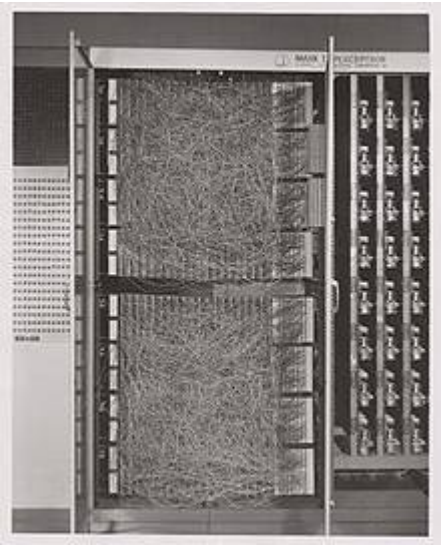
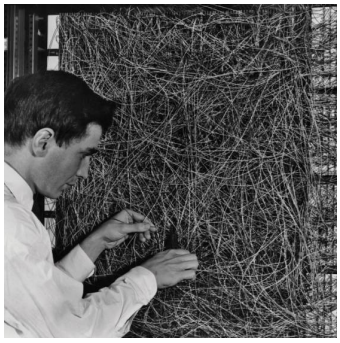
Notation: □ Means pencil-and-paper QUIZ ► Means coding QUIZ

Neural Networks (pp. 106-121)

The first artificial neural network (ANN) was the (single-layer) **perceptron**, a simplified model of a biological neuron.

"The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt, funded by the United States Office of Naval Research. The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron". This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors."[1]





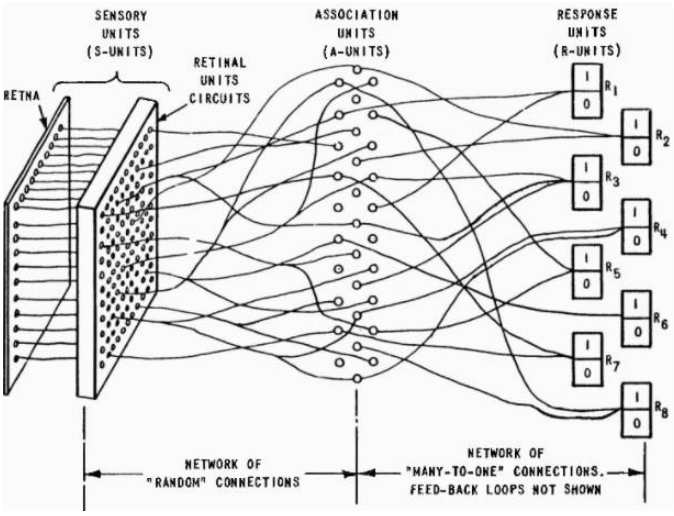


Image from Mark I Perceptron Operators' Manual[2]

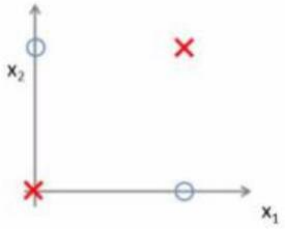
[1]https://en.wikipedia.org/wiki/Perceptron
[2]http://www.dtic.mil/dtic/tr/fulltext/u2/236965.pdf

Rosenblatt's perceptron is a **linear classifier**, similar to:

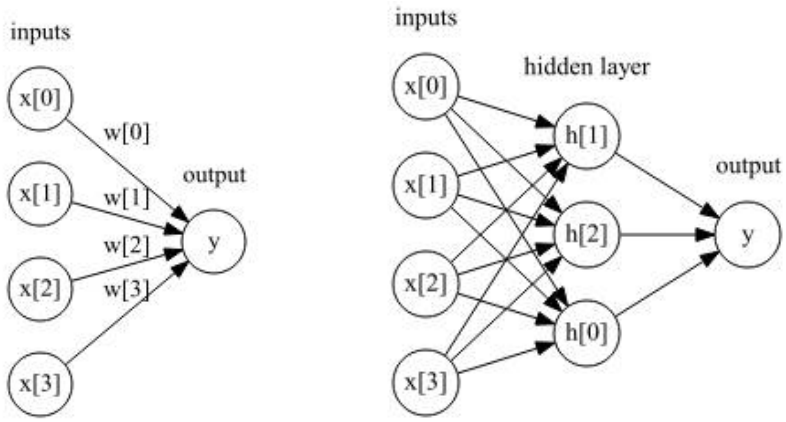
- the logistic classifier (but without the non-linear logistic function!)
- the linear SVM classifier (without the quadratic optimization needed to find the support vectors with the maximum margin!):

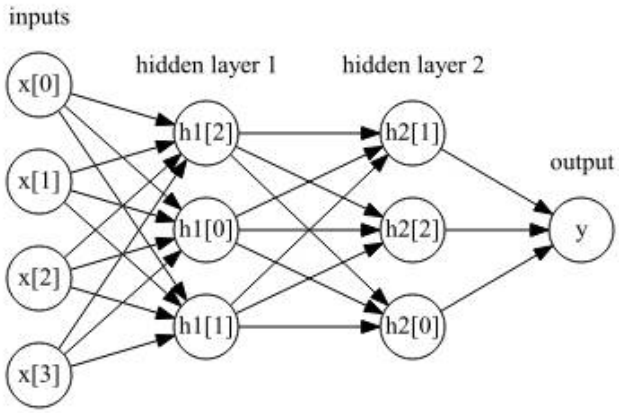
classification $\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$

As we know today, linear classifiers can only be successful on datatsets that are linearly separable. The most famous "failure" of the (single-layer) perceptron was pointed out at the time in a famous book[3]: The "XOR affair"



In contrast, a multi-layer perceptron (MLP), has at least one layer of "hidden" nodes/neurons:



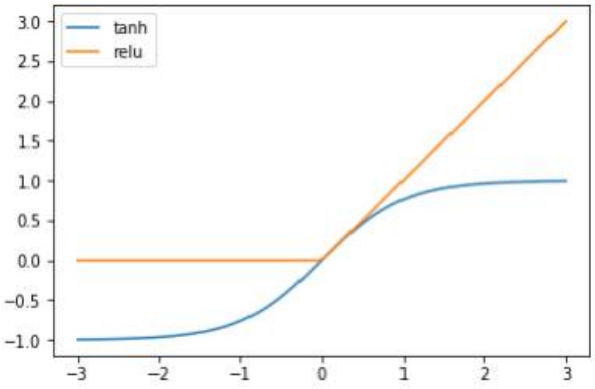


[3]https://en.wikipedia.org/wiki/Perceptrons_(book)

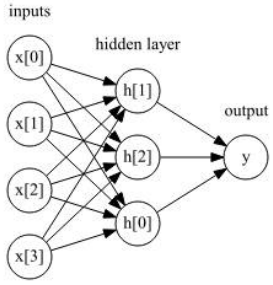
MLPs are also **non-linear** classifiers, because the output of the hidden nodes is "filtered" through a non-linear "activation function", e.g.

