[. . .]

In **The (Un)reliability of saliency methods** you propose input invariance as a prerequisite for reliable attribution and demonstrate that under a constant shift of the input, that multiple saliency methods *attribute incorrectly* by providing a counter example. Your reasoning, however, does not take into account the semantics of the methods themselves and this, in my opinion, makes the requirements of the methods to be invariant to the inputs an undesired general property for reliable saliency methods.

I will first try to follow your experimental protocol and then provide a counter example to your counter example using LRP, showing that the method is actually attributing relevance correctly, despite the constant input shift. There will be a short **TL;DR** in the end, summarizing my point.

Following the descriptions in the paper a MLP classifier on MNIST is trained with two hidden layers of 1024 neurons each, over 10 epochs using a SGD optimizer. Let's first setup some model training and then transfer the learned parameters into some LRP-capable code.

In [1]:
```
#import some required modules and define the training function
import os
import copy
import numpy as np
import matplotlib.pyplot as plt


#the MLP is trained using the MLPClassifier class provided by sklearn
from sklearn.neural_network import MLPClassifier

#LRP is performed via an adapted implementation of the python implementa
tion of the LRP Toolbox
import modules
import data_io
import model_io
import render

#globally choose a color map
plt.rcParams['image.cmap'] = 'seismic'

#define a function for training a model given training data and labels.
#after training, we transfer the learned model parameters into the layer
classes of the lrp toolbox.
def train_model(Xtrain, Ytrain):
    """ train a fixed architecture MLP given some training data and labe
ls """
    #build a cookie-cutter MLP from sklearn.
    #n-layer-network means n-1 hidden layers, 1 output layer. never sure
about that.
    net = MLPClassifier( hidden_layer_sizes=(1024, 1024),
                         activation='relu',
                         solver='sgd',
                         batch_size=20,
                         warm_start=True,
                         verbose=True,
                         max_iter=10) #max_iter with sgd as solver trains
for max_iter epochs

    net.fit(X,Y)
    #build and save lrp toolbox mlp to continue with

    layers = []
    l = modules.Linear(784, 1024)
    l.W = net.coefs_[0]
    l.B = net.intercepts_[0]
    layers.append(l)
    layers.append(modules.Rect())

    l = modules.Linear(1024, 1024)
    l.W = net.coefs_[1]
    l.B = net.intercepts_[1]
    layers.append(l)
    layers.append(modules.Rect())

    l = modules.Linear(1024, 10)
    l.W = net.coefs_[2]
    l.B = net.intercepts_[2]
    layers.append(l)
    layers.append(modules.Logistic()) #this is the default output activa
tion fxn of sklearn's MLPClassifier

    net = modules.Sequential(layers)
    #sanity-check model outputs and continue.
    np.testing.assert_array_equal(net.predict_proba(Xtest), net1.forward
(Xtest), err_msg='Results deviate')
    return net
```

Now load the MNIST data and train and evaluate the model.

In [2]:
```
#load mnist digits and labels for training.
X = data_io.read('../data/MNIST/train_images.npy').astype(np.float)
# N x D
Y = data_io.read('../data/MNIST/train_labels.npy')
# N x 1
Xtest = data_io.read('../data/MNIST/test_images.npy').astype(np.float)
# M x D
Ytest = data_io.read('../data/MNIST/test_labels.npy')
# M x 1


#normalize pixel values to [0 1] and vectorize labels to one-hot indicat
ors
X /= 255.
Xtest /= 255.

I = Y[:,0].astype(int)
Y = np.zeros([Y.shape[0], 10])
Y[np.arange(Y.shape[0]), I] = 1

I = Ytest[:,0].astype(int)
Ytest = np.zeros([Ytest.shape[0], 10])
Ytest[np.arange(Ytest.shape[0]), I] = 1


#load (if available) or train a network
modelpath = './net1.nn'
if not os.path.isfile(modelpath):
    net1 = train_model(X,Y)
    #everything is all right. save the model.
    model_io.write(net1, modelpath)
else:
    #load a precomputed model.
    net1 = model_io.read(modelpath)


#compute and print model performance
Ypred = net1.forward(Xtest)
acc1 = np.mean(np.argmax(Ypred, axis=1) == np.argmax(Ytest, axis=1))
print 'mean accuracy for net1 is at {:.2f}%'.format(acc1*100)
```

```
loading np-formatted data from ../data/MNIST/train_images.npy
loading np-formatted data from ../data/MNIST/train_labels.npy
loading np-formatted data from ../data/MNIST/test_images.npy
loading np-formatted data from ../data/MNIST/test_labels.npy
loading pickled model from ./net1.nn
mean accuracy for net1 is at 97.94%
```

Now define the constant shift to apply to the input and create the *Net2* which operates on the shifted inputs.

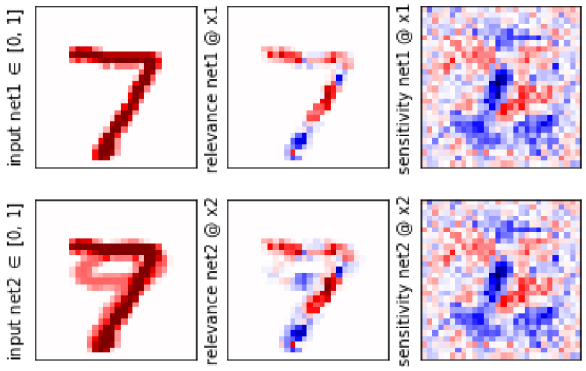
In [5]:
```
#define the constant shift to use
#m = -np.ones([784]) #constant shift of -1 per pixel
m = -Xtest[9] * -0.3 #alternatively shift by some digit pattern within [
-.3, .3]

#print X[:,0].sum(), X[:,0].max(), X[:,0].min(), X[:,0].size
#net1.train(X,Y)

#build a shifted test data set, moving pixel ranges to [-1 0]
Xtest2 = Xtest + m[None,...]

#build a net2, with a bias compensating for the constant shift per pixel
.
net2 = copy.deepcopy(net1)
net2.modules[0].B -= np.dot(m, net2.modules[0].W)

Ypred2 = net2.forward(Xtest2)
acc2 = np.mean(np.argmax(Ypred2, axis=1) == np.argmax(Ytest, axis=1))
print 'mean accuracy for net2 is at {:.2f}%'.format(acc2*100)


#build a net2b, with a bias compensating for the constant shift per pixe
l.
net2b = copy.deepcopy(net1)

Ypred2b = net2b.forward(Xtest2)
acc2b = np.mean(np.argmax(Ypred2b, axis=1) == np.argmax(Ytest, axis=1))
print 'mean accuracy for net2b is at {:.2f}%'.format(acc2b*100)



#build a net1b, with a bias compensating for the constant shift per pixe
l.
net1b = copy.deepcopy(net1)
net1b.modules[0].B = 0

Ypred1b = net1b.forward(Xtest)
acc1b = np.mean(np.argmax(Ypred1b, axis=1) == np.argmax(Ytest, axis=1))
print 'mean accuracy for net1b is at {:.2f}%'.format(acc1b*100)

#assert the models behave the same in terms of predictions and hidden un
it activations of all layers - up to numerical tolerance
np.testing.assert_array_equal(acc1, acc2, err_msg='Performance of origin
al net and net with constant shift+bias compensation differ!')
for i in xrange(len(net2.modules)):
    np.testing.assert_allclose(net1.modules[i].Y, net2.modules[i].Y, err
_msg='Layer activations differ at layer {}'.format(i))
    print 'layer {} output activations reasonably equal'.format(i)
```

```
mean accuracy for net2 is at 97.94%
mean accuracy for net2b is at 94.33%
mean accuracy for net1b is at 97.95%
layer 0 output activations reasonably equal
layer 1 output activations reasonably equal
layer 2 output activations reasonably equal
layer 3 output activations reasonably equal
layer 4 output activations reasonably equal
layer 5 output activations reasonably equal
```

