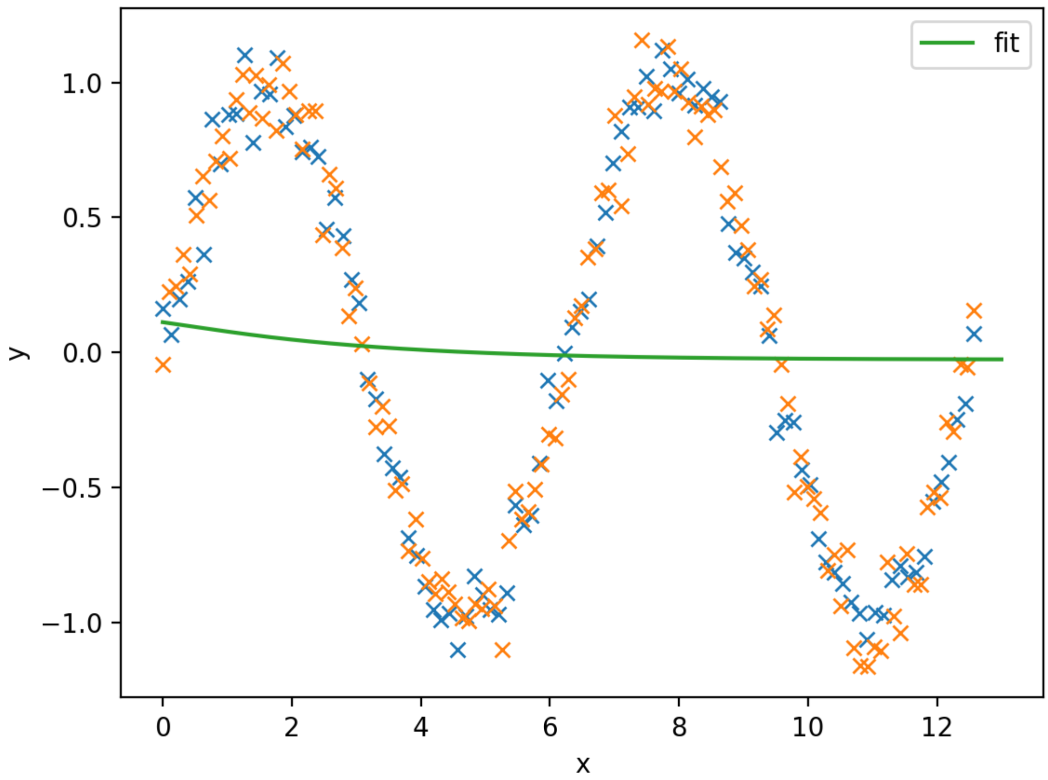



## SGD

In [128]:

```
# PyTorch provides many gradient-based optimizers; see
# https://pytorch.org/docs/stable/optim.html. You can use a PyTorch optimizer
# as follows.
train_adam = lambda model, **kwargs: fnn_train(
    X3, y3, model, optimizer=torch.optim.SGD(model.parameters(), lr=0.01), **kwargs
)
model = train3([2, 3, 2], nreps=10, train=train_adam, max_epochs=5000, tol=1e-8, ver
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
```

```
Repetition  0: best_cost=0.506
Repetition  1: best_cost=0.506
Repetition  2: best_cost=0.505
Repetition  3: best_cost=0.503
Repetition  4: best_cost=0.488
Repetition  5: best_cost=0.488
Repetition  6: best_cost=0.488
Repetition  7: best_cost=0.488
Repetition  8: best_cost=0.488
Repetition  9: best_cost=0.488
```



