pytorch-basics-s22

February 15, 2022

covers various pytorch basics; intended for interactive use. –matus

1 tensor operations

[1]:
```
import torch
# torch has its own PRNG seeds.
# setting it here so notebook is deterministic.
torch.manual_seed(0)
```

[1]:
```
<torch._C.Generator at 0x7fb510b134f0>
```

[2]:
```
# create a non-inclusive range..
torch.arange(8)
```

[2]:
```
tensor([0, 1, 2, 3, 4, 5, 6, 7])
```

[3]:
```
# ..just like a regular python non-inclusive range
torch.tensor(range(8))
```

[3]:
```
tensor([0, 1, 2, 3, 4, 5, 6, 7])
```

[4]:
```
# a similar routine, subdividing an interval equally
torch.linspace(0, 1, 5)
```

[4]:
```
tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000])
```

[5]:
```
# standard arithmetic operations act coordinate-wise
xs = torch.linspace(0, 1, 5)
print(1, xs ** 2) # coordinate-wise squaring
print(2, xs * xs) # coordinate-wise multiplication
print(3, (xs ** 2 - xs * xs) < 1e-16) # compare the above,
print(4, (2.7182818 * xs).log()) # coordinate-wise multiplication and ln
```

```
1 tensor([0.0000, 0.0625, 0.2500, 0.5625, 1.0000])
2 tensor([0.0000, 0.0625, 0.2500, 0.5625, 1.0000])
3 tensor([True, True, True, True, True])
4 tensor([   -inf, -0.3863,  0.3069,  0.7123,  1.0000])
```

[6]:
```
# pytorch tensors aren't just floating point
print(1, torch.linspace(0,1,5).dtype)
print(2, torch.arange(5).dtype)
print(3, (torch.arange(5) / 10).dtype)
print(4, (torch.arange(5) // 10).dtype)
print(5, (torch.arange(5) == 0).dtype)
```

```
1 torch.float32
2 torch.int64
3 torch.float32
4 torch.int64
5 torch.bool
```

```
/home/matus/.local/lib/python3.9/site-packages/torch/_tensor.py:575:
UserWarning: floor_divide is deprecated, and will be removed in a future version
of pytorch. It currently rounds toward 0 (like the 'trunc' function NOT
'floor'). This results in incorrect rounding for negative values.
To keep the current behavior, use torch.div(a, b, rounding_mode='trunc'), or for
actual floor division, use torch.div(a, b, rounding_mode='floor'). (Triggered
internally at  /pytorch/aten/src/ATen/native/BinaryOps.cpp:467.)
  return torch.floor_divide(self, other)
```

[7]:
```
# arithmetic operations generally convert between types
print(1, torch.arange(5) * torch.arange(5))
print(2, torch.arange(5) * (torch.arange(5) / 1))
print(3, torch.arange(5) + (torch.arange(5) / 1))
print(4, (torch.arange(5) >= 3))
print(5, 2 + (torch.arange(5)>=3) )
```

```
1 tensor([ 0,  1,  4,  9, 16])
2 tensor([ 0.,  1.,  4.,  9., 16.])
3 tensor([0., 2., 4., 6., 8.])
4 tensor([False, False, False,  True,  True])
5 tensor([2, 2, 2, 3, 3])
```

[8]:
```
# Not all types support all operations
try:
    #following does manual type conversion
    print(1, torch.arange(5).type(torch.float32).exp())
    print(2, torch.arange(5).exp()) #wow this has changed over time!
except RuntimeError as E:
    print(f"Got exception: '{E}'")
```

```
1 tensor([ 1.0000,  2.7183,  7.3891, 20.0855, 54.5981])
2 tensor([ 1.0000,  2.7183,  7.3891, 20.0855, 54.5981])
```

[9]:
```
# Here are some basic operations on matrix shapes
ns = torch.arange(12)
```

```
print(1, ns)
print(2, ns.reshape(3,-1))
# .view() is similar to .reshape() but reuses storage;
# we'll revisit it later.
print(3, ns.view(3,-1))
```

```
1 tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
2 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
3 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

[10]:
```
# We can also reshape into 3 axes
print(1, ns.reshape(2,2,3))
# .reshape() and .view() also understand "-1" which means
# "choose the appropriate size so that this works out".
print(2, ns.reshape(-1,2,3).shape)
```

```
1 tensor([[[ 0,  1,  2],
         [ 3,  4,  5]],

        [[ 6,  7,  8],
         [ 9, 10, 11]]])
2 torch.Size([2, 2, 3])
```

[11]:
```
# torch also has "0 axis" (or "0 order") tensors/"arrays".
# these are convenient because they still support .exp(), etc.
e = torch.tensor(2.7182818, dtype = torch.float32)
print(1, e, e.shape, e.log(), e.sin())
# method .item() extracts a python number
print(2, e, e.item())
# lol:
print(3, torch.tensor(2.7182818, dtype = torch.float64).item())
try:
    # .item() only works on single-element tensors
    print(4, torch.zeros(1,1,1,1).item())
    print(5, torch.zeros(2).item())
except ValueError as E:
    print(6, f"Got exception: '{E}'")
```

```
1 tensor(2.7183) torch.Size([]) tensor(1.) tensor(0.4108)
2 tensor(2.7183) 2.7182817459106445
3 2.7182818
4 0.0
6 Got exception: 'only one element tensors can be converted to Python scalars'
```

[12]:
```
# back to larger tensors,
# _some_ (but not all) operations complain about size mismatch.
try:
    vs = torch.arange(6).reshape(2,3)
    print(1, vs + vs)
    print(2, vs + vs.T)
except RuntimeError as E:
    print(3, f"Got exception: '{E}'")
```

```
1 tensor([[ 0,  2,  4],
        [ 6,  8, 10]])
3 Got exception: 'The size of tensor a (3) must match the size of tensor b (2)
at non-singleton dimension 1'
```

[13]:
```
# but some operations _do_ succeed with mismatched shapes!
vs = torch.arange(4)
print(0, vs)
print(0.5, vs.reshape(1, -1).shape)
print(1, vs.reshape(1, -1) + vs.reshape(-1, 1))
print(2, vs.reshape(1, -1, 1) + vs.reshape(-1, 1, 1) + vs.reshape(1, 1, -1))
```

```
0 tensor([0, 1, 2, 3])
0.5 torch.Size([1, 4])
1 tensor([[0, 1, 2, 3],
        [1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6]])
2 tensor([[[0, 1, 2, 3],
         [1, 2, 3, 4],
         [2, 3, 4, 5],
         [3, 4, 5, 6]],

        [[1, 2, 3, 4],
         [2, 3, 4, 5],
         [3, 4, 5, 6],
         [4, 5, 6, 7]],

        [[2, 3, 4, 5],
         [3, 4, 5, 6],
         [4, 5, 6, 7],
         [5, 6, 7, 8]],

        [[3, 4, 5, 6],
         [4, 5, 6, 7],
         [5, 6, 7, 8],
         [6, 7, 8, 9]]])
```

[14]:
```
# here's a simpler instance of the same behavior:
torch.zeros(4, 4) + torch.arange(4).reshape(-1, 1)
```

[14]:
```
tensor([[0., 0., 0., 0.],
        [1., 1., 1., 1.],
        [2., 2., 2., 2.],
        [3., 3., 3., 3.]])
```

[15]:
```
# This can be very convenient:
# here we normalize the rows of a matrix:
X = torch.randn(3,2)
print(1, X.norm(dim = 1))
# A few things going on here, I recommend trying this one
# yourself and studying each piece.
X /= X.norm(dim = 1, keepdim = True)
print(2, X.norm(dim = 1))
```