If everything works so far, then *Net2* behaves as expected and mimics *Net1* in its prediction on the shifted inputs. If we now compute relevance maps for both models given corresponding input data, the results will be different.

In [6]:
```
#choose some test sample index and
II = 0
#and get the inputs for net1 and net2
x1 = Xtest[II:II+1,...]
x2 = Xtest2[II:II+1,...]

y1 = net1.forward(x1)
y2 = net2.forward(x2)

print 'net1 predicts class {} : {}'.format(np.argmax(y1), y1.tolist())
print 'net2 predicts class {} : {}'.format(np.argmax(y2), y2.tolist())
np.testing.assert_allclose(y1, y2)

#if we manage to arrive here, both models should predict (almost) identi
cal (up to numerics).
#let's now compute relevance maps for the input pixels of both inputs an
d neural networks.
#start by masking the non-dominant outputs and only use the model output
for the predicted class
#as value for initiating lrp
c = np.argmax(y1)
mask = np.zeros_like(y1)
mask[0,c] = 1

#note that we choose the output of the last linear layer (prior to the l
ogistic output nonlinearity,
#which transforms the output of that layer into probabilities) as a star
ting point.
#this prevents sign flipped relevance maps in case the logistic layer sh
ould turn a negative output into
#a positive one and nicely maintains relative class 'presence'. Don't th
ink about it in this context.
r1 = net1.lrp(net1.modules[-2].Y * mask)
r2 = net2.lrp(net2.modules[-2].Y * mask)

#also compute the sensitivity map for both models and inputs for the pre
dicted class
s1 = net1.backward(y1)
s2 = net2.backward(y2)

#compute a common maximal absolute value for color space alignment for t
he relevace and sensitivity maps
v = max(np.abs(r1).max(), np.abs(r2).max())
s = max(np.abs(s1).max(), np.abs(s2).max())

#show the inputs, relevance and sensitivity maps as images in a 2 by 3 s
ubplot
plt.subplot(231)
plt.imshow(x1.reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('input net1 $\in$ [{}, {}]'.format(int(x1.min()), int(x1.max(
)))

plt.subplot(234)
plt.imshow(x2.reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('input net2 $\in$ [{}, {}]'.format(int(x2.min()), int(x2.max(
)))

plt.subplot(232)
plt.imshow(r1.reshape(28,28), vmin=-v , vmax=v)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('relevance net1 @ x1')

plt.subplot(235)
plt.imshow(r2.reshape(28,28), vmin=-v , vmax=v)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('relevance net2 @ x2')
```

```
net1 predicts class 7 : [[2.0678928411919765e-05, 8.237225215090837e-06,
0.0001202011260885909, 0.000492287308001941, 3.305704672718165e-08, 2.180
6811318802892e-05, 2.74579294697595e-10, 0.9990849938042885, 2.0187868035
287114e-05, 2.27304829046133e-05]]
net2 predicts class 7 : [[2.0678928411919803e-05, 8.237225215090822e-06,
0.0001202011260885911, 0.0004922873080019419, 3.305704672718165e-08, 2.18
06811318802933e-05, 2.74579294697595e-10, 0.9990849938042885, 2.018786803
5287114e-05, 2.273048290461326e-05]]
```



We can see, as you have written, that gradients (sensitivity maps) are unaffected by a constant shift in input space. The relevance maps however, are affected, despite each layer's activations remaining the same as earlier assertions have shown.

This is due to the change in handling the input. While the relevance values in inner layers of the network are equal (again, up to numerical precision), the relevance map for the input layer differs (which can also be seen in the figure above).

A quick sanity check should confirm this:

In [7]:
```
#assert the models behave the same in terms of relevance attribution in
all layers. Layer 0 is expected to fail.
for i in range(len(net2.modules))[::-1]:
    np.testing.assert_allclose(net1.modules[i].R, net2.modules[i].R, err
_msg='Input neuron relevances differ differ at layer {}'.format(i))
    print 'layer {} input neuron relevances are reasonably equal'.format
(i)
```

```
layer 5 input neuron relevances are reasonably equal
layer 4 input neuron relevances are reasonably equal
layer 3 input neuron relevances are reasonably equal
layer 2 input neuron relevances are reasonably equal
layer 1 input neuron relevances are reasonably equal

---------------------------------------------------------------------------
--
AssertionError                            Traceback (most recent call las
t)
<ipython-input-7-c03b4d7b775d> in <module>()
      1 #assert the models behave the same in terms of relevance attribut
ion in all layers. Layer 0 is expected to fail.
      2 for i in range(len(net2.modules))[::-1]:
----> 3     np.testing.assert_allclose(net1.modules[i].R, net2.modules[i]
.R, err_msg='Input neuron relevances differ differ at layer {}'.format(i)
)
      4     print 'layer {} input neuron relevances are reasonably equal'
.format(i)

/home/lapuschkin/.local/lib/python2.7/site-packages/numpy/testing/utils.p
yc in assert_allclose(actual, desired, rtol, atol, equal_nan, err_msg, ve
rbose)
   1393     header = 'Not equal to tolerance rtol=%g, atol=%g' % (rtol, a
tol)
   1394     assert_array_compare(compare, actual, desired, err_msg=str(er
r_msg),
-> 1395                          verbose=verbose, header=header, equal_na
n=equal_nan)
   1396
   1397

/home/lapuschkin/.local/lib/python2.7/site-packages/numpy/testing/utils.p
yc in assert_array_compare(comparison, x, y, err_msg, verbose, header, pr
ecision, equal_nan, equal_inf)
    776                                 names=('x', 'y'), precision=preci
sion)
    777             if not cond:
--> 778                 raise AssertionError(msg)
    779     except ValueError:
    780         import traceback

AssertionError:
Not equal to tolerance rtol=1e-07, atol=0
Input neuron relevances differ differ at layer 0
(mismatch 22.4489795918%)
 x: array([[  0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+0
0,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,..
.
 y: array([[  0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+0
0,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,..
.
```

