Lab 9 - Convolutional Neural Networks

In this lab you'll use PyTorch (https://pytorch.org/) to:

- train a *convolutional neural network* (CNN, or ConvNet) on 1D synthetic data and on 2D real data;
- learn how to inspect the filters and the feature maps of your model, to ensure you understand the inner workings (intermediate representations) that are computed during the feed-forward computation of a ConvNet;
- learn how a ConvNet training can get stuck in a local minimum, and how increasing the number of filters helps.

If you really want to understand convolutions in detail, you can also download the *convolution_layer_demo.zip* file in the course Moodle page and run the Jupyter notebook inside. (All the code in that notebook is done for you.)

Run the code cell below to import the required packages.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import copy
np.set_printoptions(precision=3, suppress=True)  # Print as 0.001 instead of 9.876e-4
torch.set_printoptions(precision=3, sci_mode=False)
from sklearn.preprocessing import MinMaxScaler
from IPython.core.display import HTML
HTML("""
<style>
.output_png {
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
</style>
""")
```

Out[1]:

Run the code cell below to define a plotting function that is useful for visualizing the weights of neural networks and images.

In [2]:

```
def plot_matrix_grid(V, cmap='bwr'):
    """
    Given an array V containing stacked matrices, plots them in a grid layout.
    V should have shape (K,M,N) where V[k] is a matrix of shape (M,N).
    The default cmap is "bwr" (blue-white-red) but can also be "gray".
    """
    if isinstance(V, torch.Tensor):
        V = V.detach().numpy()
    assert V.ndim == 3, "Expected V to have 3 dimensions, not %d" % V.ndim
    k, m, n = V.shape
    ncol = 8                                    # At most 8 columns
    nrow = min(4, (k + ncol - 1) // ncol)       # At most 4 rows
    V = V[:nrow*ncol]                           # Focus on just the matrices we'll act
ually plot
    figsize = (2*ncol, max(1, 2*nrow*(m/n)))    # Guess a good figure shape based on n
col, nrow
    fig, axes = plt.subplots(nrow, ncol, sharex=True, sharey=True, figsize=figsize)
    vmax = np.percentile(np.abs(V), [99.9])     # Show the main range of values, betwe
en 0.1%-99.9%
    for v, ax in zip(V, axes.flat):
        img = ax.matshow(v, vmin=-vmax, vmax=vmax, cmap=plt.get_cmap(cmap))
        ax.set_xticks([])
        ax.set_yticks([])
    for ax in axes.flat[len(V):]:
        ax.set_axis_off()
    fig.colorbar(img, cax=fig.add_axes([0.92, 0.25, 0.01, .5]))   # Add a colorbar on t
he right
```

---

1. Convolutional Neural Networks on Synthetic Data

Exercise 1.1–1.4 ask you to generate a synthetic data set and to inspect how a 1D convolutional neural network learns "pattern detectors" on it. Building a toy synthetic dataset is good practice when sanity-checking a model that you have never worked with. It will be easy to plot all the features, targets, and predictions all at once.

Run the code cell below to define a function that will be able to plot your synthetic training set.

In [3]:

```
def plot_named_tensors(tensor_dict):
    """
    Given a dict of {name: tensor} pairs, plots the tensors side-by-side in a common
    color scale. The name of each tensor is shown above its plot.
    """
    n = len(tensor_dict)
    vmax = max(v.abs().max() for v in tensor_dict.values())
    figsize = (2*n, 6)
    fig, axes = plt.subplots(1, n, figsize=figsize,  constrained_layout=True, squeeze=T
rue)
    axes = axes.flat if isinstance(axes, np.ndarray) else (axes,)
    for (name, v), ax in zip(tensor_dict.items(), axes):
        v = torch.squeeze(v.detach())   # Automatically convert (N,1,D) to (N,D)
        if v.ndim == 1:
            v = v.view(-1, 1)  # Automatically convert (N,) to (N,1)
        assert v.ndim == 2, "couldn't turn tensors[%d] with shape %s into 2D" % (i, v.s
hape)
        img = ax.matshow(v, vmin=-vmax, vmax=vmax, cmap=plt.get_cmap('bwr'))
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(name)
    fig.colorbar(img, cax=fig.add_axes([0.985, 0.25, 0.03, .5]))   # Add a colorbar on
 the right
```

Exercise 1.1 – Build a synthetic 1D data set

Start by building a synthetic training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where the goal is to "detect pattern $[0, 1, 0]$ in the input".

- Each input $\mathbf{x}_i \in \mathbb{R}^{20}$ represents a single-channel 'image' of width $W = 20$ having values from $\{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\}$.
- Each target $y_i \in \{0, 1\}$ represents a class label, with $y_i = 1$ if the subsequence $[0, 1, 0]$ appears somewhere in $\mathbf{x}_i$.
- The dataset should have $N = 75$ items.

In order to apply **torch.nn.Conv1d** (https://pytorch.org/docs/stable/nn.html#torch.nn.Conv1d) module to the inputs, you'll need to represent them in a single tensor having "(N, C, W) format", which here means $\mathbf{X} \in \mathbb{R}^{75 \times 1 \times 20}$. Likewise you'll need the targets to have shape $\mathbf{y} \in \mathbb{R}^{75 \times 1}$. However, to generate random numbers it is a little simpler to use Numpy like **np.random.randint** (https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html) and **np.random.choice** (https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html). Note that the *randint(a,b)* returns an integer strictly less than *b*.

Specific steps to build $\mathbf{X}$ and $\mathbf{y}$, that you will have to implement in Python:

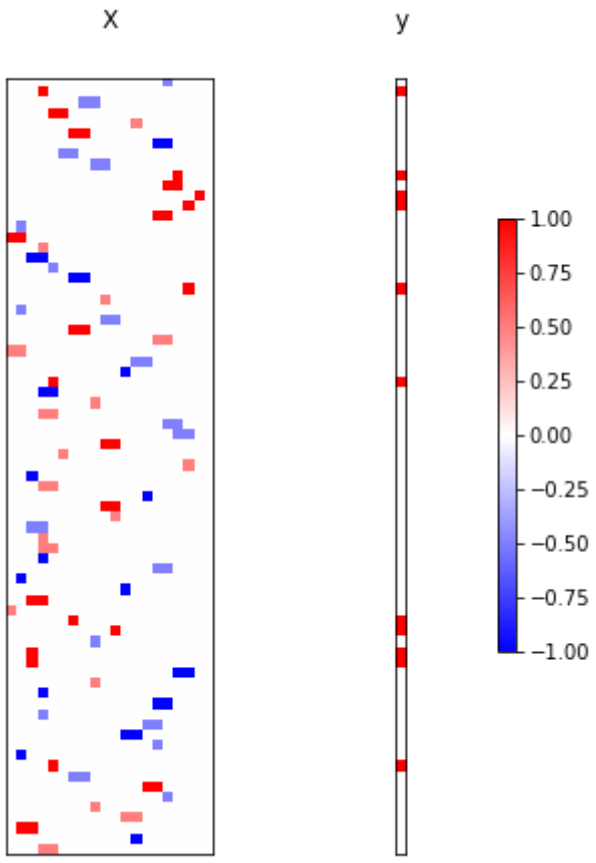
1. Create $\mathbf{X}$ and $\mathbf{y}$ as torch tensors full of zeros.
2. For each row $i = 0, \ldots, N - 1$:
   A. Sample a random integer $\text{size} \in \{1, 2\}$, *i.e.* the size of the subrange of pixels that will be assigned.
   B. Sample a random integer $\text{start} \in \{0, \ldots, W - \text{size} - 1\}$, *i.e.* the offset from the start of the image.
   C. Sample a random choice $\text{value} \in \{-1, -\frac{1}{2}, \frac{1}{2}, 1\}$, *i.e.* the value to fill for the subrange.
   D. Assign a stretch of features $X[i, 0, \text{start} : \text{start} + \text{size}] = \text{value}$, which will either be size 1 or 2.
   E. Assign target $y[i]$ to be 1 if both $\text{size} = 1$ and $\text{value} = +1$, *i.e.* if $X[i, 0, :]$ has a single "+1" pixel somewhere in it.

Complete the starter code below by completing the for loop. If PyTorch complains about not being able to convert *numpy.int32* or *numpy.bool_* to *torch.FloatTensor* you may need to explicitly convert the value to float using *float(value)*.

In [4]:

```
np.random.seed(0)  # For reproducibility

N = 75  # N = number of training cases
C = 1   # C = number of channels (just 1 in our case)
W = 20  # W = width of the 1-dimensional input image

# We create feature tensor X in "(N,C,W) format" (the shape) so that it can
# be used directly as input to PyTorch's Conv1D module.
X = torch.zeros((N, C, W))   # Start with zeros. You need to assign some of these featu
re values!
y = torch.zeros((N, 1))      # Start with zeros. You need to assign some of these targe
t values!

for i in range(N):
    size = np.random.randint(1, 3)
    start = np.random.randint(0, 20-size)
    value = np.random.choice([-1, -0.5 , 0.5, 1])
    X[i,0,start:start+size] = value
    if size == 1 and value == 1:
        y[i] = 1

    # Your code for initializing X and y here. Aim for 5-7 lines.
```

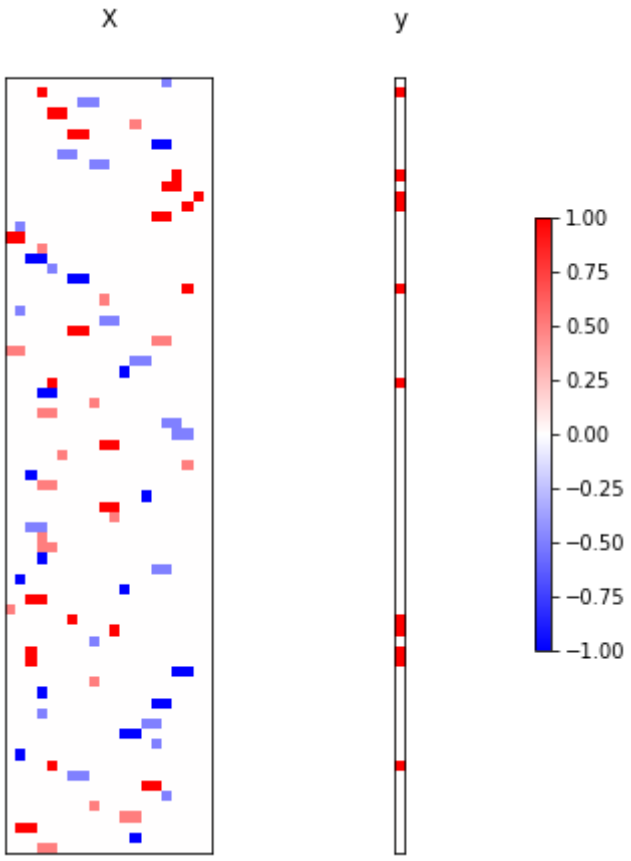
Check your answer by running the code cell below. It will plot the entire training set as a heatmap, where blue indicates negative values and red indicates positive values. If you initialized your training set correctly (and kept `np.random.seed(0)` above) you should see exactly this training set:



Here we have $N = 75$ rows and $W = 20$ columns shown for the $\mathbf{X}$ tensor. Notice that the red dots in the $\mathbf{y}$ vector correspond to the $y_i = 1$ for which there is exactly a 1-pixel red dot (a $+1$) in the corresponding input image.

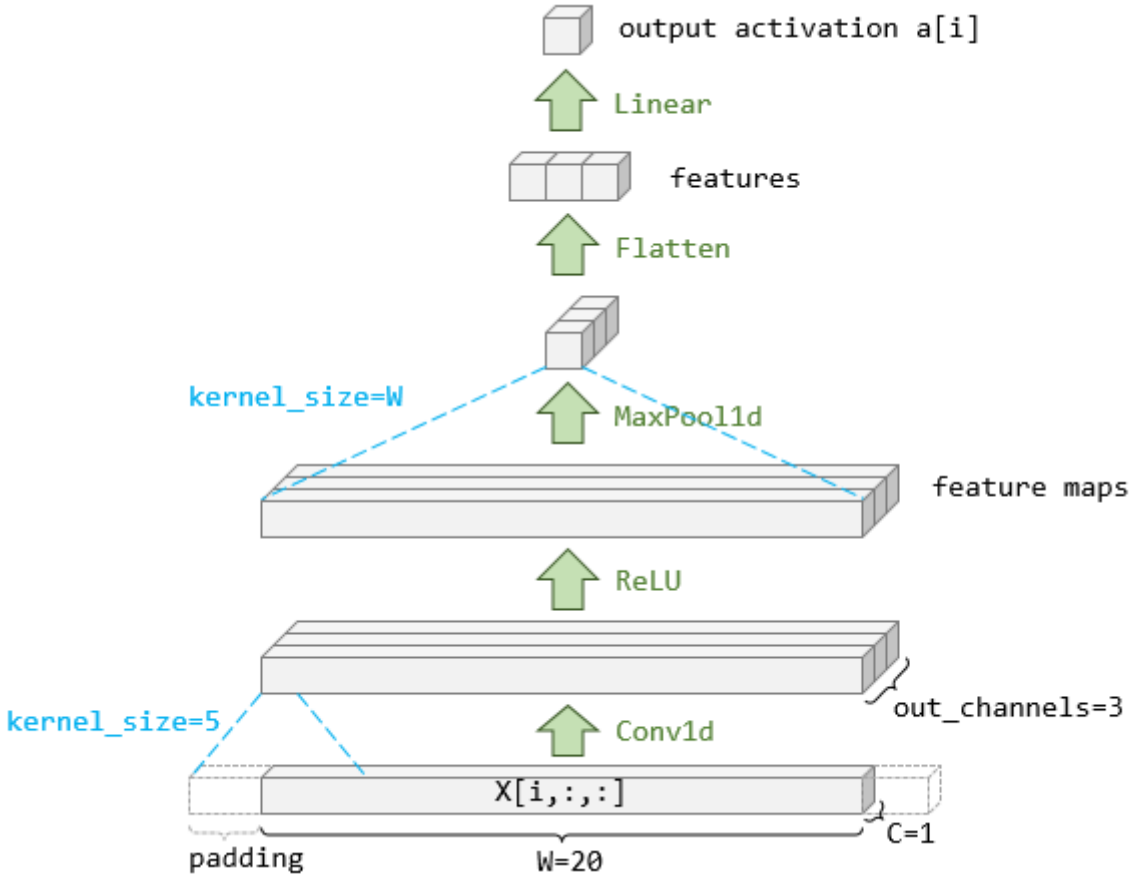
In [5]:

```
plot_named_tensors({'X': X, 'y': y})
```



Exercise 1.2 – Define a ConvNet architecture suitable for classifing your synthetic data

You are asked to define a 1D convolutional neural network with the following feed-forward architecture built from the PyTorch modules (https://pytorch.org/docs/stable/nn.html) mentioned. The architecture should transform an input tensor $X \in \mathbb{R}^{N \times C \times W}$ into a vector of scalar activations $\mathbf{a} \in \mathbb{R}^{N \times 1}$.



Comments on the architecture:

- The *convolutional* layer is depicted as having exactly enough padding (implicitly zero) to ensure that the output feature maps have the same spatial length ($W = 20$) as the input vector had. Notice that if you were to increase the filter size by 2, you need to increase the padding by 1 (padding appears on both ends) to keep the output of convolution the same length as before. See the documentation for the Conv1d module.
- The *max pooling* layer is being used here to take the maximum value across each of the 3 feature maps shown. Since we want to take the max over the entire spatial extent of the feature map, we use a large *kernel_size*.
- The 1-dimensional convolution and pooling operations require *spatial* data in (N, C, W) format (where W is the spatial dimension), so that the operators know how long the spatial dimension is. However the *linear* (fully-connected) layer doesn't know how to deal with spatial data, so the *flatten* operation simply reshapes the tensor from shape (N, C, W) to shape (N, D) where $D = C \times W$.
- Even though we'll use this architecture for binary classification, we do not add an extra *sigmoid* operation directly to the output. This is done for same reasons as for the multi-class PyTorch neural network from Lab 8. After the model is defined, we can still use it to predict binary class probabilities $\hat{y}_i \in [0, 1]$ by feeding features X through the model and then feeding the resulting activations $\mathbf{a}$ through a sigmoid so that class predictions are $\hat{\mathbf{y}} = \sigma(\mathbf{a})$.

Add a few lines of code to define a PyTorch model with the above architecture.

In [6]:

```
torch.manual_seed(0) # Ensure model weights initialized with same random numbers
import torch.nn as nn
num_filter = 3   # The number of filters to learn
filter_size = 5  # The size of each filter

model = torch.nn.Sequential(
    nn.Conv1d(1,num_filter,filter_size,padding=filter_size//2),
    nn.ReLU(),
    nn.MaxPool1d(20),
    nn.Flatten(),
    nn.Linear(num_filter,1)
    # Your code for defining the model architecture here. Aim for 5-9 lines.
)
```

Check your model architecture by running the code cell below.

In [7]:

```
assert len(model) == 5, "Should be 5 layers!"
assert isinstance(model[0], torch.nn.Conv1d), "layer 0 should be Conv1d"
assert model[0].in_channels == C, "layer 0 should expect C input channel"
assert model[0].out_channels == num_filter, "layer 0 should expect %d output channels"
% num_filter
assert model[0].kernel_size[0] == filter_size, "layer 0 filter size should be %d" % fil
ter_size
assert model[0].padding[0] == filter_size//2
assert isinstance(model[1], torch.nn.ReLU), "layer 1 should be ReLU"
assert isinstance(model[2], torch.nn.MaxPool1d), "layer 2 should be MaxPool1d"
assert model[2].kernel_size == W, "layer 2 should pool over the entire input feature ma
p"
assert isinstance(model[3], torch.nn.Flatten), "layer 3 should by Flatten"
assert isinstance(model[4], torch.nn.Linear), "layer 4 should be Linear"
assert model[4].in_features == num_filter, "layer 4 should have accept %d inputs" % num
_filter
assert model[4].out_features == 1, "layer 4 should have only 1 output"
print("Looks OK!")
```

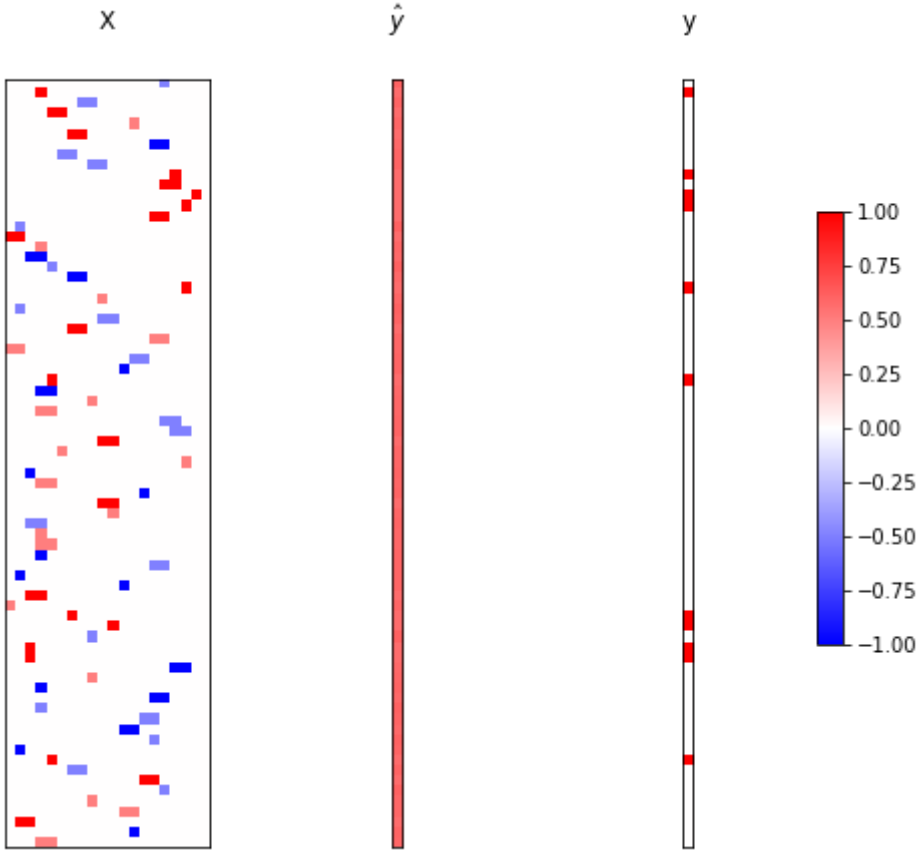
```
Looks OK!
```

Plot your model's initial predictions (i.e. *before* training) by running the code cell below. Notice that the untrained model predicts arbitrary class probabilies that all look similar. This is because the initial weights of the Conv1d and Linear layers are small random values, so the output is only weakly sensitive to the input.

In [8]:

```
y_pred = torch.sigmoid(model(X))   # Turn activations into probabilities by feeding thr
ough sigmoid
print(y_pred[:5])                  # Print the first few probabilities
plot_named_tensors({'X': X, '$\hat{y}$': y_pred, 'y': y})
```

```
tensor([[0.611],
        [0.575],
        [0.604],
        [0.576],
        [0.597]], grad_fn=<SliceBackward>)
```



Exercise 1.3 – Train the ConvNet on your synthetic data

Here you should train the ConvNet architecture from Exercise 1.2, much as you trained a PyTorch neural network in the Lab 8.

First, **run the code cell below** to make a copy of the untrained model from Exercise 1.2. This will make it easy to re-run the training code cell multiple times, each time starting from an untrained model (rather than continuing to train the same model!).

In [9]:

```
untrained_model = copy.deepcopy(model)
```

Next you'll need to define a loss function and implement a standard PyTorch training loop to train a copy of the untrained model. The PyTorch training loop for a ConvNet is identical to that of a neural network, so use Lab 8 as a guide.

However, for this exercise use the following configuration:

- *No mini-batches:* Unlike Lab 8, for this exercise do not use mini-batches. Instead feed the whole X matrix through your model, like `model(X)` to get $N = 75$ outputs. In other words, your training loop doesn't need an inner "mini-batch" loop.
- *Loss function:* Use the "binary cross entropy" loss that directly accepts activations ('logits'). In PyTorch this is implemented by the [torch.nn.BCEWithLogitsLoss (https://pytorch.org/docs/stable/nn.html#torch.nn.BCEWithLogitsLoss)](https://pytorch.org/docs/stable/nn.html#torch.nn.BCEWithLogitsLoss) module. This is similar to the *CrossEntropyLoss* in Lab 8, except it's specialized to handle a single binary output.
- *Optimizer:* This is the learning algorithm. Use the [torch.optim.SGD (https://pytorch.org/docs/stable/optim.html#torch.optim.SGD)](https://pytorch.org/docs/stable/optim.html#torch.optim.SGD) optimizer with learning rate 0.05, momentum 0.9, and weight decay (weight penalty) of 0.001.

Your training code should also print the current training loss for the first epoch and every 50 epochs after. The code for doing so is similar to in Lab 8. For example, if you computed your loss in variable `loss_value` then something like this would work:

```
for epoch in range(1, num_epoch+1):

    ... your code to apply a step of gradient descent here

    if epoch == 1 or epoch % 50 == 0:
        print("Epoch %d had training loss %.4f" % (epoch, loss_value.item()))
```

where the `.item()` is a method that converts a scalar-valued PyTorch tensor into a standard Python value, like a *float*, so that it can be more easily formatted as part of the string.

The output of your training loop should look exactly like this:

```
Epoch 1 had training loss 0.8562
Epoch 50 had training loss 0.3519
Epoch 100 had training loss 0.3281
Epoch 150 had training loss 0.3212
...
```

However, the loss should not go to zero for the current model architecture. You'll fix that shortly.

In [10]:

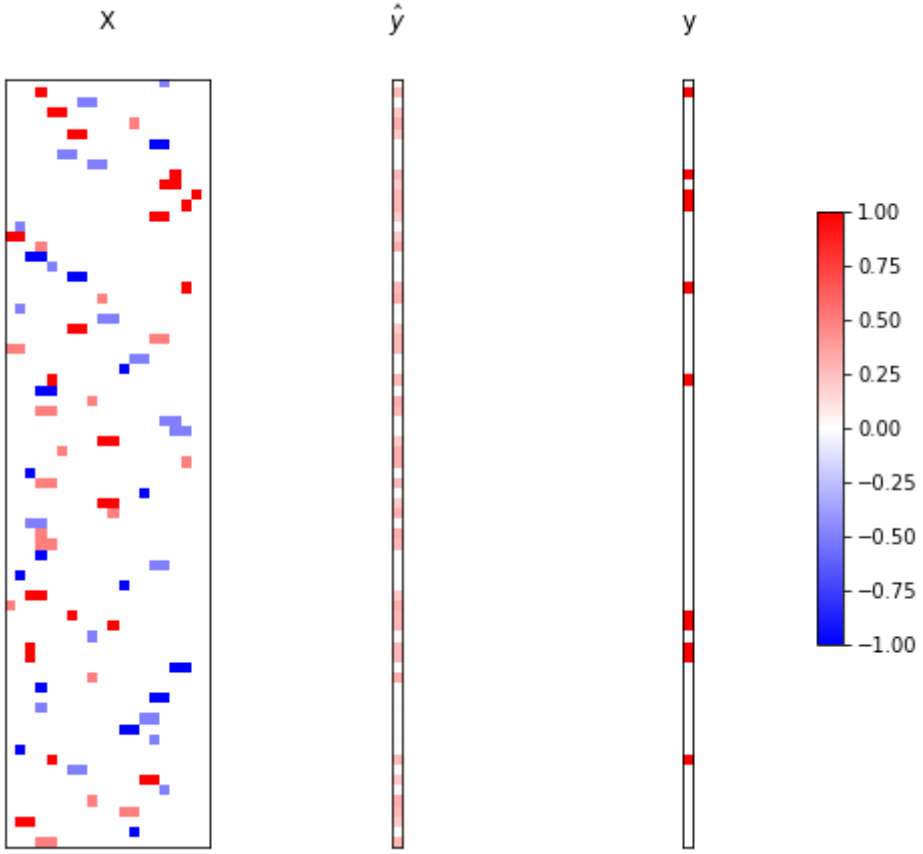
```
from torch import optim

num_epoch = 500
losses = []
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model.parameters(), lr=0.05, momentum=0.9,weight_decay=0.001)

for epoch in range(1, num_epoch+1):
        # forwad pass
        y_pred = model(X)