```
1 tensor([1.5687, 2.2517, 1.7698])
2 tensor([1.0000, 1.0000, 1.0000])
```

[16]:
```
# slicing makes it easy to access submatrices/subtensors
ns = torch.arange(12).reshape(2,6)
print(1, ns)
print(2, ns[0, :]) # first row
print(3, ns[:, 0]) # first column
ms = ns.reshape(2,2,3)
print(4, ms)
print(5, ms[0,0,:])
print(6, ms[0, ...]) # dots mean "all remaining axes/dimensions"
print(7, ms[..., 0])
```

```
1 tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])
2 tensor([0, 1, 2, 3, 4, 5])
3 tensor([0, 6])
4 tensor([[[ 0,  1,  2],
         [ 3,  4,  5]],

        [[ 6,  7,  8],
         [ 9, 10, 11]]])
5 tensor([0, 1, 2])
6 tensor([[0, 1, 2],
        [3, 4, 5]])
7 tensor([[0, 3],
        [6, 9]])
```

[17]:
```
# Slicing can also take integer lists as input
print(1, ns[:, [0, 2, 4]])
# Also boolean masks
print(2, ns[:, [ True, False, True, False, True, False]])
try:
    # you can use pytorch integer arrays as well
    print(3, ns[:, torch.arange(2)])
    # but not float arrays
    print(4, ns[:, torch.arange(2) / 1])
except IndexError as E:
    print(5, f"Got exception: '{E}'")
```

```
1 tensor([[ 0,  2,  4],
        [ 6,  8, 10]])
2 tensor([[ 0,  2,  4],
        [ 6,  8, 10]])
3 tensor([[0, 1],
        [6, 7]])
5 Got exception: 'tensors used as indices must be long, byte or bool tensors'
```

[18]:
```
# many operations have in-place versions.
# superficially this is good for efficiency reasons.
# more importantly, pytorch does some internal book-keeping
# with autodifferentiation which is lost if you do not do
# in-place operations for variables you wish to compute
# gradients with respect to.
# (This will be clarified later.)
# For now, here are some example in-place operations
v = torch.randn(5,4)
print(1, v.norm())
v += v # in-place arithmetic operations
v *= 2
print(2, v.norm())
v.clamp_(0, float('inf')) # zero out negative values, in-place
print(3, v.norm())
```

```
1 tensor(3.7304)
2 tensor(14.9216)
3 tensor(11.3672)
```

[19]:
```
# this step only matters if you have a gpu.
# this line of code is in my pytorch programs, it means
# "variable 'device' is first gpu if available, else cpu".
device = torch.device("cpu" if not torch.cuda.is_available()
                      else "cuda:0")
# that didn't put anything on gpu; we manually move things there
ns = torch.arange(4)
```

```
ns2 = ns.to(device)
print(1, ns.device, ns2.device)

try:
    # python disallows operations mixing cpu and gpu;
    # this is good, since moving data between them is expensive.
    ns + ns2
    print(2, "no exception: no gpu in use")
except Exception as E:
    print(3, f"pytorch error: {E}")
```

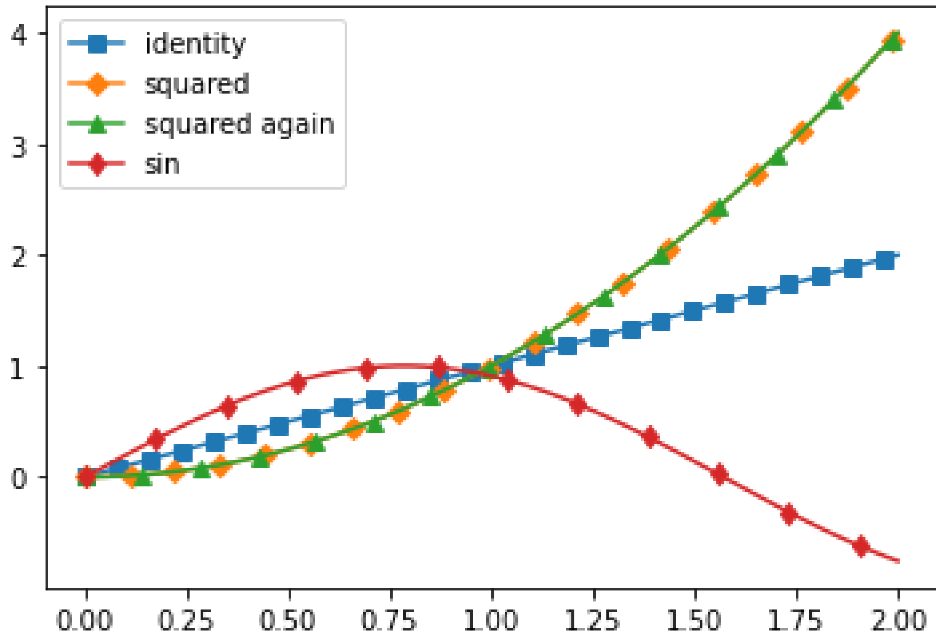
```
1 cpu cpu
2 no exception: no gpu in use
```

2 matplotlib plotting

[20]:
```
import matplotlib.pyplot as plt
```

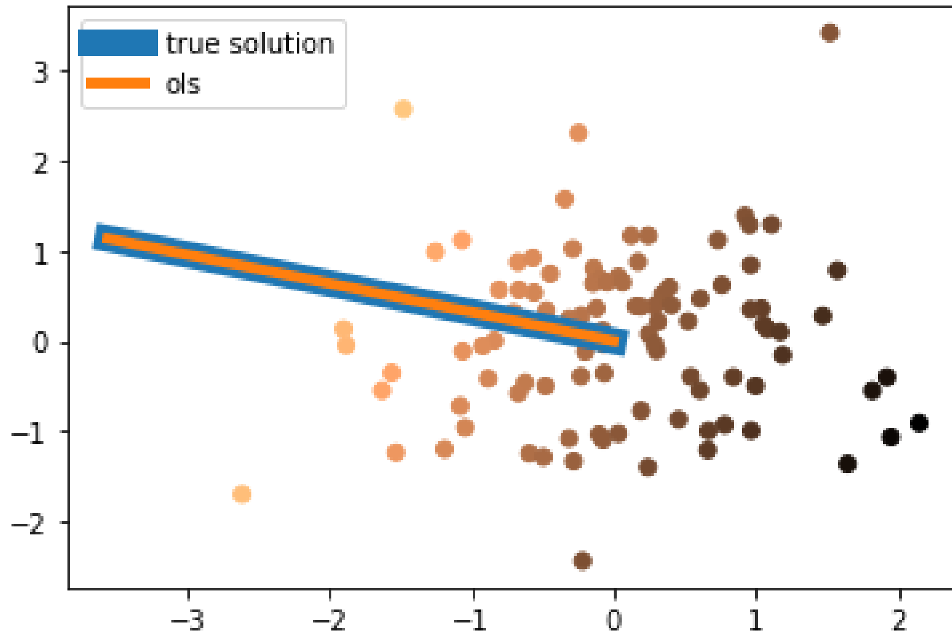
[21]:
```
# plt.plot() lets you display many curves.
# it has many parameters; in jupyter and ipythong, you can execute
# "plt.plot?" to see some of them.
# note: gpu data must be moved to cpu before being passed to matplotlib
xs = torch.linspace(0, 2, 128)
plt.plot(xs, xs, marker = 's', markevery = 5,
         label = "identity")
plt.plot(xs, xs ** 2, marker = 'D', markevery = 7,
         label = 'squared')
plt.plot(xs, xs * xs, marker = '^', markevery = 9,
         label = 'squared again')
plt.plot(xs, (2 * xs).sin(), marker = 'd', markevery = 11,
         label = 'sin')
plt.legend()
```

[21]:
```
<matplotlib.legend.Legend at 0x7fb4e131c310>
```

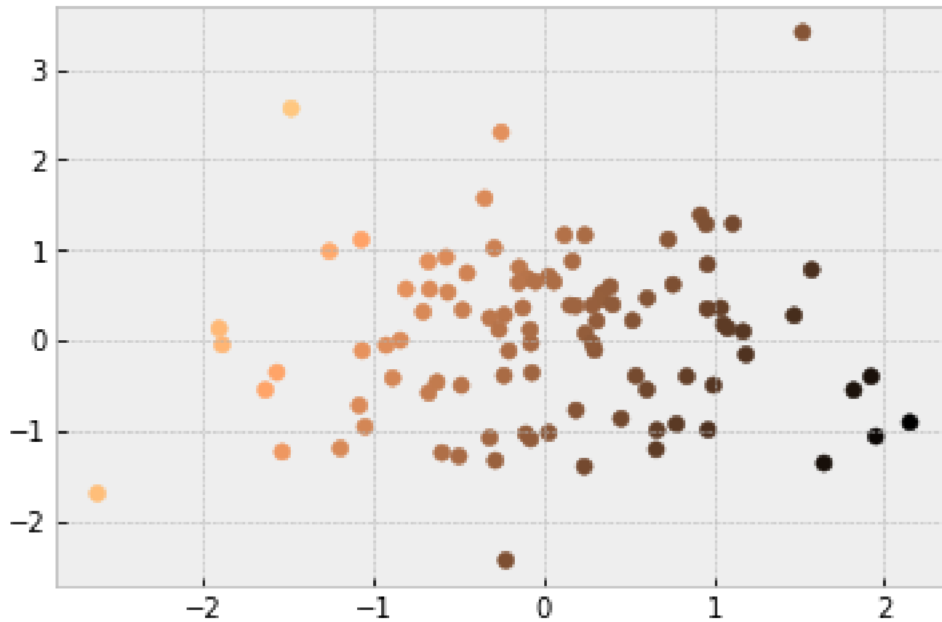


[22]:
```
# We can put scatterplots and curve plots in the same figure.
# if invoking this from a python script, you'd need to do
# plt.clf()
# to clear the figure after the previous one
# (and also plt.savefig() or plt.show() to display).
X = torch.randn(100, 2) # create some random data
u = torch.randn(2) # sample a random "correct" linear predictor
# pick a norm for u that has easy visualization:
u *= X.norm(dim = 1).max() / u.norm()
y = X @ u # label data according to the "planted" predictor
# scatterplot of data, y given by color:
plt.scatter(
    X[:, 0],
    X[:, 1],
    # color according to y:
    c = (y - y.min()) / (y.max() - y.min()),
    cmap = "copper",
)
# note that these plots are the weight vectors, not decision boundary
plt.plot([0, u[0]], [0, u[1]], lw = 10, label = "true solution")
ols = X.pinverse() @ y
plt.plot([0, ols[0]], [0, ols[1]], lw = 4, label = "ols")
plt.legend()
```

[22]:
```
<matplotlib.legend.Legend at 0x7fb4e12381f0>
```



[23]:
```
# matplotlib has many features; here's a cute one to restyle a plot:
with plt.style.context("bmh"):
    plt.scatter(
        X[:, 0],
        X[:, 1],
        # color according to y:
        c = (y - y.min()) / (y.max() - y.min()),
        cmap = "copper",
    )
```



3 autodifferentiation

[24]:
```
# let's go back to the scatterplot from before:
print(X.shape, u.shape, y.shape)
# we'll find another with "manual" gradient descent
v = torch.zeros(2)
# another with automatic gradient computation
w = torch.zeros(2, requires_grad = True)
# another with automatic gradient computation
# but also using torch.optim
z = torch.zeros(2, requires_grad = True)
# this "requires_grad = True" means
# "whenever this object appears in expressions, track how it is used,
# so we can compute gradients later"
print(v)
print(w)
# hey matus, you know, no one understands your contour plot function
```