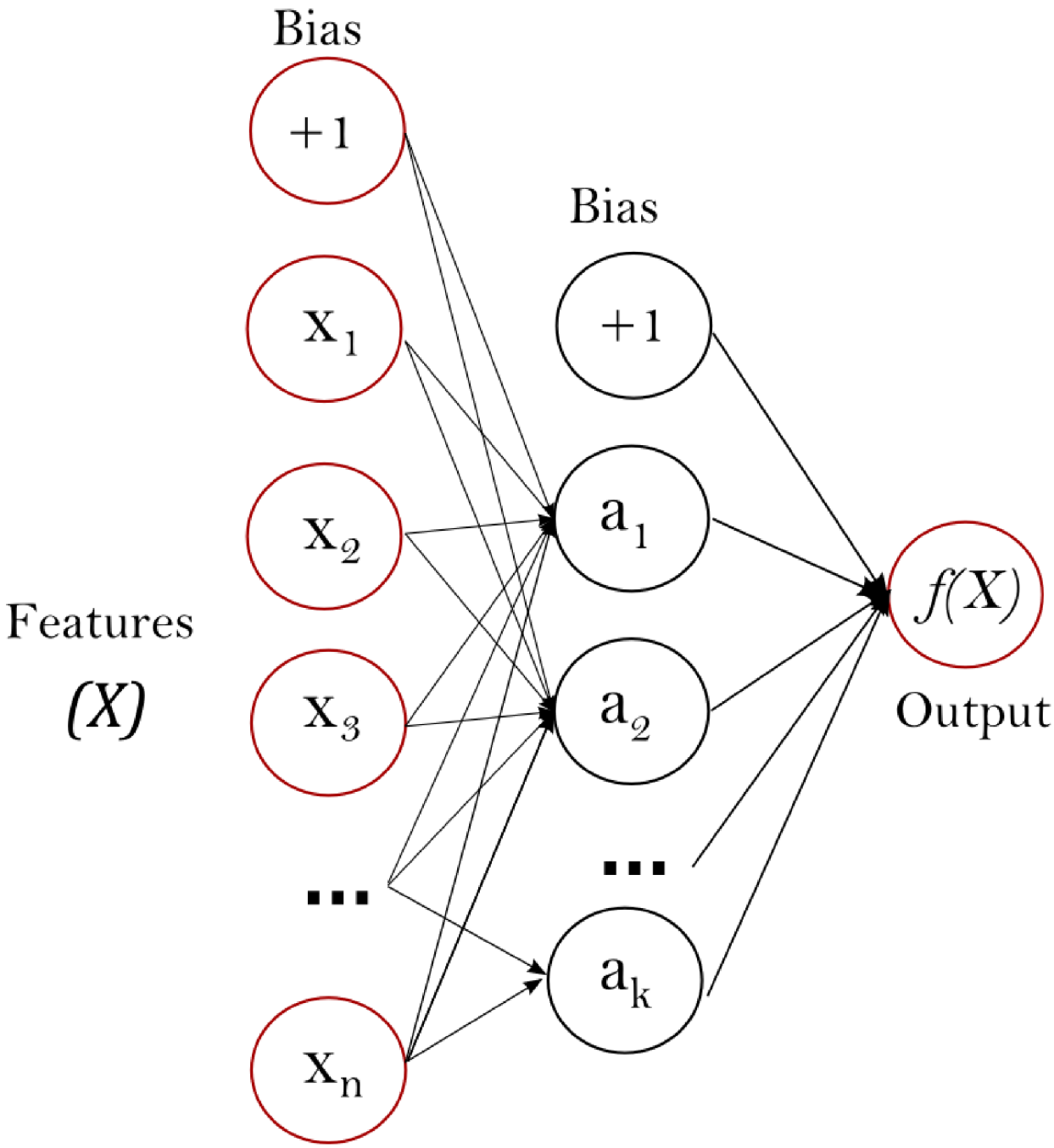
Now it is important to consider what the respective method does, and what semantics the computed saliency maps hold. Sensitivity Analysis, for example, identifies the inputs the model is most (or least, depending what you are looking for) sensible to. We can easily pick input and output indices most and least sensitive for a given class and measure the impact:

In [8]:
```
#select the inputs we talked about
least_sensitive_input = np.argmin(np.abs(s1))
most_sensitive_input = np.argmax(np.abs(s1))

#clone the network for the sake of this demonstration
clonet = copy.deepcopy(net1)

#now manipulate the input wrt to the sensitivity map and measure the eff
ect on the prediction of class c
least_impact_delta = np.zeros_like(x1)
least_impact_delta[0,least_sensitive_input] = +1

most_impact_delta = np.zeros_like(x1)
most_impact_delta[0,most_sensitive_input] = +1

#compare model outputs prior to the logistic output layer, for better un
derstanding
#the linear class ranking created by the output layer.
y_orig = clonet.forward(x1)
y_orig = clonet.modules[-2].Y
y_most = clonet.forward(x1 + most_impact_delta)
y_most = clonet.modules[-2].Y
y_least = clonet.forward(x1 + least_impact_delta)
y_least = clonet.modules[-2].Y

print 'effect on class {} prediction (pre-logistic) when altering its mo
st sensitive input : {:.5f}'.format(c,abs(y_most[0,c] - y_orig[0,c]))
print 'effect on class {} prediction (pre-logistic) when altering its le
ast sensitive input : {:.5f}'.format(c,abs(y_least[0,c] - y_orig[0,c]))
```

```
effect on class 7 prediction (pre-logistic) when altering its most sensit
ive input : 1.37330
effect on class 7 prediction (pre-logistic) when altering its least sensi
tive input : 0.02140
```

LRP however, as a saliency method designed to *tell how it is* does indeed fail the input invariance criterion: LRP measures how each input component/neuron/dimension contributes to the prediction *or* where the numbers come from which in the end sum up to $f(x)$, and therefore *should* fail the constant shift test designed and evaluated in the paper. However, this does not mean that LRP *incorrectly attributes* its relevance scores.