In [ ]:

```
# Experiment with different number of layers and activation functions. Here is
# an example with three hidden layers (of sizes 4, 5, and 6) and ReLU activations.
#
# You can also plot the outputs of the hidden neurons in the first layer (using
# the same code above).
model = train3([5,10,50], nreps=10, transfer=lambda: nn.ReLU())
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

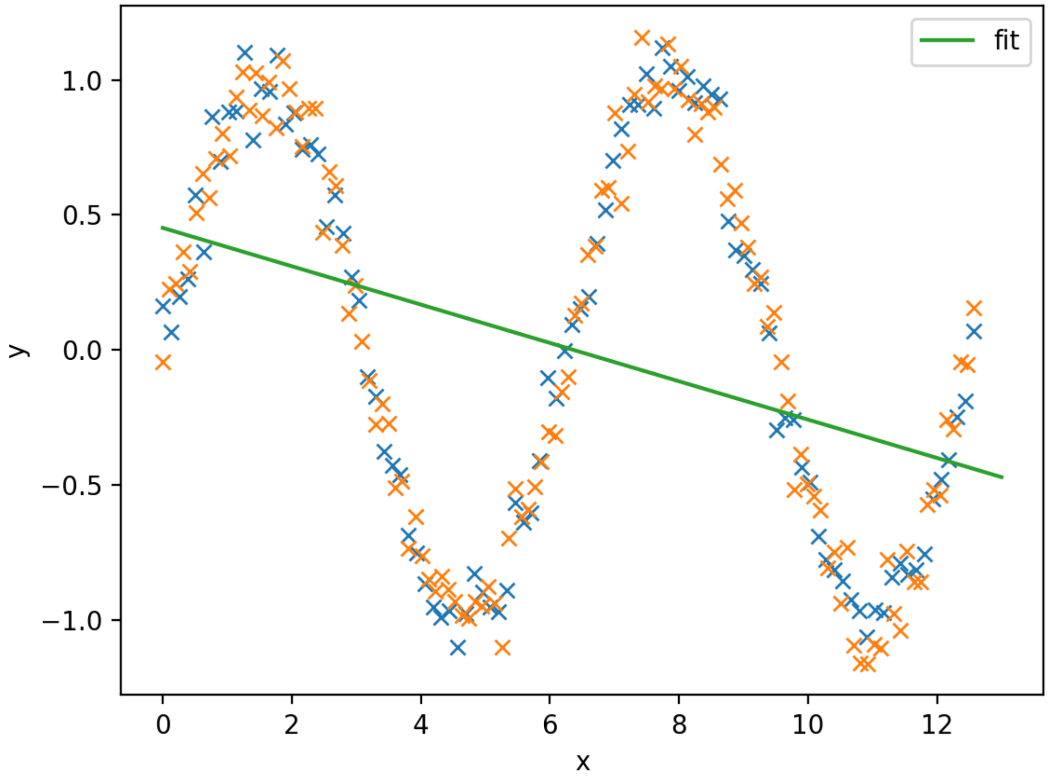
## Trying 1 hidden layer with a varying number of ReLUs

In [111]:

```
model = train3([1], nreps=10, transfer=lambda: nn.ReLU())
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 1
         Function evaluations: 3
         Gradient evaluations: 3
best_cost=0.506
Repetition  1: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 2
         Function evaluations: 3
         Gradient evaluations: 3
best_cost=0.506
Repetition  2: Optimization terminated successfully.
         Current function value: 0.438543
         Iterations: 10
         Function evaluations: 16
         Gradient evaluations: 16
best_cost=0.439
Repetition  3: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 10
         Function evaluations: 12
         Gradient evaluations: 12
best_cost=0.439
Repetition  4: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 2
         Function evaluations: 4
         Gradient evaluations: 4
best_cost=0.439
Repetition  5: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 2
         Function evaluations: 4
         Gradient evaluations: 4
best_cost=0.439
Repetition  6: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 2
         Function evaluations: 4
         Gradient evaluations: 4
best_cost=0.439
Repetition  7: Optimization terminated successfully.
         Current function value: 0.505991
         Iterations: 24
         Function evaluations: 27
         Gradient evaluations: 27
best_cost=0.439
Repetition  8: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 1
```