```
torch.Size([100, 2]) torch.Size([2]) torch.Size([100])
tensor([0., 0.])
tensor([0., 0.], requires_grad=True)
```

[25]:
```
# Can directly modify requires_grad
q = torch.randn(5,5)
print(1, q.requires_grad)
```

```
q.requires_grad = True
print(2, q.requires_grad)
q.requires_grad_(False)
print(3, q.requires_grad)
```

```
1 False
2 True
3 False
```

[26]:
```
print(1, torch.randn(5,5) @ torch.randn(5, requires_grad = True))
print(2, torch.randn(5,5) @ torch.randn(5))
```

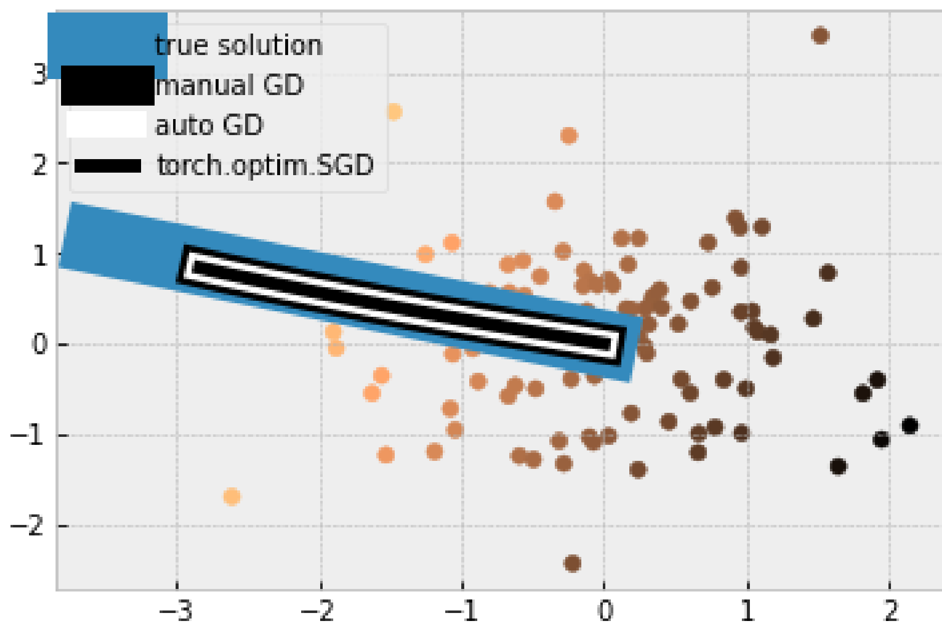
```
1 tensor([ 2.9433,  0.0733, -1.6359,  2.6543,  1.2616], grad_fn=<MvBackward>)
2 tensor([-3.7747, -2.0741,  2.0844,  2.6925,  3.1590])
```

[27]:
```
# now let's do some iterations of gradient descent on w and v
stepsize = 0.1 # a small one to help visualize
# we'll use pytorch's sgd for z:
z_optimizer = torch.optim.SGD([z], lr = stepsize)
# XXX note to future matus: gotcha about not modifying z directly
n_iters = 20
V = torch.empty(n_iters, 2)
W = torch.empty(n_iters, 2)
Z = torch.empty(n_iters, 2)
for i in range(n_iters):
    # let's save all iterates to plot them later
    V[i, :] = v
    # for w and z, if we copy them in like v, then
    # W also gets requires_grad enabled.
    # We can use .detach() to disconnect from the computation graph
    W[i, :] = w.detach()
    Z[i, :] = z.detach()

    # manual gradient computation on v:
    v_risk = ((X @ v - y) ** 2 / 2).mean()
    # take a gradient step:
    v -= stepsize * X.T @ (X @ v - y) / X.shape[0]

    # automatic gradient computation on w:
    w_risk = ((X @ w - y) ** 2 / 2).mean()
    # following line means
    # "go through the computation of 'w_risk',
    # and save gradient information for
    # tensors with all requires_grad=True"
    w_risk.backward()
    # we still need _use_ the saved gradient information
```