To make my point clear I want to make a quick excursion on how the additive bias in neural network layers work, and what effect this has on LRP and the relevance conservation criterion. On [http://scikit-learn.org/stable/modules/neural_networks_supervised.html (http://scikit-learn.org/stable/modules/neural_networks_supervised.html)] you can find an image showing the schematic of a MLP classifier, which does support my point quite well. The image is linked below:



Each linear layer of a NN usually (there are always exceptions) has an additive bias which is tuned during training time. This bias can be interpreted as an always-on-neuron, which has its own set of weights connecting it to the layer's output neurons.

In practice, the bias is not concatenated to the input vector as a constant neuron -- i.e. such that a forward pass through the layer only is a dot product between the extended inputs and one learned weight matrix -- but treated as a separate additive element. Nonetheless, it is subject to the same rules as the input-dependant neurons within each layer: it fires --

In [9]:
```
print 'Total relevance per layer for net1'
for i in range(len(net1.modules))[::-1]:
    rx, rb = np.sum(net1.modules[i].R) , np.sum(net1.modules[i].Rb)
    print 'layer {} ({}) Total: {} , R@Input: {} . R@Bias: {}'.format(i,
net1.modules[i].__class__.__name__, rx+rb, rx, rb)

    if i < len(net1.modules)-1:
        #also assert that the total relevance within one layer i equals
the input layer relevance of the succeeding layer i+1
        #again, this comparison is also done with some loose tolerances.
the relevances will usually differ very little (i.e. relative gap of 1e-
10)
        np.testing.assert_allclose(rx+rb, np.sum(net1.modules[i+1].R), e
rr_msg='Total relevance at layer {} differs from input neuron relevance
at layer {}'.format(i,i+1))


print ''
print 'Total relevance per layer for net2'
for i in range(len(net2.modules))[::-1]:
    rx, rb = np.sum(net2.modules[i].R) , np.sum(net2.modules[i].Rb)
    print 'layer {} ({}) Total: {} , R@Input: {} . R@Bias: {}'.format(i,
net2.modules[i].__class__.__name__, rx+rb, rx, rb)

    if i < len(net2.modules)-1:
        #also assert that the total relevance within one layer i equals
the input layer relevance of the succeeding layer i+1
        np.testing.assert_allclose(rx+rb, np.sum(net1.modules[i+1].R), e
rr_msg='Total relevance at layer {} differs from input neuron relevance
at layer {}'.format(i,i+1))
```

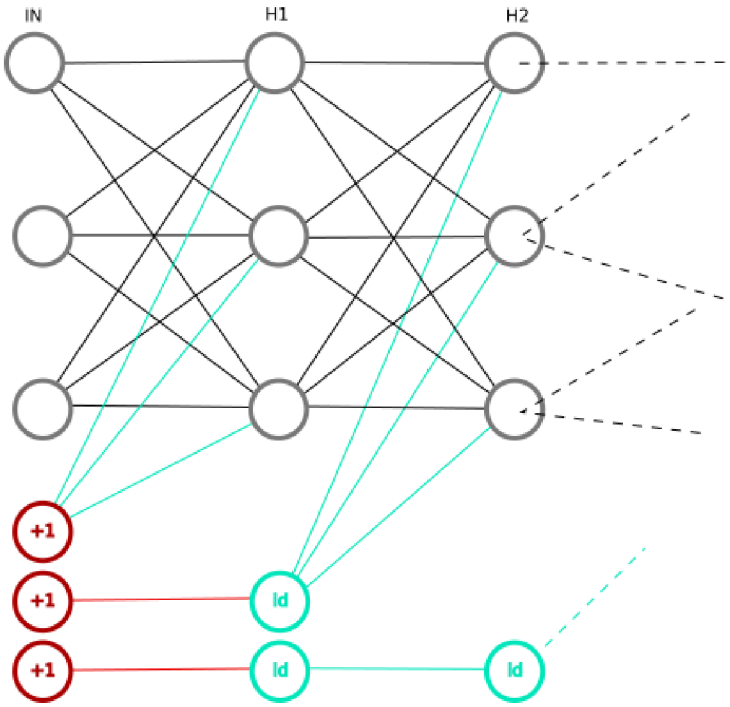
```
Total relevance per layer for net1
layer 5 (Logistic) Total: 6.99566429637 , R@Input: 6.99566429637 . R@Bias
: 0
layer 4 (Linear) Total: 6.99566429637 , R@Input: 7.14291000621 . R@Bias:
-0.147245709837
layer 3 (Rect) Total: 7.14291000621 , R@Input: 7.14291000621 . R@Bias: 0
layer 2 (Linear) Total: 7.14291000621 , R@Input: 7.24513238747 . R@Bias:
-0.102222381259
layer 1 (Rect) Total: 7.24513238747 , R@Input: 7.24513238747 . R@Bias: 0
layer 0 (Linear) Total: 7.24513238747 , R@Input: 7.04266123951 . R@Bias:
0.202471147957

Total relevance per layer for net2
layer 5 (Logistic) Total: 6.99566429637 , R@Input: 6.99566429637 . R@Bias
: 0
layer 4 (Linear) Total: 6.99566429637 , R@Input: 7.14291000621 . R@Bias:
-0.147245709837
layer 3 (Rect) Total: 7.14291000621 , R@Input: 7.14291000621 . R@Bias: 0
layer 2 (Linear) Total: 7.14291000621 , R@Input: 7.24513238747 . R@Bias:
-0.102222381259
layer 1 (Rect) Total: 7.24513238747 , R@Input: 7.24513238747 . R@Bias: 0
layer 0 (Linear) Total: 7.24513238747 , R@Input: 7.23216433296 . R@Bias:
0.0129680545062
```

If above code block runs through without assertion errors we should be able to see what amount of relevance is absorbed by the bias node (if there is any) in each layer, and how much is further transferred towards the input. Also, no errors also mean that the total amount of relevance distributed towards the input neurons (the input-dependent neurons as well as the always-on bias) equals the amount of relevance passed as input from the immediately neighboring layer closer to the output. We can also see, that the constant shift of the input and the adapted bias term compensating for the shift greatly affect the relevance distribution Net2. However, the grand total of relevance distributed within that layer stays the same.

One might argue that this relevance absorbtion violates the *relevance conservation criterion*, which is an integral part of LRP, since the total sum of relevance from layer to layer changes. There are three options arguing against this being a violation.