```
          Function evaluations: 3
          Gradient evaluations: 3
best_cost=0.439
Repetition  9: Optimization terminated successfully.
          Current function value: 0.438543
          Iterations: 8
          Function evaluations: 10
          Gradient evaluations: 10
best_cost=0.439
```



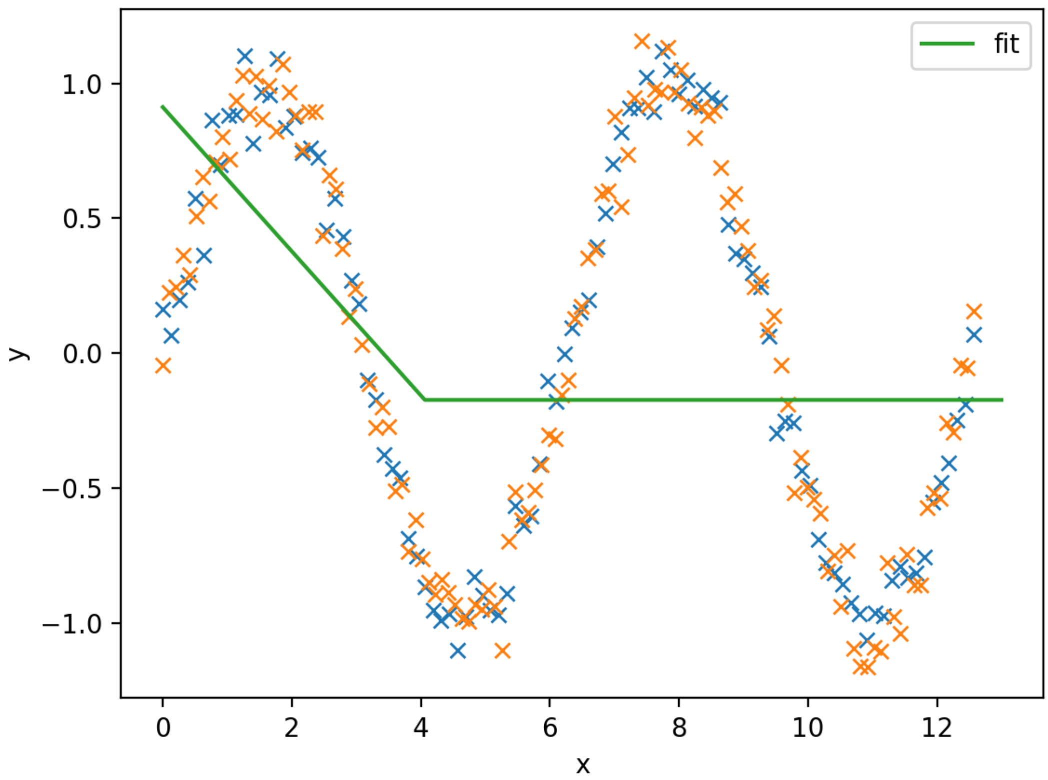

```
Training error: 0.43854334950447 08
Test error    : 0.4407091736793518
```

In [112]:

```
model = train3([2], nreps=10, transfer=lambda: nn.ReLU())
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406719
         Iterations: 22
         Function evaluations: 120
         Gradient evaluations: 112
best_cost=0.407
Repetition  1: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 8
         Function evaluations: 9
         Gradient evaluations: 9
best_cost=0.407
Repetition  2: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406719
         Iterations: 17
         Function evaluations: 130
         Gradient evaluations: 121
best_cost=0.407
Repetition  3: Optimization terminated successfully.
         Current function value: 0.506238
         Iterations: 2
         Function evaluations: 4
         Gradient evaluations: 4
best_cost=0.407
Repetition  4: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406723
         Iterations: 10
         Function evaluations: 85
         Gradient evaluations: 79
best_cost=0.407
Repetition  5: Optimization terminated successfully.
         Current function value: 0.435275
         Iterations: 49
         Function evaluations: 67
         Gradient evaluations: 67
best_cost=0.407
Repetition  6: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.504720
         Iterations: 5
         Function evaluations: 96
         Gradient evaluations: 92
best_cost=0.407
Repetition  7: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406720
         Iterations: 20
         Function evaluations: 98
```