```
    with torch.no_grad():
        # we will do that inside a torch.no_grad() block.
        # this  means "here we do not track gradient computations".
        # this particular gradient computation blows up in various
        # ways without the block.
        # for v we were fine, there are no requires_grad variables.
        w -= stepsize * w.grad
        # w.grad is where gradient information was stored.
        # we must explicitly clear it, or else it will be combined
        # with future iteration gradient information
        w.grad.zero_()
        # the .zero_() means "zero this out in place".
        #w = w - stepsize * w.grad

    # now let's do z
    z_optimizer.zero_grad() # the optimizer handles this now
    # rest is familiar from w:
    z_risk = ((X @ z - y) ** 2 / 2).mean()
    z_risk.backward()
    # now do the update.  torch.no_grad() is unnecessary
    # because .step() invokes it internally.
    z_optimizer.step() # thanks, pytorch

    # lastly let's print the empirical risk of all
    with torch.no_grad():
        print(f"iter {i}"
              f" risk {v_risk:.3g} {w_risk:.3g} {z_risk:.3g}")

with plt.style.context("bmh"):
    plt.scatter(
        X[:, 0],
        X[:, 1],
        # color according to y:
        c = (y - y.min()) / (y.max() - y.min()),
        cmap = "copper",
    )
    plt.plot([0, u[0]], [0, u[1]], lw = 25, label = "true solution")
    plt.plot(V[:, 0], V[:, 1],
             lw = 15, color = 'black',
             label = "manual GD")
    plt.plot(W[:, 0], W[:, 1],
             lw = 10, color = 'white',
             label = "auto GD")
    plt.plot(Z[:, 0], Z[:, 1],
             lw = 5, color = 'black',
             label = "torch.optim.SGD")
```

```
    plt.legend()
```

```
iter 0 risk 5.56 5.56 5.56
iter 1 risk 4.71 4.71 4.71
iter 2 risk 3.98 3.98 3.98
iter 3 risk 3.37 3.37 3.37
iter 4 risk 2.86 2.86 2.86
iter 5 risk 2.42 2.42 2.42
iter 6 risk 2.05 2.05 2.05
iter 7 risk 1.73 1.73 1.73
iter 8 risk 1.47 1.47 1.47
iter 9 risk 1.24 1.24 1.24
iter 10 risk 1.05 1.05 1.05
iter 11 risk 0.893 0.893 0.893
iter 12 risk 0.757 0.757 0.757
iter 13 risk 0.641 0.641 0.641
iter 14 risk 0.543 0.543 0.543
iter 15 risk 0.461 0.461 0.461
iter 16 risk 0.39 0.39 0.39
iter 17 risk 0.331 0.331 0.331
iter 18 risk 0.28 0.28 0.28
iter 19 risk 0.238 0.238 0.238
```



[28]:
```
# Let's take a moment to study the last w_risk
w_risk
```

[28]: tensor(0.2377, grad_fn=<MeanBackward0>)

[29]:
```
# the "grad_fn" is also part of the computation tracking.
# if we use .detach() here, similarly it clears this.
w_risk.detach()
```

[29]: tensor(0.2377)

[30]:
```
w = torch.randn(())
print(1, w, w.shape)
print(2, w.detach(), w.detach().clone(), w.item())
print(3, w.view(-1,1,1))
```

```
1 tensor(-0.4200) torch.Size([])
2 tensor(-0.4200) tensor(-0.4200) -0.4200473725795746
3 tensor([[[-0.4200]]])
```

[31]:
```
# here's an example of what goes wrong within torch.no_grad.
w2 = torch.randn(2, requires_grad = True)
risk = ((X @ w2 - y) ** 2 / 2).mean()
risk.backward()
with torch.no_grad():
    # WRONG WAY: assign to a temporary variable
    w3 = w2 - stepsize * w2.grad
    # RIGHT WAY: in place operations.
    w2 -= stepsize * w2.grad
    # same goes with other operations like torch.clamp(), etc.
    # notice that one of the following changes
    print(w3.requires_grad, w2.requires_grad)
```

```
False True
```

[32]:
```
# note that "autodifferentiation" doesn't require differentiability.
w = torch.zeros((), requires_grad = True) # scalar zero
nondiff = torch.nn.functional.relu(w)
nondiff.backward()
# relu is not differentiable at zero...
print(w.grad)
# relu has clarke differential (and subdifferential) of [0,1] at 1.
# so anything within [0,1] seems reasonable.
```

```
tensor(0.)
```

[33]:
```
# good job, pytorch, how about this one.
w.grad.zero_() # first zero out old gradient
relu = torch.nn.functional.relu # shorthand
tricky = relu(w) - relu(-w) # identity map
tricky.backward()
```

```
print(w.grad) # 1 is the only correct value
```

```
tensor(0.)
```

4 single-layer networks

[34]:
```
# a basic fully connected layer; randomly initialized
fc1 = torch.nn.Linear(5,4, bias = True)
# another one, different random init:
fc2 = torch.nn.Linear(5,4, bias = True)

# let's apply these layers to some data
x = torch.randn(5)
# you can call them like functions.  result nonzero due to random init.
print((fc1(x) - fc2(x)).norm())
```

```
tensor(2.2624, grad_fn=<CopyBackwards>)
```

[35]:
```
# A layer is itself a subclass of the general torch network class
print(1, isinstance(fc1, torch.nn.Module))
# this class contains main convenient operations.
# here are two ways to apply them to data:
print(2, (fc1(x) - fc1.forward(x)).norm())
# note that .forward()'s name matches with .backward(),
# corresponding to backpropagation.
# here we see we can print networks, useful for debugging:
print(3, fc1)
```

```
1 True
2 tensor(0., grad_fn=<CopyBackwards>)
3 Linear(in_features=5, out_features=4, bias=True)
```

[36]:
```
# torch.nn.Module instances can iterate over parameters.
# most often we use this to define gradient descent
for P in fc1.parameters():
    print(P.shape)
```

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

[37]:
```
# what about with no bias?
for P in torch.nn.Linear(5,4, bias = False).parameters():
    print(P.shape)
```

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

[38]:
```
# this zeros out gradients.
fc1.zero_grad()
# it is like accessing the weights (and biases!) within fc1
# and calling .zero_().
```

[39]:
```
fc = torch.nn.Linear(5, 1)
# torch networks can take minibatches directly as input;
# now the inputs are written as rows.
X = torch.randn(10, 5)
print(1, fc(X[0, :]).shape) # column vector in, singleton vector out
print(2, fc(X).shape) # matrix in, _matrix_ out
print(3, fc(X[:5, :]).shape) # now with a minibatch
```

```
1 torch.Size([1])
2 torch.Size([10, 1])
3 torch.Size([5, 1])
```

[40]:
```
# Typically pytorch code does not directly extract minibatches from
# a big data tensor, but uses wrappers from torch.utils.data
nb = 32
n = 256
# following wraps inputs and outputs into single object
data = torch.utils.data.TensorDataset(
    torch.arange(n).type(torch.float32), # our fake input data
    # using sequential data to tell apart shuffle and not
    torch.randint(0, 10, (n,)) # our labels
)
for shuffle in [ False, True ]:
    # DataLoader handles minibatching
    loader = torch.utils.data.DataLoader(data, batch_size = nb,
                    shuffle = shuffle, num_workers = 1)
    # loader exposes an iterable interface:
    for (i, (Xb, yb)) in enumerate(loader):
        print(f"shuffle {shuffle} {i} {Xb.min() / nb:.3g}")
```

```
shuffle False 0 0
shuffle False 1 1
shuffle False 2 2
shuffle False 3 3
shuffle False 4 4
shuffle False 5 5
shuffle False 6 6
shuffle False 7 7
shuffle True 0 0.188
shuffle True 1 0.0312
shuffle True 2 0.438
shuffle True 3 0.219
shuffle True 4 0
```