        # loss
        loss = criterion(y_pred, y)

        # backward pass
        optimizer.zero_grad()
        loss.backward()

        # update parameters
        optimizer.step()

        # report
        if epoch == 1 or epoch % 50 == 0:
            print("Epoch %d had training loss %.4f" % (epoch, loss.item()))
```

```
Epoch 1 had training loss 0.8562
Epoch 50 had training loss 0.3519
Epoch 100 had training loss 0.3281
Epoch 150 had training loss 0.3212
Epoch 200 had training loss 0.3183
Epoch 250 had training loss 0.3165
Epoch 300 had training loss 0.3150
Epoch 350 had training loss 0.3137
Epoch 400 had training loss 0.3128
Epoch 450 had training loss 0.3119
Epoch 500 had training loss 0.3113
```

Plot your trained model's predictions in the code cell below. You should see that the prediction vector has an interesting pattern now (unlike the initial model), but that the predictions still do not quite match the targets.

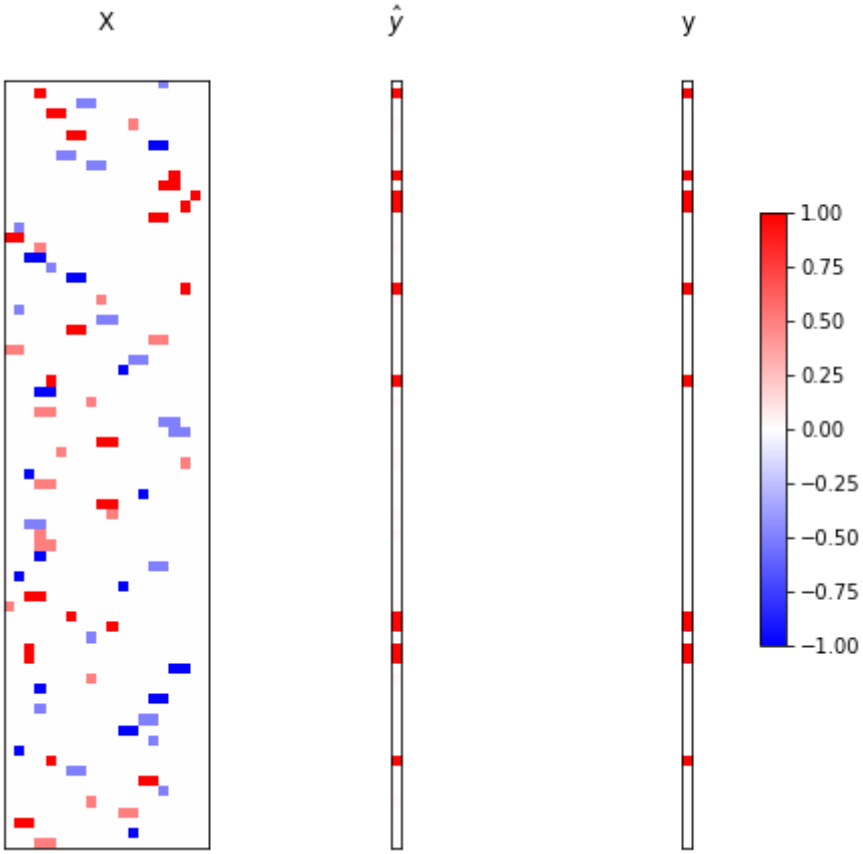
In [11]:

```
y_pred = torch.sigmoid(model(X))   # Turn activations into probabilities by feeding thr
ough sigmoid
plot_named_tensors({'X': X, '$\hat{y}$': y_pred, 'y': y})
# Your code here. See Exercise 1.2.
```



The mismatch between predictions and targets is why the loss did not get close to zero, despite the fact that your ConvNet architecture can, in principle, learn a perfect classifier.

If your model architecture was able to get the training loss to zero, the predictions would look more like this:



Currently, your ConvNet training got stuck in a local minimum! To give your ConvNet architecture a better chance at "getting lucky" and finding a path from random weights to useful weights, **try going back to Exercise 1.2 and increase the number of filters from 3 to 4**. You will then have to re-run all the code cells up to this point. With an extra filter, your ConvNet is now "lucky" enough to find at least one filter that detects a $0, 1, 0$ pattern.

In [12]:

```
torch.manual_seed(0) # Ensure model weights initialized with same random numbers
torch.nn.Flatten(start_dim=1,end_dim=-1)
num_filter = 4   # The number of filters to learn
filter_size = 5  # The size of each filter

model2 = torch.nn.Sequential(
    nn.Conv1d(1,num_filter,filter_size,padding=filter_size//2),
    nn.ReLU(),
    nn.MaxPool1d(np.size(X,2)),
    nn.Flatten(),
    nn.Linear(num_filter,1)
    # Your code for defining the model architecture here. Aim for 5-9 lines.
)
```

In [13]:

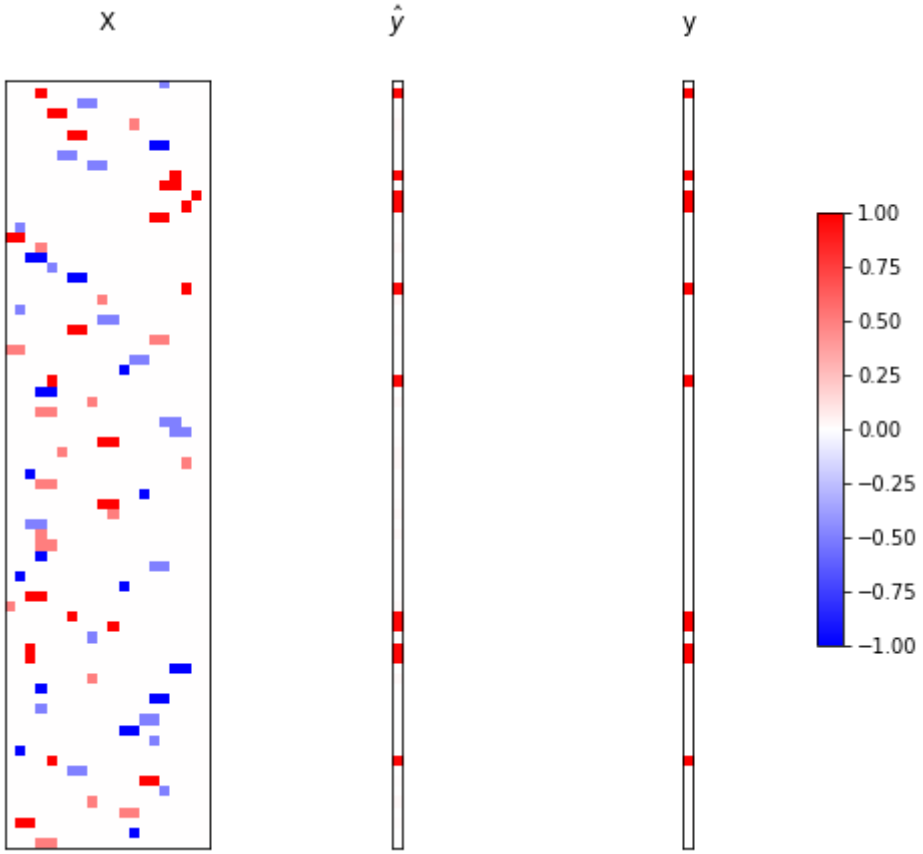
```
num_epoch = 500
losses = []
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model2.parameters(), lr=0.05, momentum=0.9,weight_decay=0.001)