```
          Gradient evaluations: 90
best_cost=0.407
Repetition  8: Optimization terminated successfully.
          Current function value: 0.437668
          Iterations: 17
          Function evaluations: 19
          Gradient evaluations: 19
best_cost=0.407
Repetition  9: Optimization terminated successfully.
          Current function value: 0.506238
          Iterations: 3
          Function evaluations: 5
          Gradient evaluations: 5
best_cost=0.407
```



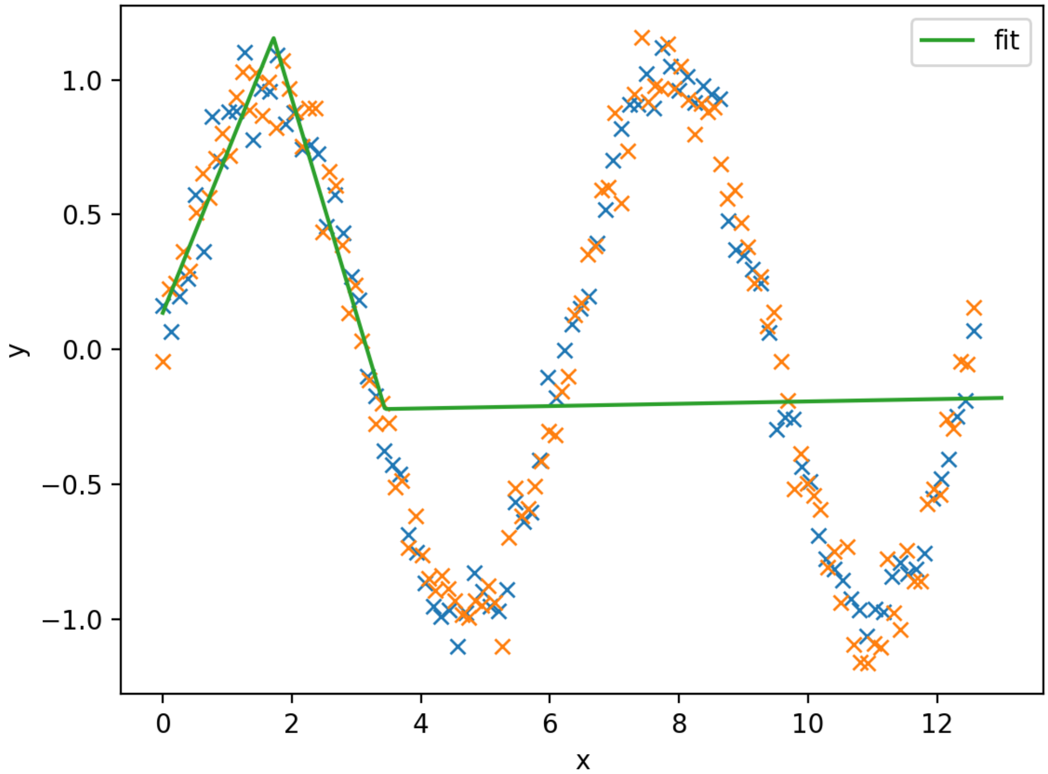
```
Training error: 0.4067193567752838
Test error    : 0.40883004665374756
```

In [113]:

```
model = train3([3], nreps=10, transfer=lambda: nn.ReLU())
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406722
         Iterations: 13
         Function evaluations: 101
         Gradient evaluations: 94
best_cost=0.407
Repetition  1: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.437669
         Iterations: 23
         Function evaluations: 112
         Gradient evaluations: 106
best_cost=0.407
Repetition  2: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.357718
         Iterations: 27
         Function evaluations: 118
         Gradient evaluations: 115
best_cost=0.358
Repetition  3: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406719
         Iterations: 18
         Function evaluations: 94
         Gradient evaluations: 85
best_cost=0.358
Repetition  4: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406720
         Iterations: 16
         Function evaluations: 92
         Gradient evaluations: 85
best_cost=0.358
Repetition  5: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406723
         Iterations: 14
         Function evaluations: 100
         Gradient evaluations: 95
best_cost=0.358
Repetition  6: Optimization terminated successfully.
         Current function value: 0.357620
         Iterations: 34
         Function evaluations: 40
         Gradient evaluations: 40
best_cost=0.358
Repetition  7: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406722
```