```
shuffle True 5 0.125
shuffle True 6 0.0625
shuffle True 7 0.0938
```

[41]:
```
# here's a gotcha!
# for linear logistic regression, we did y * (X @ w).  now:
y = torch.randn(X.shape[0])
print((y * fc(X)).shape) # OOPS
print((y * fc(X).view(-1)).shape) # correct...
```

```
torch.Size([10, 10])
torch.Size([10])
```

[42]:
```
# We can also move entire networks to gpu with one function call
# (in this case, it moves the weights and the biases).
fc.to(device)
```

[42]:
```
Linear(in_features=5, out_features=1, bias=True)
```

[43]:
```
lin = torch.nn.Linear(4,5)
lin(torch.randn(3,4))
```

[43]:
```
tensor([[ 0.0391,  0.2412, -0.1293, -0.7226,  0.1816],
        [ 0.0357, -0.0997,  0.1184,  0.4253, -0.2658],
        [-0.2063,  0.1042,  0.2319,  0.0209,  0.0116]],
       grad_fn=<AddmmBackward>)
```

[44]:
```
# We can also create convolutional layers easily
conv = torch.nn.Conv2d(5, 4, 2)
# example random data with 10 6x6 images using 5 channels:
X = torch.randn(10, 5, 6, 6)
print(1, conv(X[:1, ...]).shape) #output on first example
print(2, conv(X).shape) #output on whole batch
try:
    # unfortunately, unlike for linear layers,
    # we _must_ use inputs with 4 axes of input
    print(3, conv(X[0, ...]).shape)
except Exception as E:
    print(4, f"Got exception: '{E}'")
# just a sanity check:
# indeed convolutional layers, unlike linear layers,
# can handle different choices of input width x height
print(5, conv(torch.randn(10, 5, 10, 10)).shape)
```

```
1 torch.Size([1, 4, 5, 5])
2 torch.Size([10, 4, 5, 5])
4 Got exception: 'Expected 4-dimensional input for 4-dimensional weight [4, 5,
2, 2], but got 3-dimensional input of size [5, 6, 6] instead'
```

```
5 torch.Size([10, 4, 9, 9])
```

[45]:
```
# convolutional layers also have biases on by default!
for P in conv.parameters():
    print(P.shape)
```

```
torch.Size([4, 5, 2, 2])
torch.Size([4])
```

[46]:
```
# activations also subclass torch.nn.Module
relu = torch.nn.ReLU()
print(1, relu)
print(2, len(list(relu.parameters())))
v = torch.linspace(-3, 3, 7)
print(3, relu(v))
# many layer types can also be invoked "functionally",
# without creating a layer object.
print(4, torch.nn.functional.relu(v))
# we can also call relu directly on tensors.
print(5, v.relu())
```

```
1 ReLU()
2 0
3 tensor([0., 0., 0., 0., 1., 2., 3.])
4 tensor([0., 0., 0., 0., 1., 2., 3.])
5 tensor([0., 0., 0., 0., 1., 2., 3.])
```

[47]:
```
# We also have softmax layers...
v = torch.randn(4)
softmax = torch.nn.Softmax(dim = 0)
print(1, softmax(v))
print(2, torch.nn.functional.softmax(v, dim = 0))
print(3, v.softmax(dim = 0))
# let's sanity check
print(4, v.exp() / v.exp().sum())
```

```
1 tensor([0.3339, 0.4058, 0.1528, 0.1075])
2 tensor([0.3339, 0.4058, 0.1528, 0.1075])
3 tensor([0.3339, 0.4058, 0.1528, 0.1075])
4 tensor([0.3339, 0.4058, 0.1528, 0.1075])
```

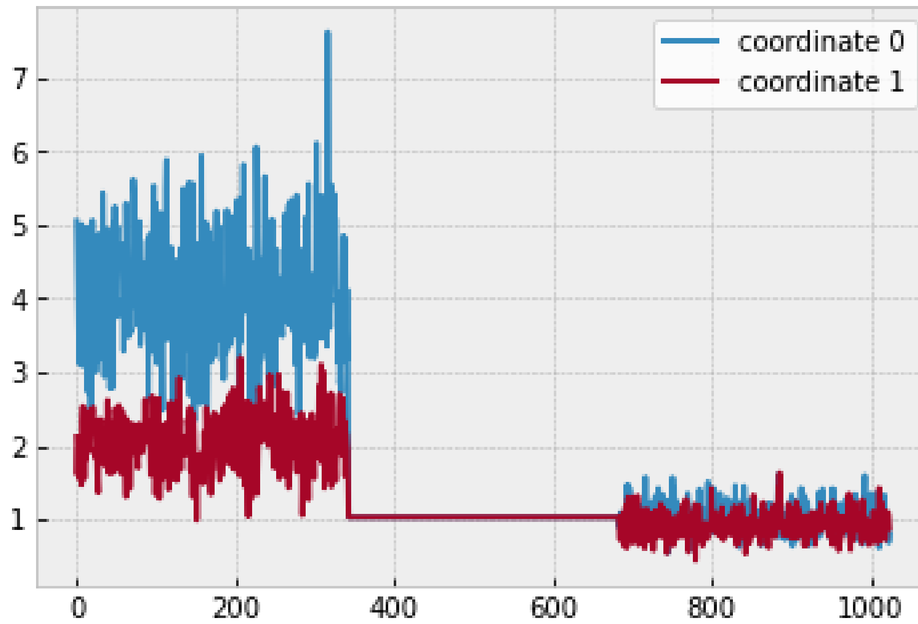
[48]:
```
# Let's further sanity check softmax with cross entropy
yhat = v.view(1, -1)
y = torch.ones(1,).type(torch.long)
print(1, torch.nn.functional.cross_entropy(yhat, y))
# It also exists as a layer!
risk = torch.nn.CrossEntropyLoss()
assert(len(list(risk.parameters())) == 0)
```

```
print(2, risk(yhat, y))
# variable name "risk" because averages batches:
yhat = torch.randn(10, 5)
y = torch.randint(0, yhat.shape[1], (yhat.shape[0],))
risk1 = risk(yhat, y)
assert(len(risk1.shape) == 0)
# lastly let's check the computation manually.
# pytorch has .logsumexp() for numerical reasons;
# following slicing has a tiny gotcha, can't use : in place of range.
risk2 = (- yhat[torch.arange(10), y] + yhat.logsumexp(dim = 1)).mean()
# another way
risk3 = -yhat.softmax(dim = 1)[torch.arange(10), y].log().mean()
assert((risk2 - risk1).abs().item() < 1e-6 and
      (risk3 - risk2).abs().item() < 1e-6)
```

```
1 tensor(0.9020)
2 tensor(0.9020)
```

[49]:
```
# Lastly, let's study batch norm a little bit.
# Let's play with the .train() and eval() routines,
# and also see if we can observe the normalization.
d = 2
bn = torch.nn.BatchNorm1d(d)
bn.eval() # disable tracking of statistics
# Gaussian data, axis aligned, different variances.
X = torch.randn(1024, d) @ torch.tensor([[4.0, 0,], [0, 2]])
n_iters = 1024
stddevs = torch.empty(n_iters, d)
import random
for iters in range(n_iters):
    x_mb = X[random.sample(range(X.shape[0]), 16), :]
    out = bn(x_mb)
    with torch.no_grad():
        if iters == n_iters // 3:
            bn.train()
        elif iters == 2 * n_iters // 3:
            bn.eval()
        stddevs[iters, :] = out.std(dim = 0)
with plt.style.context('bmh'):
    for j in range(d):
        plt.plot(range(n_iters), stddevs[:, j],
                 label = f"coordinate {j}")
plt.legend()
```

[49]: <matplotlib.legend.Legend at 0x7fb4de84d1f0>



[50]:
```
# note that batch norm has those "affine" parameters by default.
# we did not do any updates do them, so they're still default.
for P in bn.parameters():
    print(P.data)
```

```
tensor([1., 1.])
tensor([0., 0.])
```

5 multi-layer networks

[51]:
```
# now let's work with multi-layer networks.
# for networks that just stack standard types of layers,
# here is an easy way:
net = torch.nn.Sequential(
    torch.nn.Linear(5,100),
    torch.nn.ReLU(),
   # torch.nn.Linear(100,1)
)
# note that "number layers" is already ambiguous and inconsistent
# across neural net conventions...