- hyperbolic tangent
- logistic/sigmoid function
- rectified linear unit ⇦ esp. useful in "deep" NNs!

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
```



► Plot the logistic function $f(z) = \frac{1}{1 + \exp(-z)}$ on the same graph, extending the x axis to -4 ... +4.



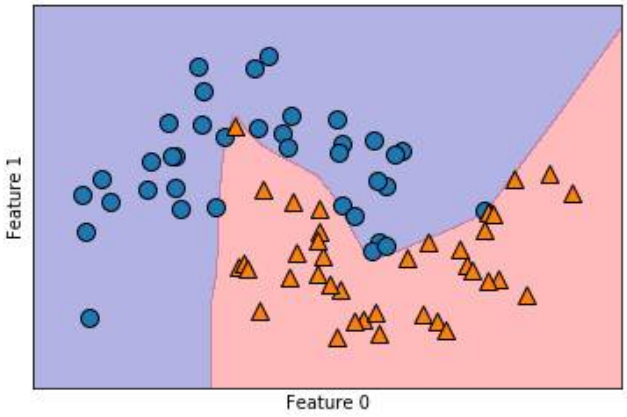
$$h[0] = \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[0])$$
$$h[1] = \tanh(w[0, 1] * x[0] + w[1, 1] * x[1] + w[2, 1] * x[2] + w[3, 1] * x[3] + b[1])$$
$$h[2] = \tanh(w[0, 2] * x[0] + w[1, 2] * x[1] + w[2, 2] * x[2] + w[3, 2] * x[3] + b[2])$$
$$\hat{y} = v[0] * h[0] + v[1] * h[1] + v[2] * h[2] + b$$

Important hyper-parameters:

- how many hidden layers?
- how many nodes in each hidden layer?

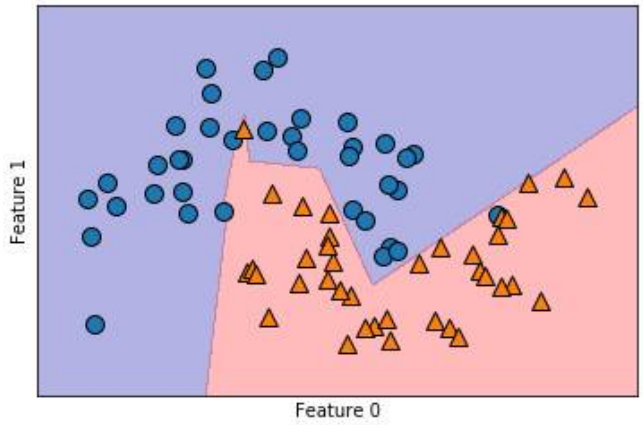
Due to their non-linearities, MLPs can easily learn non-linear boundaries (including the infamous XOR!)

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```



LBFGS stands for Limited-memory BFGS (Broyden–Fletcher–Goldfarb–Shanno), a class of gradient-descent algorithms. By default, it uses one hidden layer with 100 nodes, and *relu* for the activation function. We can try fewer nodes:

```
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
```



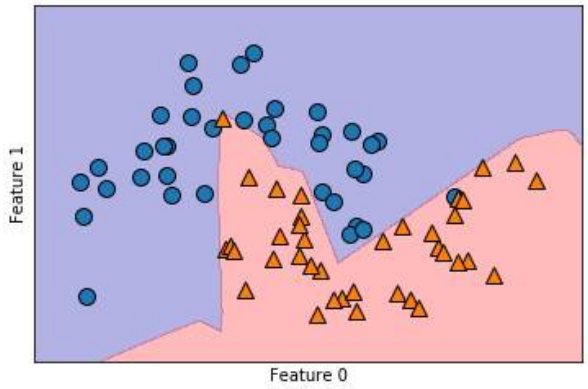
□ Remember overfitting? What is your verdict?

⇩

Fewer nodes reduce complexity/overfitting.

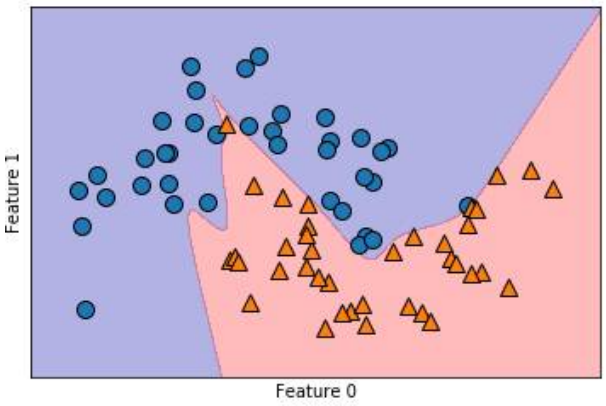
Or we can try multiple hidden layers:

```
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    hidden_layer_sizes=[10, 10])
```

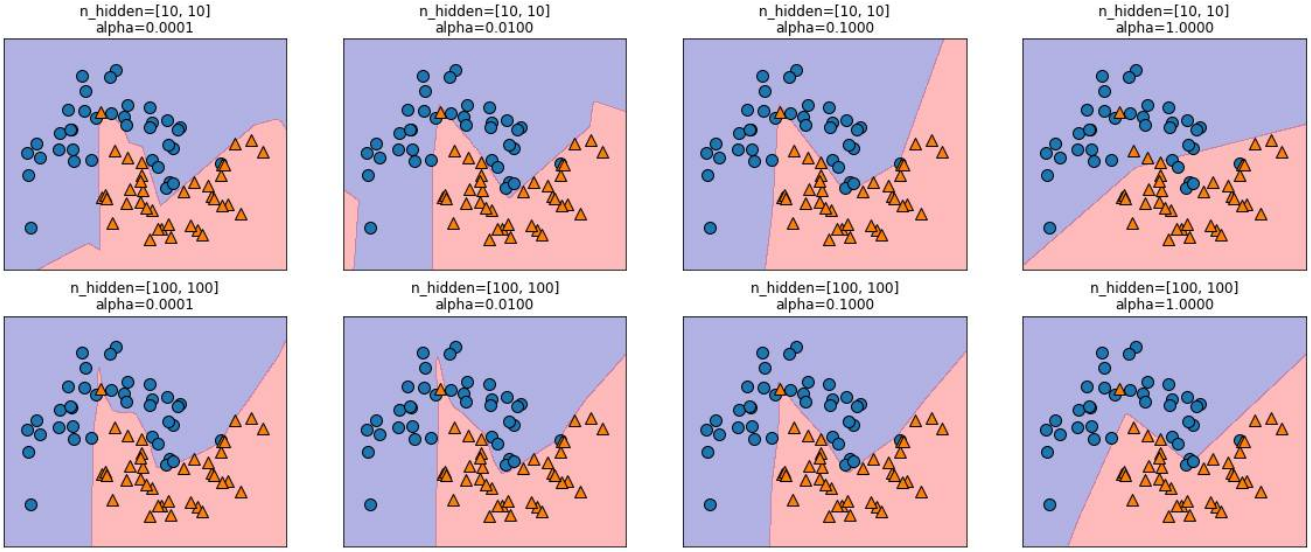


Or we can obtain a smooth boundary by replacing *relu* with *tanh* for the activation function:

```
lp = MLPClassifier(solver='lbfgs', activation='tanh',
                   random_state=0, hidden_layer_sizes=[10, 10])
```



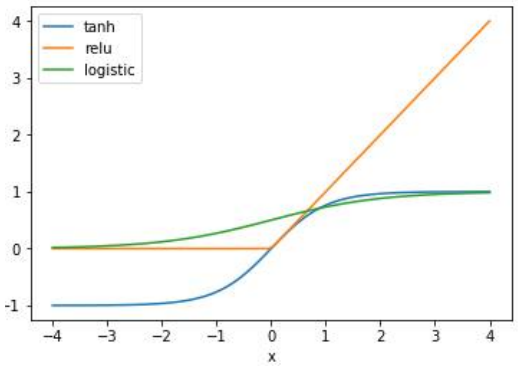
As in ridge and lasso regression, it is possible to reduce complexity/overfitting by penalizing the linear coefficients through the hyper-parameter *alpha*, whose default value is 10^{-4}:



Solutions:

► Plot the logistic function $f(z) = \frac{1}{1+\exp(-z)}$ on the same graph, extending the x axis to -4 ... +4