```
         Iterations: 30
         Function evaluations: 120
         Gradient evaluations: 112
best_cost=0.358
Repetition  8: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406735
         Iterations: 11
         Function evaluations: 84
         Gradient evaluations: 76
best_cost=0.358
Repetition  9: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.406724
         Iterations: 20
         Function evaluations: 98
         Gradient evaluations: 96
best_cost=0.358
```



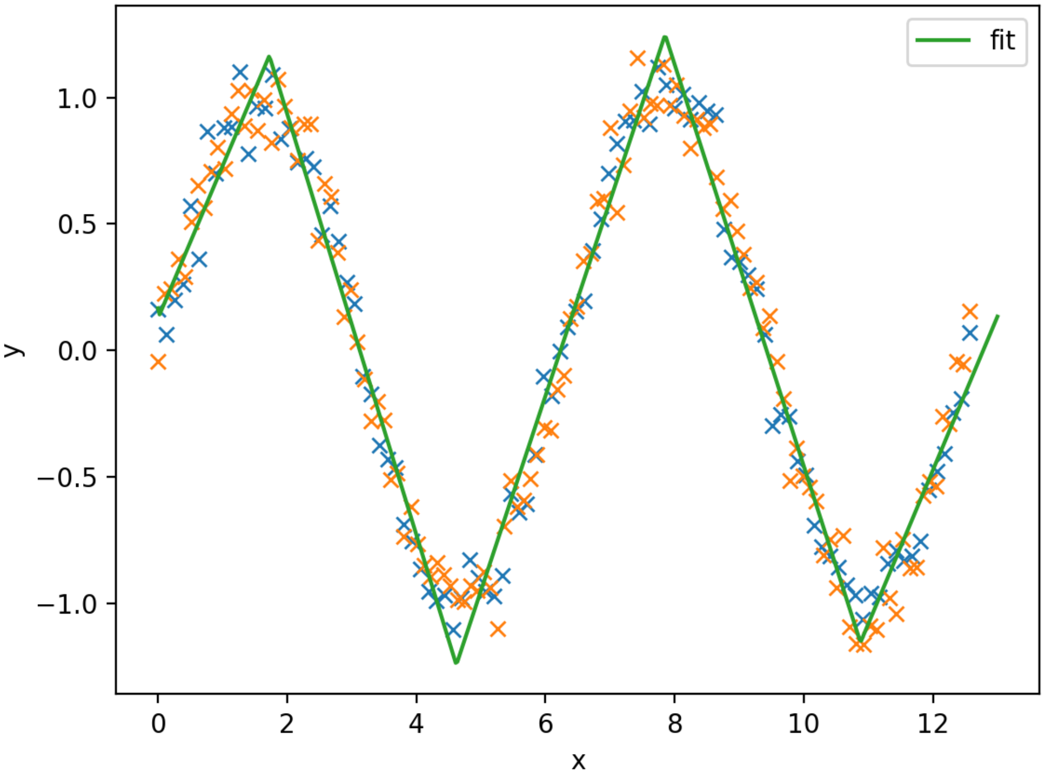

```
Training error: 0.35762014985084534
Test error    : 0.360356867313385
```

In [114]:

```
model = train3([10], nreps=10, transfer=lambda: nn.ReLU())
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.309325
         Iterations: 35
         Function evaluations: 117
         Gradient evaluations: 108
best_cost=0.309
Repetition  1: Optimization terminated successfully.
         Current function value: 0.085044
         Iterations: 107
         Function evaluations: 197
         Gradient evaluations: 196
best_cost=0.085
Repetition  2: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.084857
         Iterations: 145
         Function evaluations: 271
         Gradient evaluations: 267
best_cost=0.085
Repetition  3: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.082022
         Iterations: 110
         Function evaluations: 216
         Gradient evaluations: 207
best_cost=0.082
Repetition  4: Optimization terminated successfully.
         Current function value: 0.356643
         Iterations: 68
         Function evaluations: 80
         Gradient evaluations: 80
best_cost=0.082
Repetition  5: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.013686
         Iterations: 89
         Function evaluations: 165
         Gradient evaluations: 159
best_cost=0.014
Repetition  6: Optimization terminated successfully.
         Current function value: 0.356162
         Iterations: 47
         Function evaluations: 56
         Gradient evaluations: 56
best_cost=0.014
Repetition  7: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.356805
         Iterations: 29
         Function evaluations: 121
```