# we still have .forward(), function call, .zero_grad(),
#net(x)
net(torch.randn(5))
print([1,])
```

[1]

[52]:
```
print(net) # compare this with printing just torch.nn.Linear
```

```
Sequential(
  (0): Linear(in_features=5, out_features=100, bias=True)
  (1): ReLU()
)
```

[53]:
```
# since now we have multiple layers,
# .parameters() may be confusing for debugging purposes
# since each layer can have multiple parameters (e.g., due to bias).
# instead, we can use .named_parameters().
# here we can see the default names:
for (Pname, P) in net.named_parameters():
    print(Pname, P.shape)
```

```
0.weight torch.Size([100, 5])
0.bias torch.Size([100])
```

[54]:
```
# also convenient to define networks as classes.
class SquaredReLUNet(torch.nn.Module):
    def __init__(self, d, width):
        super(SquaredReLUNet, self).__init__() # boilerplate
        self.d = d
        self.width = width
        self.fc1 = torch.nn.Linear(d, width, bias = False)
        self.relu = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(width, 1, bias = False)
        # torch.nn.Module "sees" fc1 and fc2 and they are
        # accessed by operations like zero_grad(), parameters(), etc.
        # for more exotic architectures, you need to
        # manually register with self.add_module().

    def forward(self, x):
        x = self.fc1(x)
        # squared ReLU; more convenient than with torch.nn.Sequential
        x = self.relu(x) ** 2
        return self.fc2(x)

net = SquaredReLUNet(5, 128)
net(torch.randn(50, 5)).shape
```

[54]:
```
torch.Size([50, 1])
```

[55]:
```
net.zero_grad() # let's clear all gradient information.
(X, y) = (torch.randn(50, 5), torch.randn(50))
risk = ((net(X).view(-1) - y) ** 2).mean() / 2 # note ".view(-1)"
```

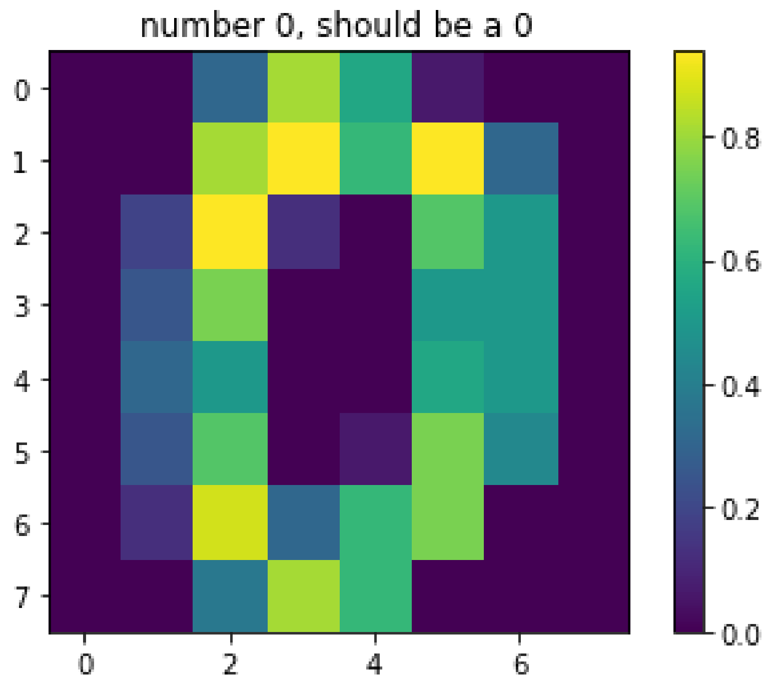
```
risk.backward()
with torch.no_grad():
     # they all get magically registered in the __init__
    for (Pi, P) in enumerate(net.parameters()):
        print(Pi, P.shape, P.grad.shape)
        P -= 0.01 * P.grad # in place operations !