```
line = np.linspace(-4, 4, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.plot(line, 1/(1+np.exp(-line)), label="logistic")
plt.legend(loc="best")
plt.xlabel("x")
```



■

Review questions for ANNs:

1. What does the acronym ANN stand for?
2. What was the first ANN ever built, by whom, and in what decade?
3. What is an activation function? Give three examples.
4. What are the differences between a single-layer and multi-layer perceptron (MLP)?
5. How are MLPs implemented in Scikit-learn?
6. How many layers and what number of neurons in each layer does the default MLPClassifier have in Scikit-learn?
7. What hyper-parameters can we use to control model complexity in MLPClassifier?

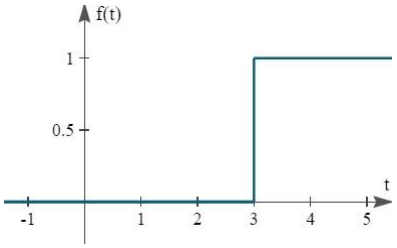
---------- We are on p.115 of our text ----------

---------- Start of material not in text ----------

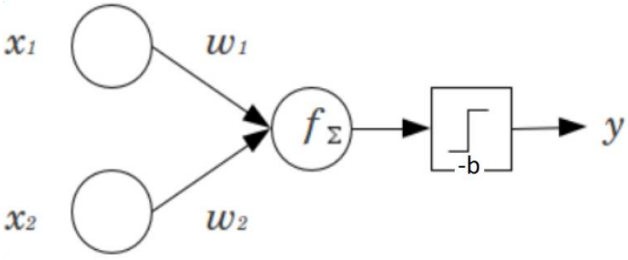
Understading ANN operation through examples (not in text)

Before, we had this formula for linear classification: $\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b$, where b is called the *bias*. If the entire sum above is <0, then the answer is 0 (class 0), and if it is >0, then the answer is 1 (class 1).

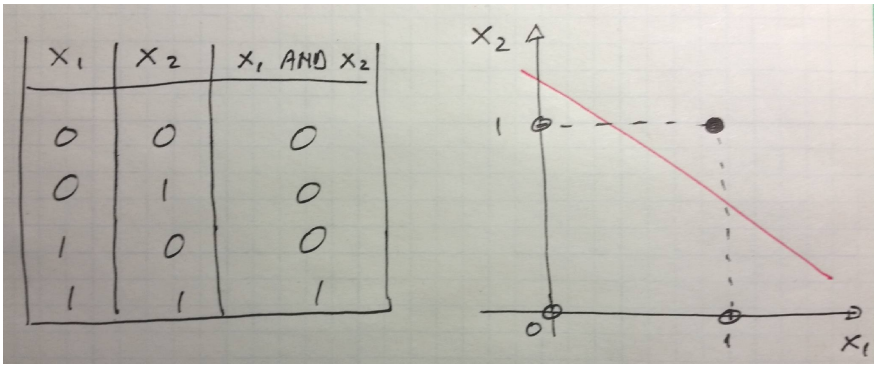
We can subtract the bias, so we compare only the sum of products with -b. Conceptually, this is equivalent to applying to the result a *step function*, with the step located at -b. To avoid the negative sign, it is customary to rename -b = θ (the greek letter theta). In the figure below, we have a step function with -b = θ = 3:



As the first example, we are using this single-layer perceptron with only two inputs:



We want to correctly classify this data set, equivalent to the logical function AND:



The initial values of the parameters are: w1 = -0.5, w2 = 0.5, θ = 1.5. Starting from them, the perceptron learning algorithm proceeds iteratively, changing the weights and the threshold in each iteration thus:

θnew - θold = $\Delta\theta$ = output - target = -d

Δw1 = (target - output)··x1 = d·x1

Δw2 = (target - output)··x2 = d·x2

□ Fill out the boxes in the table below to perform the first iteration:

x_1	x_2	y	target	w_1	w_2	θ
1	1	?	1	-0.5	0.5	1.5
				?	?	?

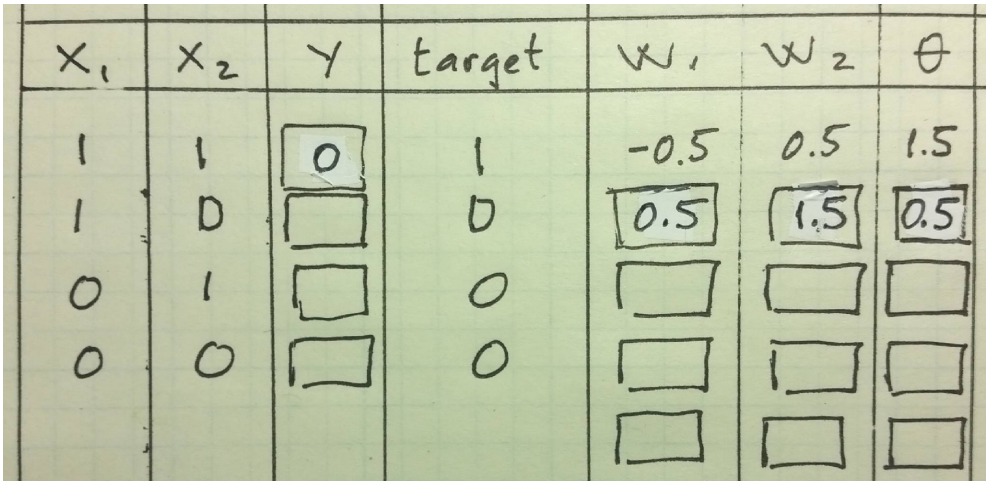
⇩

x_1	x_2	y	target	w_1	w_2	θ
1	1	0	1	-0.5	0.5	1.5
1	1		1	0.5	1.5	0.5

□ Do you understand why the signs are opposite in the threshold and weights formulas above?

□ What is the output if we now apply the same combination of inputs, (1, 1)? Do we need to continue with the algorithm?

□ Perform the next 3 iterations with the other 3 data points, in the order shown:



x_1	x_2	y	target	w_1	w_2	θ
1	1	0	1	-0.5	0.5	1.5
1	0		0	0.5	1.5	0.5
0	1		0			
0	0		0			

► Store the dataset in a Numpy array named **andy**, in the same order as above, i.e.

[[1, 1, 1], [1, 0, 0], [0, 1, 0], [0, 0, 0]]

► Write a Python function **perc_out( )** to calculate the output of our perceptron. The function takes a row from the array **andy** as argument, and it returns the output value. Implement w1 = -0.5, w2 = 0.5, θ = 1.5 as global variables.

► Write a Python function **perc_iter( )** to implement one iteration of the algorithm. Pass **x1**, **x2**, and **target** as function arguments.

The function calculates the difference **d** and updates the globals w1, w2, θ based on it.

The function returns a new triplet w1, w2, θ (as a tuple).

Call the function in a loop over the rows of **andy** and verify the manual calculations above.

► Write a Python function **perc_pred( )** to predict the class for any pair of inputs.

Call the function in a loop over the rows of **andy** and decide if if classifies the training data correctly.

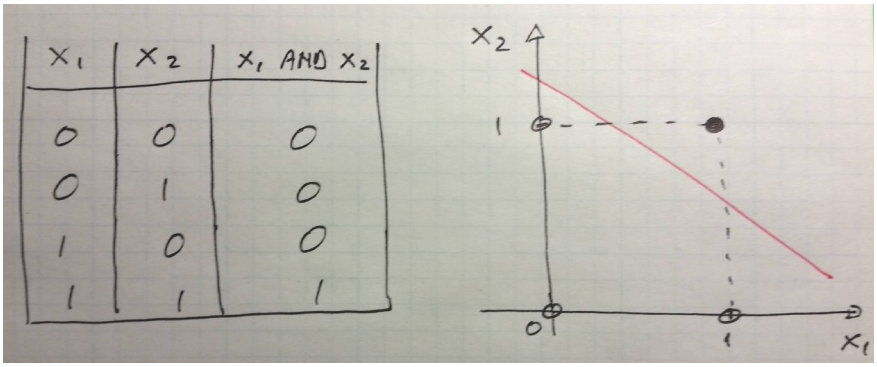
■

We start from the program developed in the previous session.

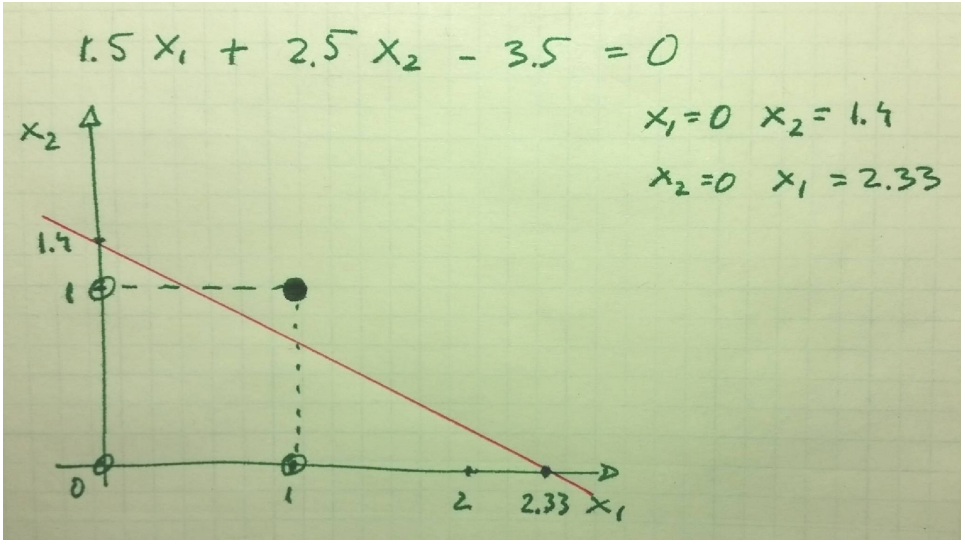
► Run the program until all the inputs are classified correctly. It is recommended to organize the iterations in "passes", where each "pass" iterates through the entire dataset (4 data points in our case).

How many passes are needed?

□ Draw the (linear) boundary corresponding to the solution above:



⇩



Follow one parameter (any one, e.g. w1) through the iterations. We notice that the parameters take large "swings", oscillating up and down before they reach the solution. This problem only gets worse when we increase the number of data points, so it will take a lot ot iterations to reach the solution.

To mitigate this problem, we update the parameters with a reduced quantity delta; the reduction factor is called the *learning rate*, **r**:

$\Delta\theta$ = r·(output - target) = -r·d

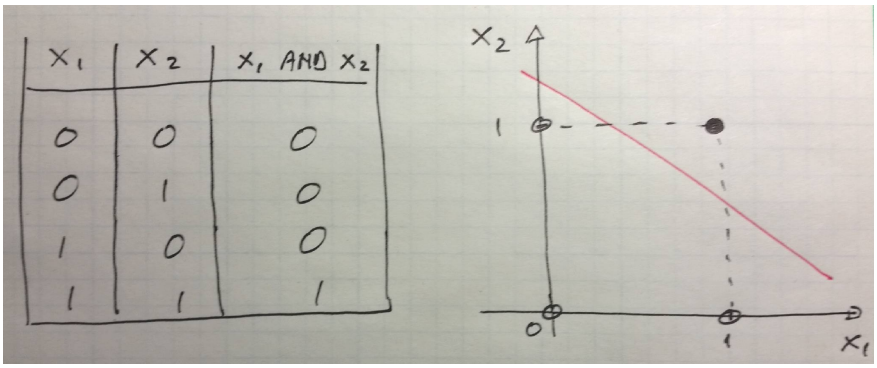
Δw1 = r·(target - output)·x1 = r·d·x1

Δw2 = r·(target - output)·x2 = r·d·x2

► Modify the program to implement the new update rules shown above. **r** is a hyper-parameter, implemented as another global variable.

How many passes are required for convergence now?

□ Draw the (linear) boundary corresponding to the new solution above:



□ Which solution is closer to the one that a SVM would find?

In Scikit-learn's **MLPClassifier**, the learning rate hyper-parameter is provided as two optional parameters of the constructor, **learning_rate** and **learning_rate_init**. Note that not all solvers use them:

learning_rate : {'constant', 'invscaling', 'adaptive'}, default 'constant'

Learning rate schedule for weight updates.

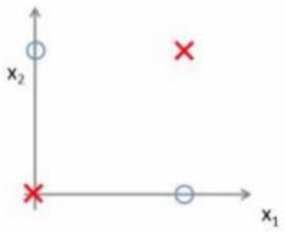
- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate `learning_rate_` at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when `solver='sgd'`.

learning_rate_init : double, optional, default 0.001

The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.

► Modify the program to use the **XOR** dataset instead:

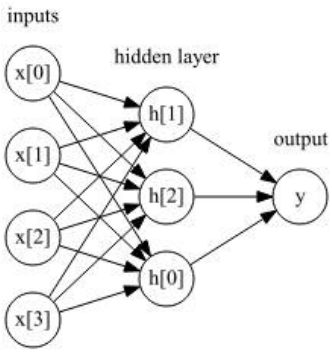


What do you notice?

The non-linear boundary problem can be solved in (at least) two ways:

1] Recoding the input (see next lab)

2] Adding hidden layers with non-linear activation functions⇩



The **delta rule**(s) for this two-layer ANN are as follows:

A. Rules for the weights going into the output node:

The update for the weight w_j connecting hidden node $\mathbf{h_j}$ to output node **o** is based on the quantity "delta":

- **Δ = (target – output)** if **o** does not have an activation function
- Δ = (target – output)*output*(1 – output) if **o** has sigmoid activation function
- Δ = (target – output)*(1+output)*(1 – output) if **o** has tanh activation function

In general, we can write

- Δ = (target – output)*act'(output), where act' is the derivative of the activation function. Use of the derivative makes BP a gradient descent algorithm.

$$\mathbf{\Delta w_j = \Delta * output\_of\_h_j}$$

Note: The (optional) bias of the output node, $\mathbf{b_{out}}$, is equivalent to a weight from a "virtual" node with an input of 1, so its update is:

- $\Delta b_{out} = \Delta * 1 = \Delta$

B. Rules for the weights going into the hidden nodes:

The update for the weight w_{ij} connecting input i to hidden node h_j is:

$\Delta w_{ij} = \Delta * w_j$ * act'(output_of_h_j) * x_i , where, as above, act' is the derivative of the activation function.
Unlike the output node, the hidden nodes must have (non-linear) activation functions, e.g. sigmoid or tanh.
For the sigmoid, we have:

$$\mathbf{\Delta w_{ij} = \Delta * w_j * output\_of\_h_j * (1 - output\_of\_h_j) * x_i}$$

Note: The (optional) bias of the hidden node, $\mathbf{b_j}$, is equivalent to a weight from a "virtual" node with an input of 1, so its update is (assuming sigmoid activation):

- $\Delta b_j = \Delta * wj$ * output_of_h_j*(1- output_of_h_j) * 1 = $\Delta * wj$ * output_of_h_j*(1- output_of_h_j)

Important:

- The outputs of all nodes are after squashing, i.e. after applying the activation function!
- When calculating Δw_{ij}, use the old value of w_j!
- When applying the updates, don't forget to multiply them by the learning rate!

---------- end delta rule ---------- to be applied in the following lab ----------

■

Lab 8 is in a separate file.

■

This session is using the program developed in the lab, **28_MLP_XOR_3hidden_rate_biases.py** , as starting point. The program implements a MLP with one hidden layer of 3 nodes. Ask the instructor for a copy if you do not have it.

► Modify the program to use only 2 nodes in the hidden layer. Run it multiple times; it is OK to stop if it hasn't reached a solution in 100,000 iterations.

What do you notice?

⇩

Execution 1: **1,000,000 iter. no convergence**

Execution 2:

```
############### Success on iteration 5045  ###########
outputs:2.598 0.4151 0.8221
Weights: 2.1 -1.85 2.6 0.407 3.52 1.62
Biases:  bout   b1     b2
        0.000  0.000  0.000
target   binary output   output
    0          0         0.46563
    1          1         0.84780
    1          1         0.50042
    0          0         0.12732
  Total Error = 0.50577120
```

Execution 3:

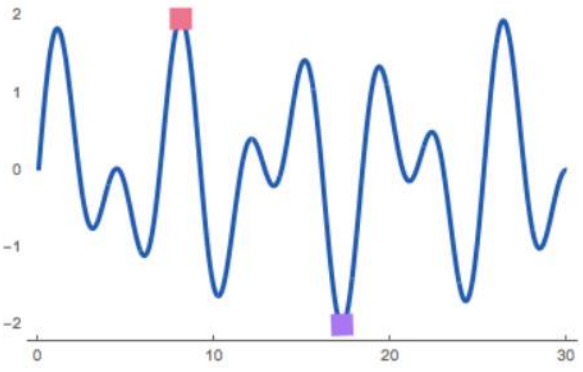
```
############### Success on iteration 3543  ###########
outputs:0.0 0.0 0.52
Weights: 2.44 -1.45 -0.473 -1.49 1.28 3.47
Biases:  bout   b1     b2
        0.000  0.000  0.000
target   binary output   output
    0          0         0.41006
    1          1         0.66910
    1          1         0.50010
    0          0         0.49401
  Total Error = 0.77158232
```

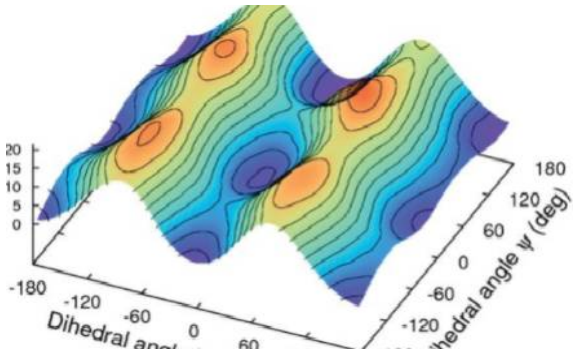
Execution 4: **1,000,000 iter. no convergence**

Execution 5:

```
############### Success on iteration 1840  ###########
outputs:-4.517 -1.033 0.5196
Weights: -1.58 2.03 -2.45 -0.623 -2.07 -0.451
Biases:  bout   b1     b2
        0.000  0.000  0.000
target   binary output   output
    0          0         0.49993
    1          1         0.58340
    1          1         0.61220
    0          0         0.22520
  Total Error = 0.62459199
```

This phenomenon is common in all search algorithms that are based on local conditions (e.g. gradient descent), and it is due to the existence of multiple local minima of the error function, which are not the same as the global minimum.





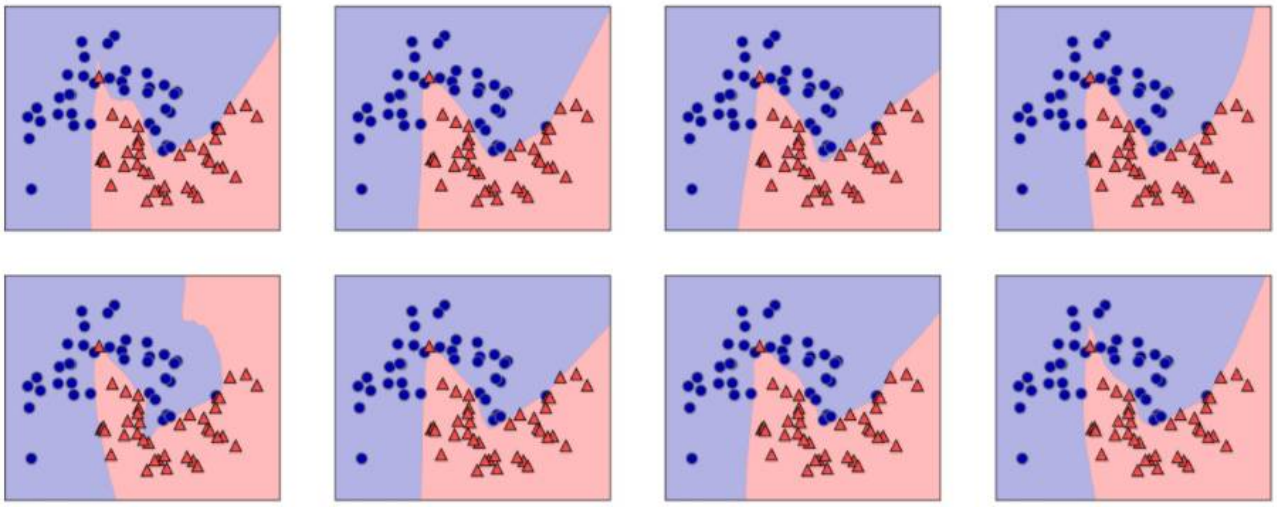
Save the program above (with 2 nodes in the hidden layer) under the name **29_MLP_XOR_2hidden.py** - you will need it for the homework!

---------- End of material not in text ----------

---------- Returning to text, on page 115 ----------

The difference in solutions can also be observed with Scikit-learn's MLPClassifier, by using different random seeds:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                        hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```



Note that the boundary is very similar in the dense areas, but it can be very different in sparse areas.

Applying MLPClassifier on a larger dataset

Let us first find the maxima and their positions for each of the 30 features. Note the use of the function *argmax( )*:

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))
print("Cancer maxima indices:\n{}".format(cancer.data.argmax(axis=0)))
```

```
Cancer data per-feature maxima:
[  2.81100000e+01   3.92800000e+01   1.88500000e+02   2.50100000e+03
   1.63400000e-01   3.45400000e-01   4.26800000e-01   2.01200000e-01
   3.04000000e-01   9.74400000e-02   2.87300000e+00   4.88500000e+00
   2.19800000e+01   5.42200000e+02   3.11300000e-02   1.35400000e-01
   3.96000000e-01   5.27900000e-02   7.89500000e-02   2.98400000e-02
   3.60400000e+01   4.95400000e+01   2.51200000e+02   4.25400000e+03
   2.22600000e-01   1.05800000e+00   1.25200000e+00   2.91000000e-01
   6.63800000e-01   2.07500000e-01]
Cancer maxima indices:
[212 239 212 461 504  78 122 122  25   3 212 192 212 461 213 190 152 152
  78 152 461 259 461 461 203   9  68 108   3   9]
```