ONE: We accept relevance is absorbed wherever source neurons (bias or input neurons) are activated and we are okay with that

TWO: We can create proxy neurons for each bias neuron, which are placed at all layers preceding and including the bias neuron's actual host layer and do nothing but relay the constant activation forward. The constantly firing source neuron is now placed at the input layer. This provides a workaround to above described violation of the relevance conservation criterion by representing the absorbed relevance quantities at every layer in the bias proxies labelled as 'id'. teal connections are the *bias weights* and red connections have weight 1 (and implement a identity function). This is the option I prefer and which is, in my opinion, the best way to deal with the bias terms, despite option ONE also being without error.



THREE: The relevance quantity absorbed by the bias term can be redistributed across the input variant neurons, such that relevance is preserved across all layers (but under sligtly incorrect attribution)

Let us below quickly simulate option TWO by also accumulating and adding the sum over all relevance quantities absorbed by bias neurons in succeeding layers when computing the sum over relevances for each layer. An assertion is then used to ensure the relevance conservation criterion holds:

In [10]:
```
print 'Total relevance per layer for net1 with bias proxy neurons'
for i in range(len(net1.modules))[::-1]:
    #compute the sum over input neuron relevances of layer i
    #and add the relevance quantities absorbed by bias nodes at layer j>
=i to simulate the bias proxies.
    rtotal = np.sum(net1.modules[i].R) + np.sum([np.sum(net1.modules[j].
Rb) for j in range(i, len(net1.modules))])
    print 'layer {} ({}) Total: {}'.format(i, net1.modules[i].__class__.
__name__, rtotal)

    np.testing.assert_allclose(rtotal, np.sum(net1.modules[-1].R) + np.s
um(net1.modules[-1].Rb), err_msg='Relevance Conservation Criterion viola
ted at layer {} ({})'.format(i, net1.modules[i].__class__.__name__))


print ''
print 'Total relevance per layer for net2 with bias proxy neurons'
for i in range(len(net2.modules))[::-1]:
    #compute the sum over input neuron relevances of layer i
    #and add the relevance quantities absorbed by bias nodes at layer j>
=i to simulate the bias proxies.
    rtotal = np.sum(net2.modules[i].R) + np.sum([np.sum(net2.modules[j].
Rb) for j in range(i, len(net2.modules))])
    print 'layer {} ({}) Total: {}'.format(i, net2.modules[i].__class__.
__name__, rtotal)

    np.testing.assert_allclose(rtotal, np.sum(net2.modules[-1].R) + np.s
um(net2.modules[-1].Rb), err_msg='Relevance Conservation Criterion viola
ted at layer {} ({})'.format(i, net2.modules[i].__class__.__name__))
```

```
Total relevance per layer for net1 with bias proxy neurons
layer 5 (Logistic) Total: 6.99566429637
layer 4 (Linear) Total: 6.99566429637
layer 3 (Rect) Total: 6.99566429637
layer 2 (Linear) Total: 6.99566429637
layer 1 (Rect) Total: 6.99566429637
layer 0 (Linear) Total: 6.99566429637

Total relevance per layer for net2 with bias proxy neurons
layer 5 (Logistic) Total: 6.99566429637
layer 4 (Linear) Total: 6.99566429637
layer 3 (Rect) Total: 6.99566429637
layer 2 (Linear) Total: 6.99566429637
layer 1 (Rect) Total: 6.99566429637
layer 0 (Linear) Total: 6.99566429637
```

Let's get back to the constant shift applied to the input samples and the adaptive bias term compensating for the change in input layer output neuron activation. As we can see above, the bias terms of all the linear layers within our models Net1 and Net2 absorb some quantity of positive or netagitive relevance, corresponding to the *share* the bias contributes to the layer's output activations. Since the composition of sources contributing to the output activation of the input layer of Net2 greatly shifts compared to Net2, this also reflects in the relevance maps on pixel level.

Since the adapted bias b_2 of Net2 depends on the originally learned bias b_1 in Net1, the applied shift m (per pixel) and the learned weights w shared by both models, we can model the bias adaption explicitly as an always on bias *vector* at the input level due to the law of commutativity.

The first layer activations for Net1 are computed as:

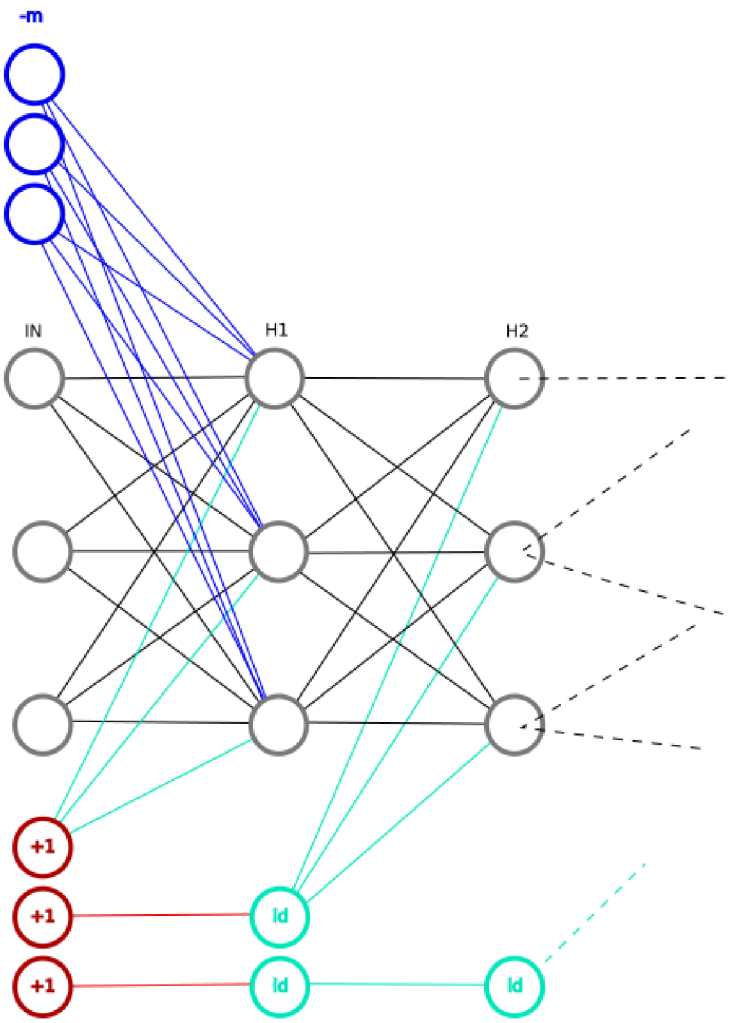
$$z = w^T x_1 + b_1$$

Similarly, the first layer activations for Net2 , when expressed wrt to the shift m applied to the data and the compensation via bias b_2, can be written as:

$$z = w^T x_2 + b_2 = (w^T x_1 + w^T m) + (b_1 - w^T m) = \mathbf{w^T x_2 + b_1 - w^T m}$$

Looking at the last term in the equation for the input layer activations of Net2, we can see that z is the sum of the forward propagated constantly shifted network input, the learned bias and a term compensating the constant shift, which is part of the b_2 in Net2. Net2 still receives the shifted inputs, just the (compensating) bias is formulated differently.