    # it is possible that it doesn't decrease,
    # since the step size is fixed but the random data could be wild...
    print(f"risk should decrease: init {risk:.3g}, "
           f"one iter {(((net(X).view(-1) - y) ** 2).mean() / 2):.3g}")
```

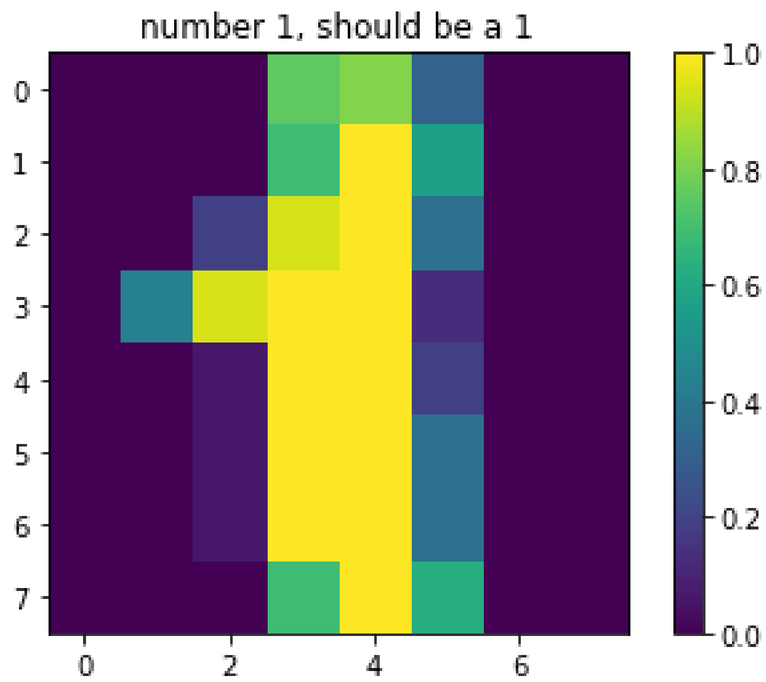
```
0 torch.Size([128, 5]) torch.Size([128, 5])
1 torch.Size([1, 128]) torch.Size([1, 128])
risk should decrease: init 0.547, one iter 0.54
```

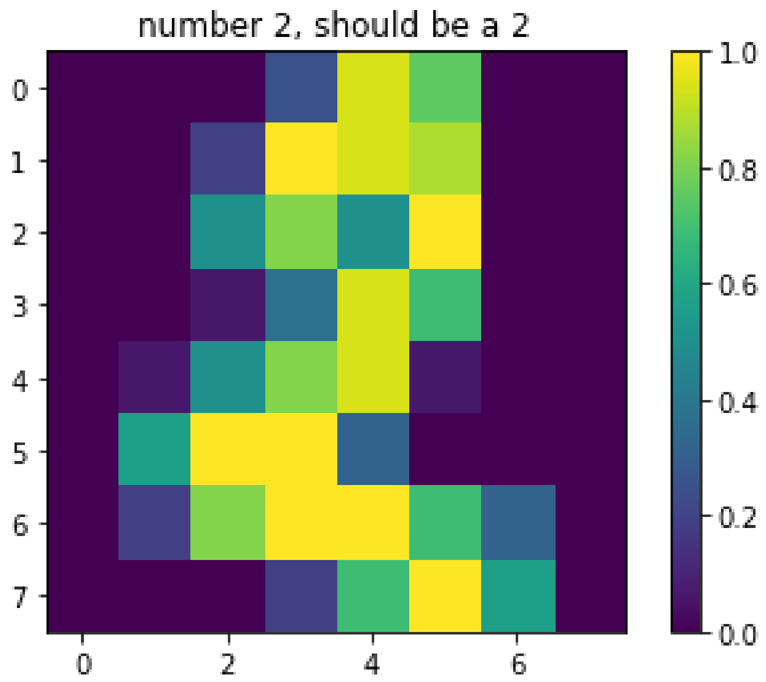
6 Digit experiment

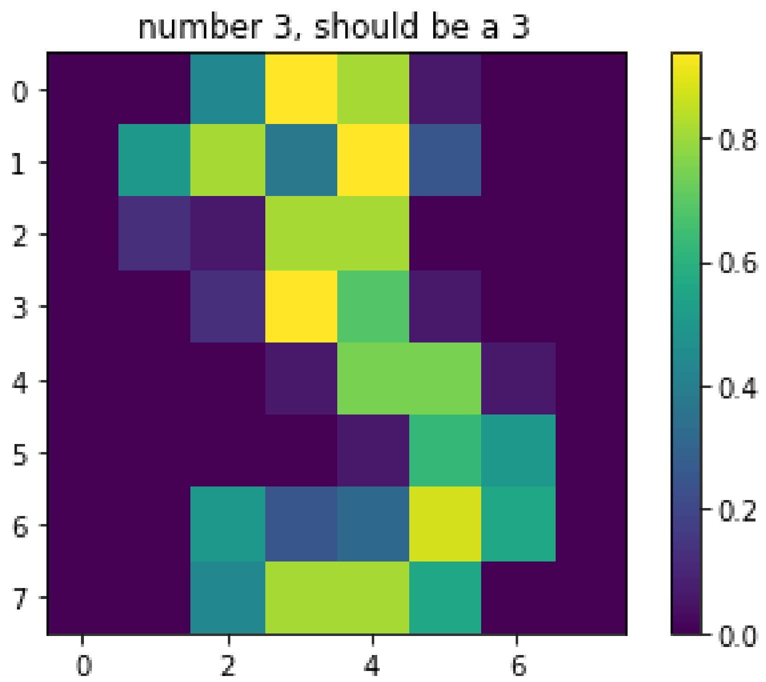
[56]:
```
import sklearn.datasets
sk_digits = sklearn.datasets.load_digits()
(X, Y) = (torch.tensor(sk_digits.data).type(torch.float), torch.
 ↪tensor(sk_digits.target))
print(X.shape, y.shape, X.max(), X.min())
X /= X.max()
for i in range(20):
    plt.figure(1 + i)
    ax = plt.imshow(X[i, ...].view(-1, 8))
    plt.colorbar(ax)