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Accuracy on training set: {:.2f}".format(mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test, y_test)))
```

```
Accuracy on training set: 0.91
Accuracy on test set: 0.88
```

Now let us scale the data.

► The text does it manually, but, since we already covered the scalers in Ch.3, use the *StandardScaler*.

⇩

```
from sklearn.preprocessing import StandardScaler
sca = StandardScaler()
sca.fit(cancer.data)
cancer_scaled = sca.transform(cancer.data)
```

► Show that, in the scaled dataset, the maxima occur at the same indices!

► Repeat the classification, using the scaled data.

⇩

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer_scaled, cancer.target, random_state=0)
mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Accuracy on training set: {:.4f}".format(mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.4f}".format(mlp.score(X_test, y_test)))
```

```
Accuracy on training set: 0.9930
Accuracy on test set: 0.9650
```

```
C:\ProgramData\Anaconda2\lib\site-packages\sklearn\neural_network
\multilayer_perceptron.py:564: ConvergenceWarning: Stochastic Optimizer: Maximum
iterations (200) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

Increasing the nr. of iterations does not increase the testing set accuracy, however:

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train, y_train)
```

```
Accuracy on training set: 0.9930
Accuracy on test set: 0.9650
```

Next refinement:

```
'''Increasing alpha (from default 10^-4) to 1 decreases the complexity
and the overfitting of the classifier'''
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train, y_train)

print("Accuracy on training set: {:.4f}".format(
    mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.4f}".format(mlp.score(X_test, y_test)))
```

■

Solutions:

► Show that, in the scaled dataset, the maxima occur at the same indices!

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))
print("Cancer maxima indices:\n{}".format(cancer.data.argmax(axis=0)))

from sklearn.preprocessing import StandardScaler
sca = StandardScaler()
sca.fit(cancer.data)
cancer_scaled = sca.transform(cancer.data)

print("Cancer data per-feature maxima:\n{}".format(cancer_scaled.max(axis=0)))
print("Cancer maxima indices:\n{}".format(cancer_scaled.argmax(axis=0)))
```

```
Cancer data per-feature maxima:
[  2.81100000e+01   3.92800000e+01   1.88500000e+02   2.50100000e+03
   1.63400000e-01   3.45400000e-01   4.26800000e-01   2.01200000e-01
   3.04000000e-01   9.74400000e-02   2.87300000e+00   4.88500000e+00
   2.19800000e+01   5.42200000e+02   3.11300000e-02   1.35400000e-01
   3.96000000e-01   5.27900000e-02   7.89500000e-02   2.98400000e-02
   3.60400000e+01   4.95400000e+01   2.51200000e+02   4.25400000e+03
   2.22600000e-01   1.05800000e+00   1.25200000e+00   2.91000000e-01
   6.63800000e-01   2.07500000e-01]
Cancer maxima indices:
[212 239 212 461 504  78 122 122  25   3 212 192 212 461 213 190 152 152
  78 152 461 259 461 461 203   9  68 108   3   9]
Cancer data per-feature maxima:
[  3.97128765   4.65188898   3.97612984   5.25052883   4.77091122
   4.56842498   4.24358882   3.92792966   4.48475086   4.91091929
   8.90690934   6.65527935   9.46198577  11.04184226   8.02999927
   6.14348219  12.0726804    6.64960079   7.07191706   9.85159257
   4.09418939   3.88590505   4.28733746   5.9301724    3.95537411
   5.11287727   4.7006688    2.68587702   6.04604135   6.84685604]
Cancer maxima indices:
[212 239 212 461 504  78 122 122  25   3 212 192 212 461 213 190 152 152
  78 152 461 259 461 461 203   9  68 108   3   9]
```

or directly this:

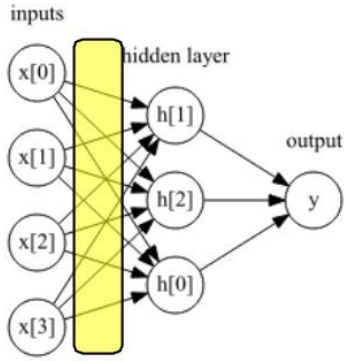
```
print cancer.data.argmax(axis=0) - cancer_scaled.argmax(axis=0)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