We can explicitly model the compensator term $-w^T m$ as an always on MNIST image sized secondary and constant image-shaped bias input, which is then transformed by W. Adapting the previous example image, such a network could be constructed as follows, with the black weighted edges sharing parameters (i.e. using the same set of weights W) with the blue ones.



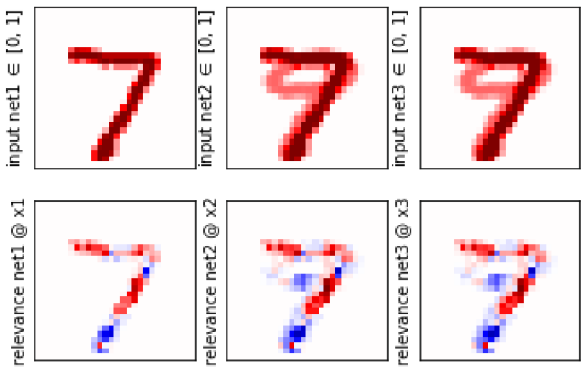
In [12]:
```
#construct a network behaving just like net1 and net2, but with a custom
input layer
net3 = copy.deepcopy(net1)
customlayer = modules.CustomLinear(net3.modules[0].m, net3.modules[0].n)
customlayer.W = net3.modules[0].W
customlayer.B = net3.modules[0].B
customlayer.M = m.reshape(1,m.size)
net3.modules[0] = customlayer

Ypred = net1.forward(Xtest) #run on the test data again on net1
acc1 = np.mean(np.argmax(Ypred, axis=1) == np.argmax(Ytest, axis=1))

Ypred3 = net3.forward(Xtest2)
acc3 = np.mean(np.argmax(Ypred3, axis=1) == np.argmax(Ytest, axis=1))
print 'mean accuracy for net2 is at {:.2f}%'.format(acc3*100)

#assert the models behave the same in terms of predictions and hidden un
it activations of all layers - up to numerical tolerance
np.testing.assert_array_equal(acc1, acc3, err_msg='Performance of origin
al net and net with custom layer differ!')
for i in xrange(len(net3.modules)):
    np.testing.assert_allclose(net1.modules[i].Y, net3.modules[i].Y, err
_msg='Layer activations differ at layer {}'.format(i))
    print 'layer {} output activations reasonably equal'.format(i)
```

```
mean accuracy for net2 is at 97.94%
layer 0 output activations reasonably equal
layer 1 output activations reasonably equal
layer 2 output activations reasonably equal
layer 3 output activations reasonably equal
layer 4 output activations reasonably equal
layer 5 output activations reasonably equal
```

In the forward pass, Net3 behaves as expected. In the relevance backward propagation pass, Net3 and Net2 should deliver the same relevance maps per layer and a difference between Net1 and Net3 should cause an assertion error at layer 0. Let's reuse and adapt the same relevance computation block as above.

In [14]:
```
#and get the inputs for net1 and net2
x1 = Xtest[II:II+1,...]
x2 = Xtest2[II:II+1,...]
x3 = Xtest2[II:II+1,...] #same as x2

y1 = net1.forward(x1)
y2 = net2.forward(x2)
y3 = net3.forward(x3)

print 'net1 predicts class {} : {}'.format(np.argmax(y1), y1.tolist())
print 'net2 predicts class {} : {}'.format(np.argmax(y2), y2.tolist())
print 'net3 predicts class {} : {}'.format(np.argmax(y3), y3.tolist())
np.testing.assert_allclose(y1, y2)
np.testing.assert_allclose(y2, y3)

#if we manage to arrive here, both models should predict (almost) identi
cal (up to numerics).
#let's now compute relevance maps for the input pixels of both inputs an
d neural networks.
#start by masking the non-dominant outputs and only use the model output
for the predicted class
#as value for initiating lrp
c = np.argmax(y1)
mask = np.zeros_like(y1)
mask[0,c] = 1

#note that we choose the output of the last linear layer (prior to the l
ogistic output nonlinearity,
#which transforms the output of that layer into probabilities) as a star
ting point.
#this prevents sign flipped relevance maps in case the logistic layer sh
ould turn a negative output into
#a positive one and nicely maintains relative class 'presence'. Don't th
ink about it in this context.
r1 = net1.lrp(net1.modules[-2].Y * mask)
r2 = net2.lrp(net2.modules[-2].Y * mask)
r3 = net3.lrp(net3.modules[-2].Y * mask)

#compute a common maximal absolute value for color space alignment for t
he relevace and sensitivity maps
v = max(np.abs(r1).max(), np.abs(r2).max(), np.abs(r3).max())


#show the inputs and relevance maps as images in a 3 by 2 subplot
plt.subplot(231)
plt.imshow(x1.reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('input net1 $\in$ [{}, {}]'.format(int(x1.min()), int(x1.max(
)))

plt.subplot(232)
plt.imshow(x2.reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('input net2 $\in$ [{}, {}]'.format(int(x2.min()), int(x2.max(
)))

plt.subplot(233)
plt.imshow(x3.reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('input net3 $\in$ [{}, {}]'.format(int(x3.min()), int(x3.max(
)))


plt.subplot(234)
plt.imshow(r1.reshape(28,28), vmin=-v , vmax=v)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('relevance net1 @ x1')
```

```
net1 predicts class 7 : [[2.0678928411919765e-05, 8.237225215090837e-06,
0.0001202011260885909, 0.000492287308001941, 3.305704672718165e-08, 2.180
6811318802892e-05, 2.74579294697595e-10, 0.9990849938042885, 2.0187868035
287114e-05, 2.27304829046133e-05]]
net2 predicts class 7 : [[2.0678928411919803e-05, 8.237225215090822e-06,
0.0001202011260885911, 0.0004922873080019419, 3.305704672718165e-08, 2.18
06811318802933e-05, 2.74579294697595e-10, 0.9990849938042885, 2.018786803
5287114e-05, 2.273048290461326e-05]]
net3 predicts class 7 : [[2.0678928411919765e-05, 8.237225215090822e-06,
0.00012020112608859132, 0.0004922873080019419, 3.305704672718165e-08, 2.1
806811318802933e-05, 2.74579294697595e-10, 0.9990849938042885, 2.01878680
35287114e-05, 2.273048290461326e-05]]
```



Numerically asserting the equality of input neuron relevances between all layers for Net2 and Net3 should pass, while failure is expected for comparing Net1 and Net3.