for epoch in range(1, num_epoch+1):
        # forwad pass
        y_pred = model2(X)

        # loss
        loss = criterion(y_pred, y)

        # backward pass
        optimizer.zero_grad()
        loss.backward()

        # update parameters
        optimizer.step()

        # report
        if epoch == 1 or epoch % 50 == 0:
            print("Epoch %d had training loss %.4f" % (epoch, loss.item()))
```

```
Epoch 1 had training loss 0.5462
Epoch 50 had training loss 0.3510
Epoch 100 had training loss 0.2914
Epoch 150 had training loss 0.1697
Epoch 200 had training loss 0.0635
Epoch 250 had training loss 0.0318
Epoch 300 had training loss 0.0205
Epoch 350 had training loss 0.0153
Epoch 400 had training loss 0.0125
Epoch 450 had training loss 0.0107
Epoch 500 had training loss 0.0095
```

In [14]:

```
y_pred = torch.sigmoid(model2(X)) # Turn activations into probabilities by feeding thro
ugh sigmoid
print(y_pred[:5])
plot_named_tensors({'X': X, '$\hat{y}$': y_pred, 'y': y})
# Your code here. See Exercise 1.2.
```

```
tensor([[    0.000],
        [    0.971],
        [    0.000],
        [    0.008],
        [    0.024]], grad_fn=<SliceBackward>)
```



Exercise 1.4 Inspect the filters and weights of your ConvNet

Here you should visualize both the filter weights and the linear (fully-connected) weights. Use the *plot_matrix_grid* function. If you managed to get your training loss to be small (close to zero) then at least one of your filters should look like this:

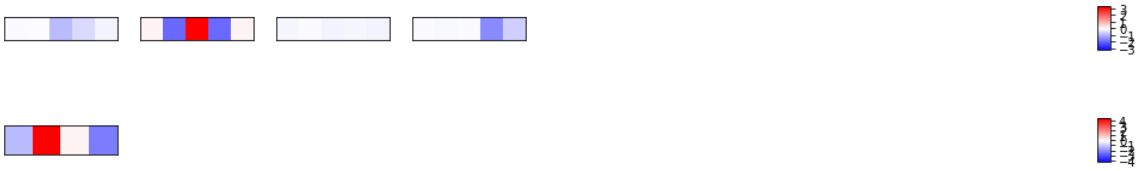
where red means positive and blue means negative. When this filter is convolved with a sequence containing pattern $\ldots, 0, 1, 0, \ldots$ it will "activate" and have a large output at the position centered on the 1. That activation then gets selected by the max pooling operation, and is then used to activate the output of the network, giving correct classification.

Write a few lines of plotting code to visualize the weights of both layers. Do you see a correspondence between the filter that looks "right" and the positive weight(s) in the linear layer?

In [15]:

```
W1, b1, W2, b2 = model2.parameters()
print(W1.shape)  # The tensor of 4 filters, each with 1 channel and kernel size 5
print(W2.shape)  # The matrix of linear (fully-connected) weights that combine the filt
er responses

# Your plotting code here. Aim for 2-4 lines of code.
plot_matrix_grid(W1)
plot_matrix_grid(W2.reshape(1,1,4))
```

```
torch.Size([4, 1, 5])
torch.Size([1, 4])
```



---

2. Digit classification with Convolutional Neural Networks

Exercise 2.1–2.2 expand on Lab 8, this time asking you to train a convolutional neural network on the MNIST data set rather than a fully-connected neural network.

Exercise 2.1 Load and preprocess MNIST

Implement the *load_mnist_for_convnet* function below. Rather than normalizing the features with scikit-learn, perform a simple normalization by scaling the pixel intensities from range $[0, 255]$ down to $[0, 1]$. This will be good enough.

In [16]:

```
def load_mnist_for_convnet(filename):
    """
    Loads the MNIST data from a Numpy NPZ file and returns two PyTorch tensors:
    X: a float tensor with shape (N,1,28,28) where N is the number of images in the fil
e
    y: an int64 tensor with shape (N,) containing the class targets for the images.
    The pixels values are scaled to be in range [0,1] where 0 is black and 1 is white.
    """
    # Your code here. Aim for 7-10 lines.
    data = np.load(filename)
    X = data['X']
    y = data['y']

    N = X.shape[0]

    X = np.reshape(X,(N,1,28,28))

    X_t = torch.from_numpy(X/abs(X).max()).float()
    y_t = torch.from_numpy(y).long()

    return X_t, y_t


X_trn, y_trn = load_mnist_for_convnet("mnist_train.npz")
X_tst, y_tst = load_mnist_for_convnet("mnist_test.npz")
```

Check your answer by running the code cell below.

In [17]:

```
assert isinstance(X_trn, torch.FloatTensor), "Features should be float32!"
assert isinstance(y_trn, torch.LongTensor), "Targets should be int64 or long!"
assert X_trn.shape == (60000, 1, 28, 28), "X_trn has wrong shape"
assert y_trn.shape == (60000,), "y_trn has wrong shape"
assert X_tst.shape == (10000, 1, 28, 28), "X_tst has wrong shape"
assert y_tst.shape == (10000,), "y_tst has wrong shape"
assert X_trn.min() == 0 and X_trn.max() == 1, "Features don't seem to be scaled right!"
print("Looks good!")
```