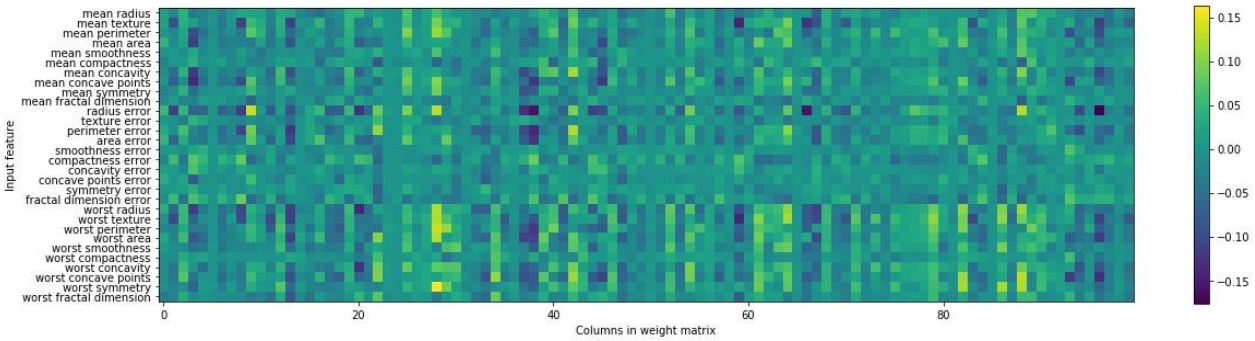
■

How to make sense of an MLP model: What has the model learned, exactly?

Let us visualize the weights connecting the inputs to the hidden layer:




```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()
```



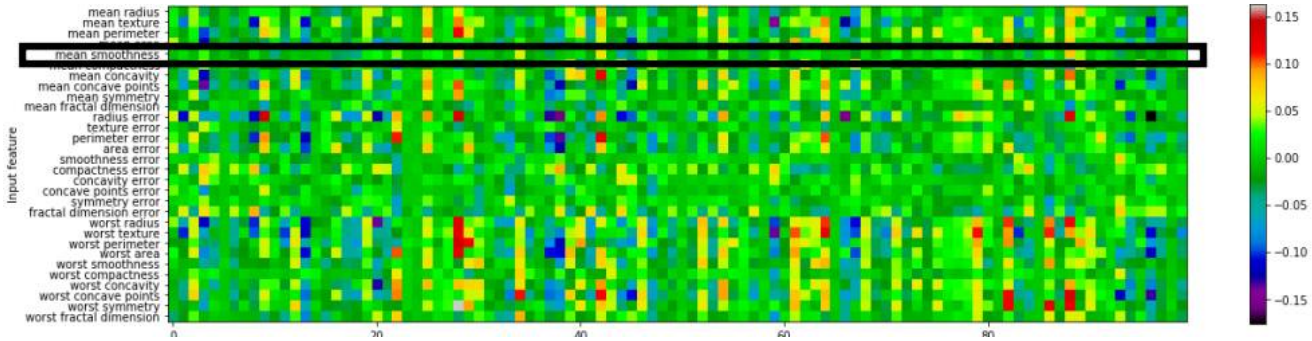
► Experiment with other color maps - see

https://matplotlib.org/examples/color/colormaps_reference.html

⇩

My favorite:

```
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='nipy_spectral')
```

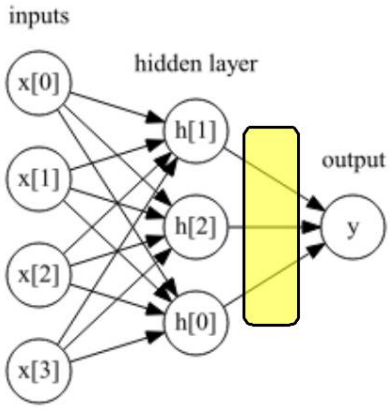


One way to interpret the weights is to find features whose weights are not very large (in absolute value), like *mean smoothness* shown above, and conclude that they do not contribute much to the classification. The conclusion is not so straightforward, however, because we cannot really consider particular weights in isolation: their effect also depends on:

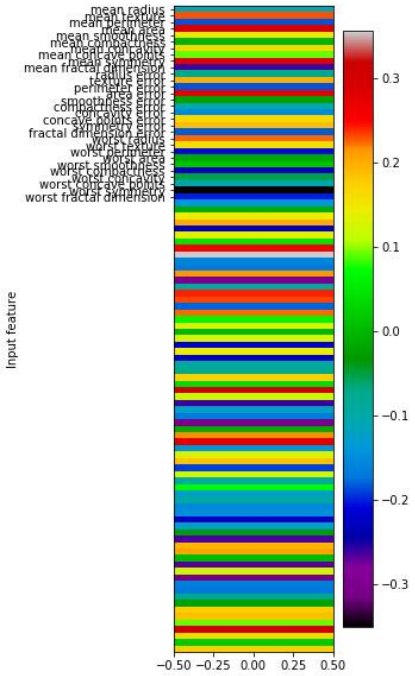
- All the other weights in the same layer
- The corresponsing weights in adjacent layers.

One way to "prove" that some features have indeed little relevance is to leave them out of the dataset, repeat the classification, and then compare the results. This is not the end of the story, however. It may be possible that such a feature is simply not scaled appropriately - if we could find the "correct" scaling (or some other transformation), all of a sudden we would have a better classification!

► Visualize in a similar manner the weights connecting the hidden layer to the outputs:



```
plt.figure(figsize=(3, 10))
plt.imshow(mlp.coefs_[1], interpolation='none', cmap='nipy_spectral',
           aspect='auto')
plt.yticks(range(30), cancer.feature_names)
#plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()
```



Strengths, weaknesses, complexity

Strengths

- ANNs can handle large amounts of data, and build complex models in a systematic way
- Their computations can be easily parallelized, to take advantage of multiple CPUs/GPUs

Weaknesses

- ANNs need large numbers of data points to make good predictions (e.g. Google's "Cat" network was trained with 10 million images). If we have moderate or small numbers of data points, Decision Trees with bagging (Random Forests) or boosting are likely a better choice.
- ANNs need to be parallelized, because otherwise the training times would be too large (e.g. Google's "Cat" network had 1 Billion parameters (≈ weights); it was trained in 2012 for 3 days on a cluster with 1000 machines/16,000 CPUs)
- Sensitive to scaling, they perform best on homogeneous data.
- Need careful tuning of parameters for good performance.

□ Complexity: How many weights are there in an ANN with **i** input nodes, **h1** nodes in the first hidden layer, **h2** nodes in the second hidden layer, and **o** output nodes?

Homework #5 was assigned (last homework for this class!)

■