In [15]:
```
print 'comparing net2 and net3 layer input relevances'
for i in range(len(net2.modules))[::-1]:  #this should pass
    np.testing.assert_allclose(net2.modules[i].R, net3.modules[i].R, err
_msg='Input neuron relevances differ differ at layer {}'.format(i))
    print 'layer {} input neuron relevances are reasonably equal'.format
(i)

print ''
print 'comparing net1 and net3 layer input relevances'
for i in range(len(net1.modules))[::-1]:  #this should fail at layer 0
    np.testing.assert_allclose(net1.modules[i].R, net3.modules[i].R, err
_msg='Input neuron relevances differ differ at layer {}'.format(i))
    print 'layer {} input neuron relevances are reasonably equal'.format
(i)
```

```
comparing net2 and net3 layer input relevances
layer 5 input neuron relevances are reasonably equal
layer 4 input neuron relevances are reasonably equal
layer 3 input neuron relevances are reasonably equal
layer 2 input neuron relevances are reasonably equal
layer 1 input neuron relevances are reasonably equal
layer 0 input neuron relevances are reasonably equal

comparing net1 and net3 layer input relevances
layer 5 input neuron relevances are reasonably equal
layer 4 input neuron relevances are reasonably equal
layer 3 input neuron relevances are reasonably equal
layer 2 input neuron relevances are reasonably equal
layer 1 input neuron relevances are reasonably equal

---------------------------------------------------------------------------
--
AssertionError                            Traceback (most recent call las
t)
<ipython-input-15-188706e431f8> in <module>()
      7 print 'comparing net1 and net3 layer input relevances'
      8 for i in range(len(net1.modules))[::-1]:  #this should fail at la
yer 0
----> 9     np.testing.assert_allclose(net1.modules[i].R, net3.modules[i]
.R, err_msg='Input neuron relevances differ differ at layer {}'.format(i)
)
     10     print 'layer {} input neuron relevances are reasonably equal'
.format(i)

/home/lapuschkin/.local/lib/python2.7/site-packages/numpy/testing/utils.p
yc in assert_allclose(actual, desired, rtol, atol, equal_nan, err_msg, ve
rbose)
   1393     header = 'Not equal to tolerance rtol=%g, atol=%g' % (rtol, a
tol)
   1394     assert_array_compare(compare, actual, desired, err_msg=str(er
r_msg),
-> 1395                          verbose=verbose, header=header, equal_na
n=equal_nan)
   1396
   1397

/home/lapuschkin/.local/lib/python2.7/site-packages/numpy/testing/utils.p
yc in assert_array_compare(comparison, x, y, err_msg, verbose, header, pr
ecision, equal_nan, equal_inf)
    776                                 names=('x', 'y'), precision=preci
sion)
    777             if not cond:
--> 778                 raise AssertionError(msg)
    779     except ValueError:
    780         import traceback

AssertionError:
Not equal to tolerance rtol=1e-07, atol=0
Input neuron relevances differ differ at layer 0
(mismatch 22.4489795918%)
 x: array([[  0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+0
0,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,..
.
 y: array([[  0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+0
0,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,
          0.000000e+00,   0.000000e+00,   0.000000e+00,   0.000000e+00,..
.
```

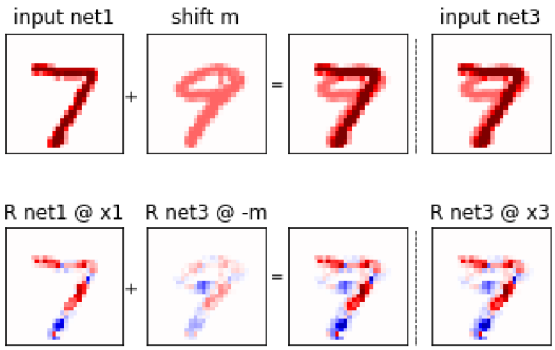
However, as the code block below will show, the relevance totals per layer remain the same.

In [16]:
```
print 'Total relevance per layer for net1 with bias proxy neurons'
for i in range(len(net1.modules))[::-1]:
    #compute the sum over input neuron relevances of layer i
    #and add the relevance quantities absorbed by bias nodes at layer j>
=i to simulate the bias proxies.
    rtotal = np.sum(net1.modules[i].R) + np.sum([np.sum(net1.modules[j].
Rb) for j in range(i, len(net1.modules))])
    print 'layer {} ({}) Total: {}'.format(i, net1.modules[i].__class__.
__name__, rtotal)

    np.testing.assert_allclose(rtotal, np.sum(net1.modules[-1].R) + np.s
um(net1.modules[-1].Rb), err_msg='Relevance Conservation Criterion viola
ted at layer {} ({})'.format(i, net1.modules[i].__class__.__name__))


print ''
print 'Total relevance per layer for net2 with bias proxy neurons'
for i in range(len(net2.modules))[::-1]:
    #compute the sum over input neuron relevances of layer i
    #and add the relevance quantities absorbed by bias nodes at layer j>
=i to simulate the bias proxies.
    rtotal = np.sum(net2.modules[i].R) + np.sum([np.sum(net2.modules[j].
Rb) for j in range(i, len(net2.modules))])
    print 'layer {} ({}) Total: {}'.format(i, net2.modules[i].__class__.
__name__, rtotal)

    np.testing.assert_allclose(rtotal, np.sum(net2.modules[-1].R) + np.s
um(net2.modules[-1].Rb), err_msg='Relevance Conservation Criterion viola
ted at layer {} ({})'.format(i, net2.modules[i].__class__.__name__))

print ''
print 'Total relevance per layer for net3 with bias proxy neurons and cu
stom input layer'
for i in range(len(net3.modules))[::-1]:
    #compute the sum over input neuron relevances of layer i
    #and add the relevance quantities absorbed by bias nodes at layer j>
=i to simulate the bias proxies.
    rtotal = np.sum(net3.modules[i].R) + np.sum([np.sum(net3.modules[j].
Rb) for j in range(i, len(net3.modules))])
    if i == 0:
        rtotal += np.sum(net3.modules[0].Rm) # handle custom layer relev
ance attribution

    print 'layer {} ({}) Total: {}'.format(i, net3.modules[i].__class__.
__name__, rtotal)

    np.testing.assert_allclose(rtotal, np.sum(net3.modules[-1].R) + np.s
um(net3.modules[-1].Rb), err_msg='Relevance Conservation Criterion viola
ted at layer {} ({})'.format(i, net3.modules[i].__class__.__name__))
```

```
Total relevance per layer for net1 with bias proxy neurons
layer 5 (Logistic) Total: 6.99566429637
layer 4 (Linear) Total: 6.99566429637
layer 3 (Rect) Total: 6.99566429637
layer 2 (Linear) Total: 6.99566429637
layer 1 (Rect) Total: 6.99566429637
layer 0 (Linear) Total: 6.99566429637

Total relevance per layer for net2 with bias proxy neurons
layer 5 (Logistic) Total: 6.99566429637
layer 4 (Linear) Total: 6.99566429637
layer 3 (Rect) Total: 6.99566429637
layer 2 (Linear) Total: 6.99566429637
layer 1 (Rect) Total: 6.99566429637
layer 0 (Linear) Total: 6.99566429637

Total relevance per layer for net3 with bias proxy neurons and custom inp
ut layer
layer 5 (Logistic) Total: 6.99566429637
layer 4 (Linear) Total: 6.99566429637
layer 3 (Rect) Total: 6.99566429637
layer 2 (Linear) Total: 6.99566429637
layer 1 (Rect) Total: 6.99566429637
layer 0 (CustomLinear) Total: 6.99566429637
```