```
          Gradient evaluations: 112
best_cost=0.014
Repetition  8: Warning: Desired error not necessarily achieved due to
precision loss.
          Current function value: 0.081694
          Iterations: 111
          Function evaluations: 204
          Gradient evaluations: 197
best_cost=0.014
Repetition  9: Warning: Desired error not necessarily achieved due to
precision loss.
          Current function value: 0.356233
          Iterations: 32
          Function evaluations: 121
          Gradient evaluations: 115
best_cost=0.014
```



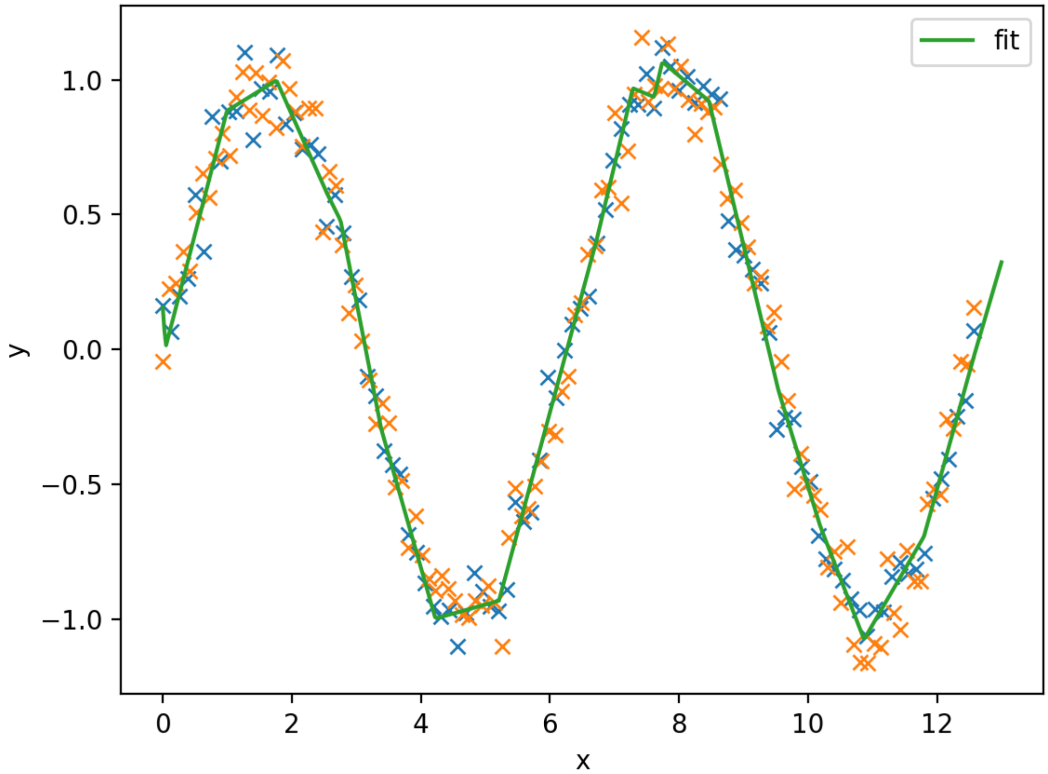

```
Training error: 0.013686132617294788
Test error    : 0.017258409410715103
```

In [115]:

```
model = train3([50], nreps=10, transfer=lambda: nn.ReLU())
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.009470
         Iterations: 197
         Function evaluations: 328
         Gradient evaluations: 324
best_cost=0.009
Repetition  1: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.005514
         Iterations: 335
         Function evaluations: 470
         Gradient evaluations: 463
best_cost=0.006
Repetition  2: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.007878
         Iterations: 135
         Function evaluations: 234
         Gradient evaluations: 227
best_cost=0.006
Repetition  3: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.081621
         Iterations: 92
         Function evaluations: 195
         Gradient evaluations: 190
best_cost=0.006
Repetition  4: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.081450
         Iterations: 192
         Function evaluations: 298
         Gradient evaluations: 292
best_cost=0.006
Repetition  5: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.006917
         Iterations: 143
         Function evaluations: 235
         Gradient evaluations: 228
best_cost=0.006
Repetition  6: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.008208
         Iterations: 201
         Function evaluations: 328
         Gradient evaluations: 324
best_cost=0.006
Repetition  7: Warning: Desired error not necessarily achieved due to
precision loss.
```