```
Looks good!
```

Exericise 2.2 Train a Convolutional Neural Network on MNIST

You are asked to train a 2D ConvNet to classify MNIST digits. You'll need the *Conv2d* and *MaxPool2d* modules as described in the PyTorch modules documentation (https://pytorch.org/docs/stable/nn.html). Your architecture will be similar to Part 1 of this lab, except rather than pooling over the entire input signal you will pool only over small 2x2 regions, preserving more spatial information for the subsequent linear (fully-connected) layer to use in classification.

Write a few lines of code to define a 2D ConvNet with the following feed-forward architecture:

1. Convolution with 8 filters each of size 5x5 and enough padding to ensure that each of the resulting feature maps has spatial dimensions 28x28, just like the input.
2. Rectified linear transformation (ReLU).
3. Max pooling with kernel size 2x2 and a non-overlapping stride (i.e. stride same as kernel size).
4. Flatten the spatial information
5. Linear (fully-connected) layer to use all the outputs of the max pooling layer as features to make a prediction.

(To predict class probabilities you'll have to apply a softmax function to the output, but that is normal in many deep learning frameworks because it makes training more numerically stable.)

Hint: the trickiest part

In [18]:

```
torch.manual_seed(0) # Ensure model weights initialized with same random numbers

num_filter = 8   # The number of filters to learn
filter_size = 5  # The size of each filter

model = torch.nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=num_filter, kernel_size=filter_size, padding=
2),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2,stride=2),
    nn.Flatten(),
    nn.Linear(in_features=1568, out_features=10)
)
# Your code here. Aim for 8-11 lines.
```

Check your model architecture by trying to feed some inputs through. If an error is raised, then something is mis-configured in your network. The most likely error is that you did not correctly calculate the expected number of *in_features* for the linear layer, leading to a "size mismatch" in the tensors.

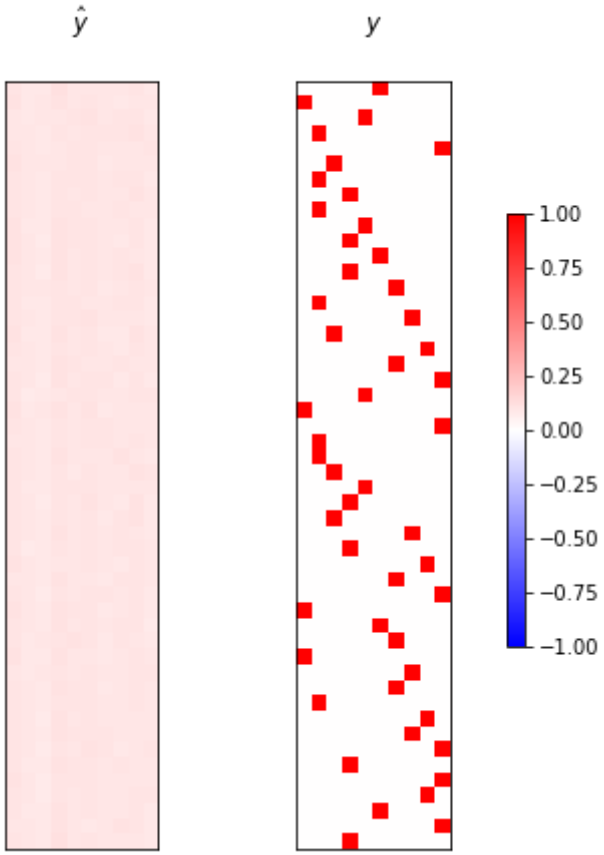
In [19]:

```
y_pred = model(X_trn[:5])  # Check model architecture by trying to feed inputs through
 it.
assert y_pred.shape == (5, 10), "Expected a batch of 5 images to produce output of shap
e (5, 10)"
```

Plot your initial model's predictions on a the first 30 training inputs by running the code cell below. Notice that once again all the predictions are pretty much the same, and are far from the targets.

In [20]:

```
# Feed the model's predictions on 30 training cases through a row-wise softmax.
y_pred = torch.softmax(model(X_trn[:50]), dim=1)

# Convert the first 30 training targets from index format {0,...,9} to a 1-hot format,
# for easier side-by-side comparison with the 10-dimensional output prediction.
y_true = torch.zeros((50, 10))
y_true[torch.arange(50), y_trn[:50]] = 1

plot_named_tensors({'$\hat{y}$': y_pred, '$y$': y_true})
```

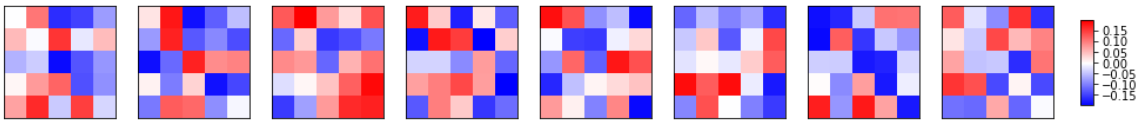


Inspect your initial model's filters by plotting them with the *plot_matrix_grid* function. They should look random and have small values roughly in range $[-0.1, 0.1]$ (Only plot the first layer filter weights, not the linear layer weights.)

In [21]:

```
W1, b1, W2, b2 = model.parameters()
print(W1.shape)  # The tensor of 4 filters, each with 1 channel and kernel size 5

plot_matrix_grid(W1.reshape(8,5,5))

# Your plotting code here. Aim for 1-2 lines of code.
```

```
torch.Size([8, 1, 5, 5])
torch.Size([10, 1568])
```



Train your model with the following configuration:

- Use the *CrossEntropyLoss* (since this is multi-class classification, not binary)
- Use the exact same optimizer configuration you used for Part 1 of this lab (learning rate 0.05, etc).
- Use mini-batch training like you did in the neural network lab, with a batch size of 100.
- Train for 5 epochs (5 passes over the full training set)

First, define the loss function and optimizer in a separate code cell below:

In [22]:

```
batch_size = 100
num_epoch = 5

loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.05, momentum=0.9,weight_decay=0.00
1)