Since we *unrolled* the input shift compensating forward pass in Net3 to $w^T x_2 + b_1 - w^T m$, we can easily visualize that the input activations m subtracted from x_1 to obtain x_2 -- which have then been added to the forward pass again as $-w^T m$ to achieve the same neuron activations as Net1 and Net2 (instead of packaging them into the bias b_2 as in Net2) -- will absorb the relevance quantities describing the difference between the pixel-wise heatmaps obtained with Net1 for some input of choice and Net2 with the same input under constant shift m.

In [17]:
```
#test if the difference in relevance attribution between net1 and net3 c
an be found in the neurons describing -m, the shift compensator term
rx1 = net1.modules[0].R
rx3 = net3.modules[0].R
rm3 = net3.modules[0].Rm
#numerically assert that what we are about to show is true
np.testing.assert_allclose(x1+m, x3)     #original inputs shifted are eq
ual to shifted inputs
np.testing.assert_allclose(rx1, rx3+rm3) #relevances for original digits
are equal to heatmaps for shifted digits plus heatmaps for shift compens
ator term


#compute a common maximal absolute value for color space alignment for t
he relevace and sensitivity maps
v = max(np.abs(rx1).max(), np.abs(rx3).max(), np.abs(rm3).max())




#show the inputs and relevance maps as images in a 3 by 2 subplot
plt.subplot(241)
plt.imshow(x1.reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.title('input net1')

plt.subplot(242)
plt.imshow(m.reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.title('shift m')
plt.ylabel('+')

plt.subplot(243)
plt.imshow((x1+m).reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('=  ',rotation=0)

plt.subplot(244)
plt.imshow((x3).reshape(28,28), vmin=-1., vmax=1.)
plt.xticks([],[]); plt.yticks([],[])
plt.title('input net3')
plt.ylabel('--------------------')


plt.subplot(245)
plt.imshow(rx1.reshape(28,28), vmin=-v , vmax=v)
plt.xticks([],[]); plt.yticks([],[])
plt.title('R net1 @ x1')

plt.subplot(246)
plt.imshow(-rm3.reshape(28,28), vmin=-v , vmax=v)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('+')
plt.title('R net3 @ -m')

plt.subplot(247)
plt.imshow((rx1-rm3).reshape(28,28), vmin=-v , vmax=v)
plt.xticks([],[]); plt.yticks([],[])
plt.ylabel('=  ',rotation=0)

plt.subplot(248)
plt.imshow(rx3.reshape(28,28), vmin=-v , vmax=v)
plt.xticks([],[]); plt.yticks([],[])
plt.title('R net3 @ x3')
plt.ylabel('--------------------')


plt.show()
```



Above plots demonstrate that LRP does in no way attribute its relevance as a variant of saliency incorrectly. It works as intended and is able to successfully identify the source of a neuron activation despite, or rather *because* it is not invariant to constant shifts in the input space.

TL;DR: The (Un)reliability of saliency methods you trains a fully network on MNIST for digit classification. All layers are linear ones, followed by ReLU activation units, except at the output. The forward pass of the input layer is thus $w^T x + b$, with x being the input (image), w the learned set of weights (on the [0,1]-normalized pixel values) and b the learned bias.

You then modify the data x and create $x_2 = x + m$, with m being a constant shift applied to all samples on the original data and copy the original network with the intent to use it to predict on x_2. With goal to reproduce the output (and hidden activations) in modified network, you adapt the input layer forward pass to
$w^T x_2 + b_2 = \underbrace{w^T(x+m)}_{\text{f.p. on shifted data}} + \underbrace{w^T(-m)}_{\text{compensating the shift}} + b$, i.e. by using the law of distributivity, you constantly subtract from the forward pass in the input layer, what you uniformly add to all samples x. Now instead of hiding the term $w^T(-m)$ designed to compensate for the shift in hidden unit activation caused by the translation of the input data in an adapted bias term b_2, we can construct it as a constant and image-shaped bias input, as shown somewhere above. The activations of all hidden units might be exactly the same for the original net and its adapted copy, but the composition of source (neuron) activations of the output neuron activations differs.

At hand of LRP -- a method designed to provide an analysis of a prediction in terms of *the source of the decision made* and wrt to *the interaction of the input(s) and the model itself in the forward pass* -- we provide a counter example demonstrating that input invariance is not a desired property for all semantics of saliency: LRP attributes, wrt to the purpose it was designed for, its saliency scores (aka "relevance") correctly. BTW: For the chosen network architecture, LRP defaults to the Gradient times Input (GI) algorithm (without bias neurons present) , closing the loop to the evaluation of GI in your manuscript.

The key message here is to consider the goals the compared and evaluated saliency methods try to achieve, and the semantic context the obtained saliency maps try to convey. For some methods, input invariance may be a requirement for reliable attribution while for others it clearly is not and would, if fulfilled, even break the method. Coherently, it is important to restrict the comparing experiments to methods where this is a desired property and communicate the findings accordingly.