```
         Current function value: 0.006437
         Iterations: 371
         Function evaluations: 529
         Gradient evaluations: 524
best_cost=0.006
Repetition  8: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.081728
         Iterations: 126
         Function evaluations: 218
         Gradient evaluations: 210
best_cost=0.006
Repetition  9: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.081937
         Iterations: 51
         Function evaluations: 151
         Gradient evaluations: 144
best_cost=0.006
```



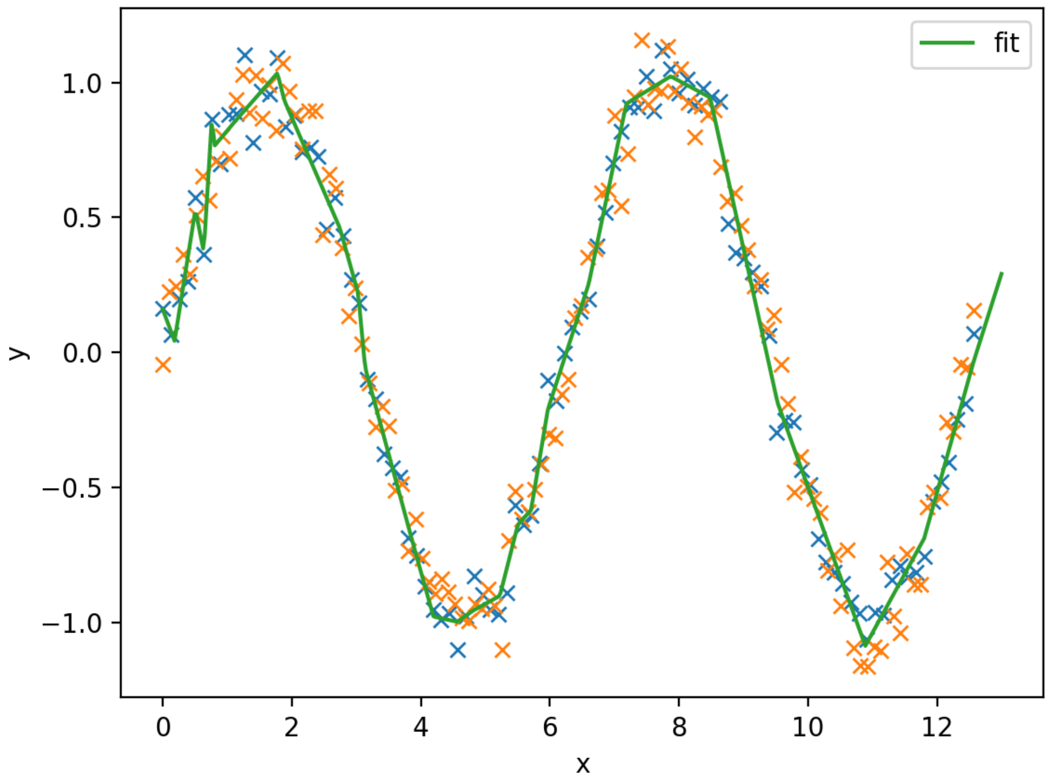

```
Training error: 0.005512818694114685
Test error    : 0.011191939003765583
```

In [116]:

```
model = train3([100], nreps=10, transfer=lambda: nn.ReLU())
nextplot()
plot1(X3, y3, label="train")
plot1(X3test, y3test, label="test")
plot1fit(torch.linspace(0, 13, 500).unsqueeze(1), model)
print("Training error:", F.mse_loss(y3, model(X3)).item())
print("Test error    :", F.mse_loss(y3test, model(X3test)).item())
```

```
Repetition  0: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.005350
         Iterations: 297
         Function evaluations: 415
         Gradient evaluations: 408
best_cost=0.005
Repetition  1: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.081523
         Iterations: 122
         Function evaluations: 189
         Gradient evaluations: 182
best_cost=0.005
Repetition  2: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.081756
         Iterations: 78
         Function evaluations: 157
         Gradient evaluations: 153
best_cost=0.005
Repetition  3: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.005301
         Iterations: 372
         Function evaluations: 496
         Gradient evaluations: 489
best_cost=0.005
Repetition  4: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.005127
         Iterations: 382
         Function evaluations: 470
         Gradient evaluations: 462
best_cost=0.005
Repetition  5: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.005984
         Iterations: 339
         Function evaluations: 459
         Gradient evaluations: 450
best_cost=0.005
Repetition  6: Warning: Desired error not necessarily achieved due to
precision loss.
         Current function value: 0.004477
         Iterations: 542
         Function evaluations: 703
         Gradient evaluations: 696
best_cost=0.004
Repetition  7: Warning: Desired error not necessarily achieved due to
precision loss.
```

```
          Current function value: 0.080794
          Iterations: 261
          Function evaluations: 365
          Gradient evaluations: 361
best_cost=0.004
Repetition  8: Warning: Desired error not necessarily achieved due to
precision loss.
          Current function value: 0.006943
          Iterations: 196
          Function evaluations: 323
          Gradient evaluations: 317
best_cost=0.004
Repetition  9: Warning: Desired error not necessarily achieved due to
precision loss.
          Current function value: 0.006016
          Iterations: 225
          Function evaluations: 322
          Gradient evaluations: 318
best_cost=0.004
```




```
Training error: 0.004473547451198101
Test error    : 0.012758485972881317
```