# Your code to define loss function and optimizer here. Aim for 2 lines.
```

Then, write your training loop in a separate code cell below, so that you can re-run this code cell to "continue" training with the same optimizer object, if you want to train your model longer.

I recommend you add a print statement to report progress like we did in Lab 8, such as:

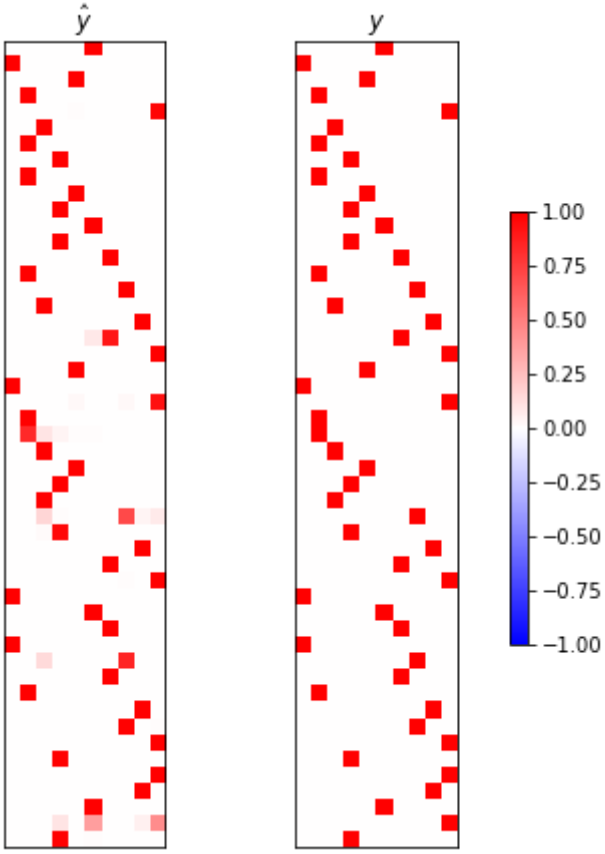
```
Epoch 1 final minibatch had loss 0.2452
Epoch 2 final minibatch had loss 0.2077
...
```

In [23]:

```
next_epoch = 1

for epoch in range(next_epoch, next_epoch+num_epoch):

    # Make an entire pass (an 'epoch') over the training data in batch_size chunks
    for i in range(0, len(X_trn), batch_size):
        X = X_trn[i:i+batch_size]     # Slice out a mini-batch of features
        y = y_trn[i:i+batch_size]     # Slice out a mini-batch of targets

        y_pred = model(X)                   # Make predictions (final-layer activation
s)
        l = loss(y_pred, y)                 # Compute loss with respect to predictions

        model.zero_grad()                   # Reset all gradient accumulators to zero
 (PyTorch thing)
        l.backward()                        # Compute gradient of loss wrt all paramete
rs (backprop!)
        optimizer.step()                    # Use the gradients to take a step with SG
D.

    print("Epoch %2d: final minibatch had loss: %.4f" % (epoch, l.item()))

print("Epoch %2d: loss on test set: %.4f" % (epoch, loss(model(X_tst), y_tst)))
next_epoch = epoch+1
# Your mini-batch training loop here. Aim for 9-12 lines.
```

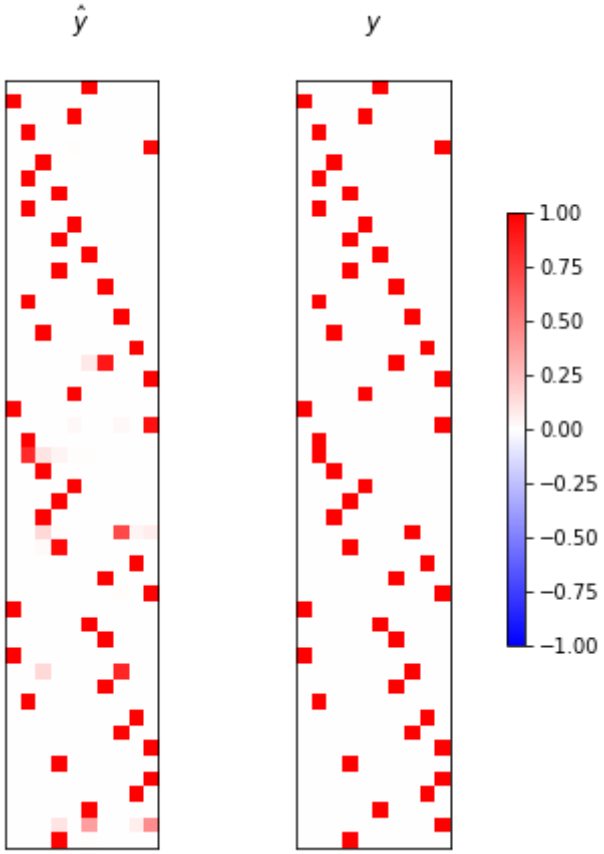
```
Epoch  1: final minibatch had loss: 0.2452
Epoch  2: final minibatch had loss: 0.2077
Epoch  3: final minibatch had loss: 0.1976
Epoch  4: final minibatch had loss: 0.1902
Epoch  5: final minibatch had loss: 0.1885
Epoch  5: loss on test set: 0.0640
```

Plot your trained model's predictions. The predictions should look like before, now matching the targets better than before training.

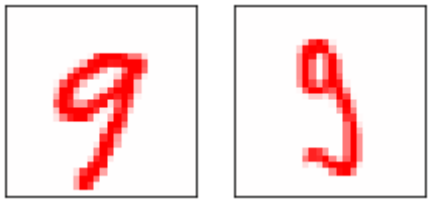


In [24]:

```
y_true = torch.zeros((50, 10))
y_true[torch.arange(50), y_trn[:50]] = 1
y_pred = torch.softmax(model(X_trn[:50]), dim=1)

plot_named_tensors({'$\hat{y}$': y_pred, '$y$': y_true})
# Your code here. Aim for 2-3 lines. Re-use the y_true matrix from the earlier code cel
l.
```



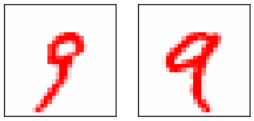
If you only trained your ConvNet for 5 epochs total, you should be able to identify at least one training case that appears to be "harder" than the others, *i.e.* the model has a harder time giving a confident prediction. For example, here is an easy "9" and a hard "9" side by side:

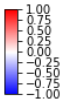


Plot an 'easy' training example and a 'hard' training example side-by-side using the *plot_matrix_grid* function. This will require you to pull out two separate rows of *X_trn* and turn them into a tensor with shape (2,28,28) so that the *plot_matrix_grid* knows what to plot.

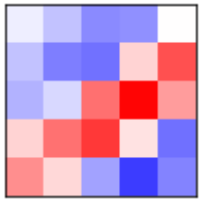
In [25]:

```
plot_matrix_grid(torch.vstack((X_trn[19,:],X_trn[4,:])))

# You code here. Aim for 1-4 lines.
```





Inspect your trained model's filters by plotting them with the *plot_matrix_grid* function. They should no longer be completely random and instead contain structures that look like little "edge" detectors, such as the "diagonal line" detector shown below



In [26]:

```
w1,w2,w3,w4 = model.parameters()
plot_matrix_grid(w1.reshape((8,5,5)))
# Your code here. Aim for 2-3 lines.
```