    plt.title(f"number {i}, should be a {Y[i]}")
```

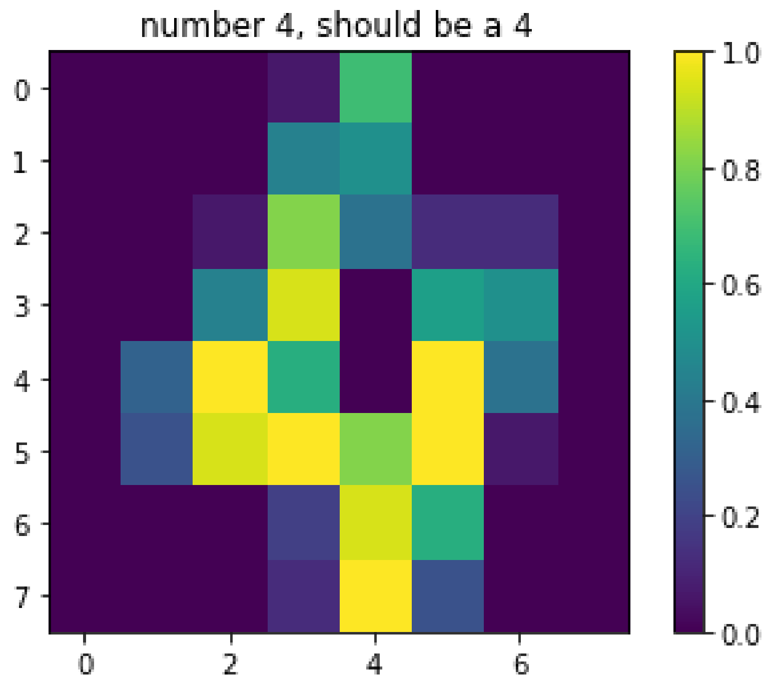
```
torch.Size([1797, 64]) torch.Size([50]) tensor(16.) tensor(0.)
```

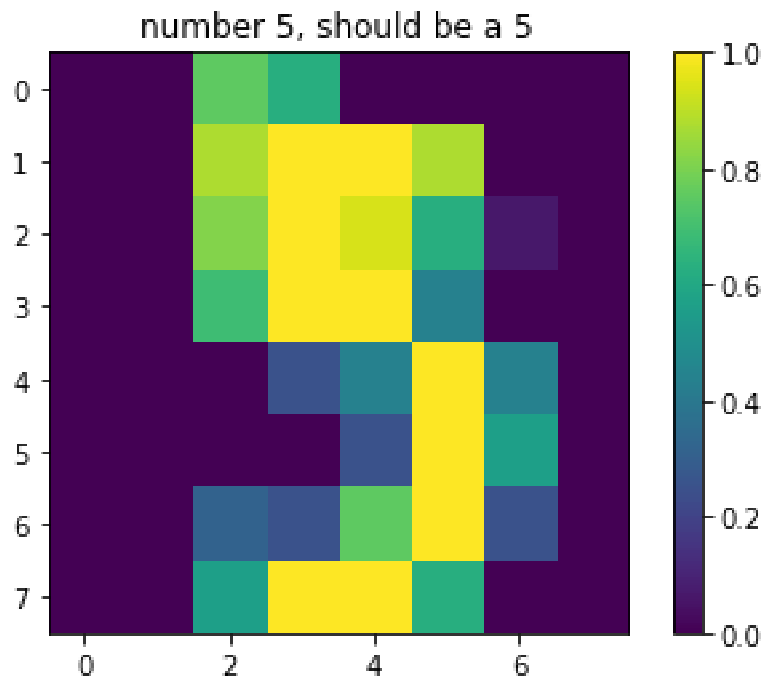

number 0, should be a 0
0
1
2
3
4
5
6
7
0
2
4
6
0.0
0.2
0.4
0.6
0.8

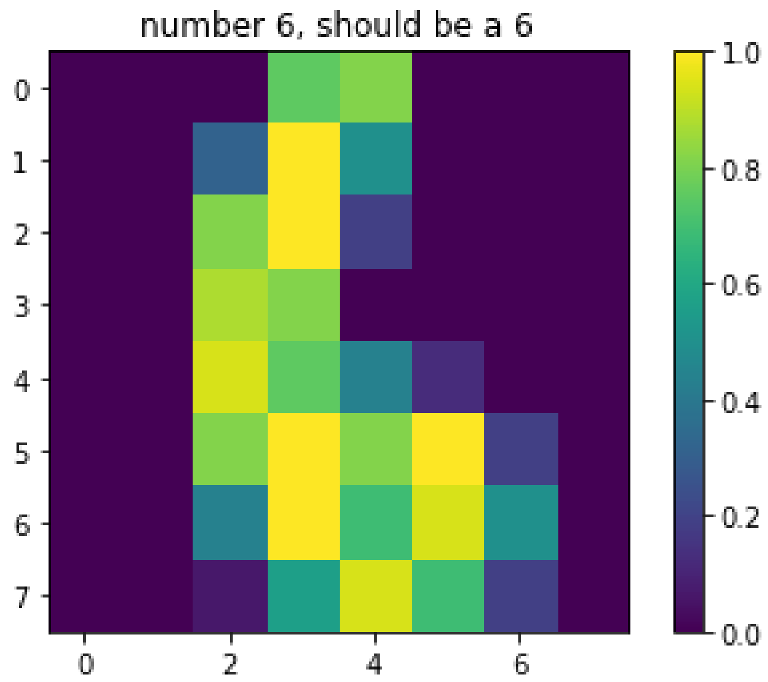

number 1, should be a 1
0
1
2
3
4
5
6
7
0
2
4
6
0.0
0.2
0.4
0.6
0.8
1.0

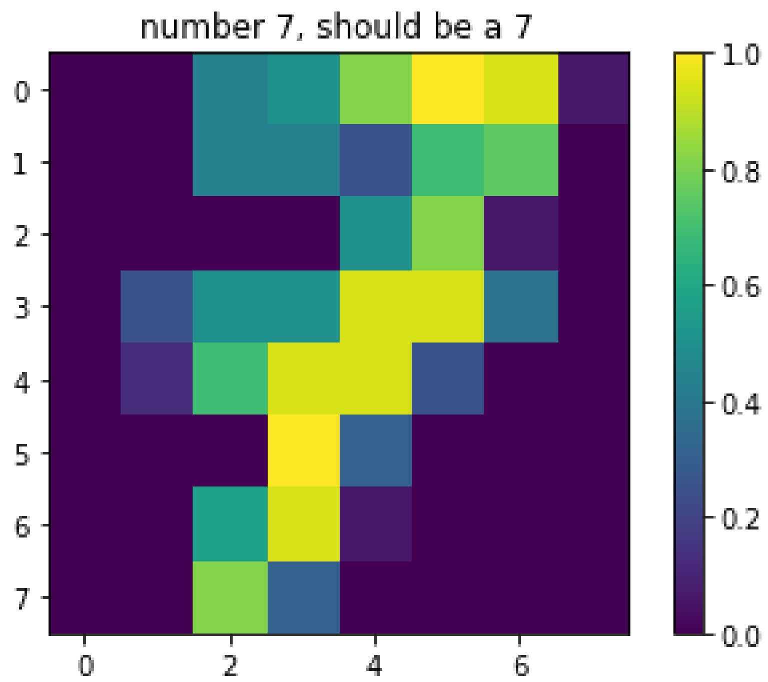

number 2, should be a 2
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

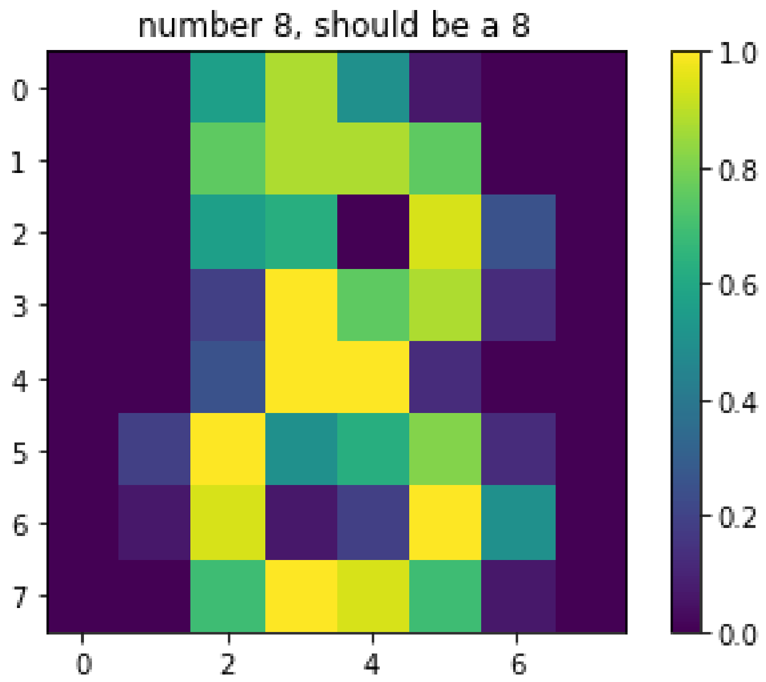

number 3, should be a 3
0
1
2
3
4
5
6
7
0
2
4
6
0.8
0.6
0.4
0.2
0.0

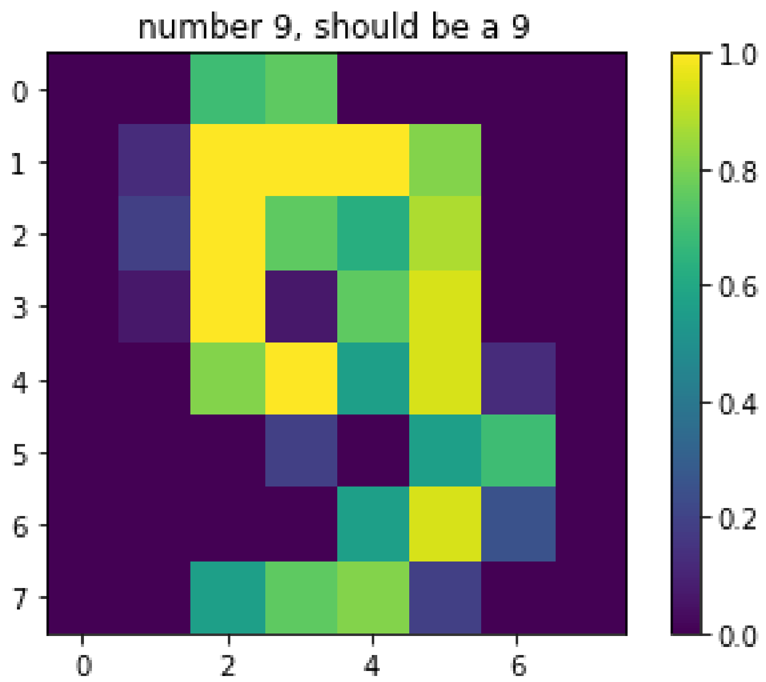

number 4, should be a 4
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

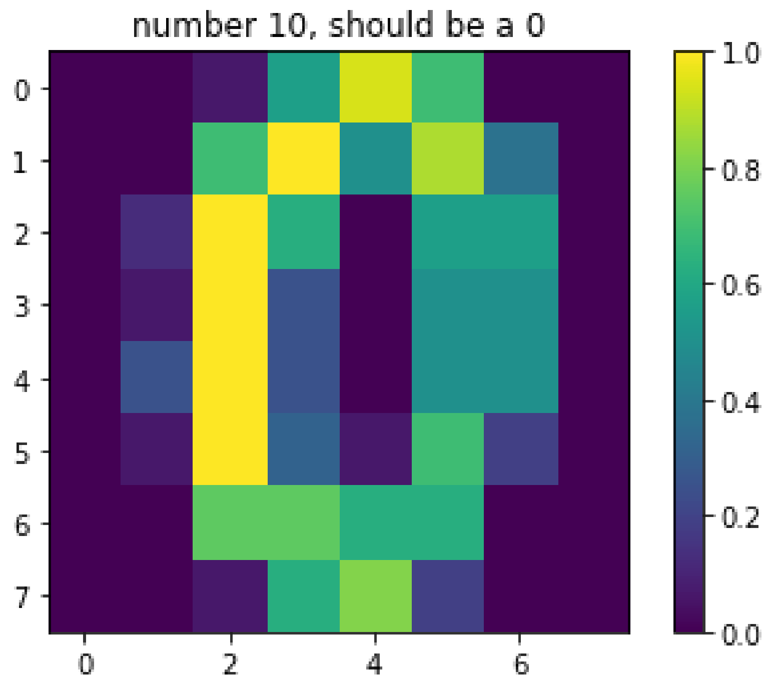

number 5, should be a 5
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

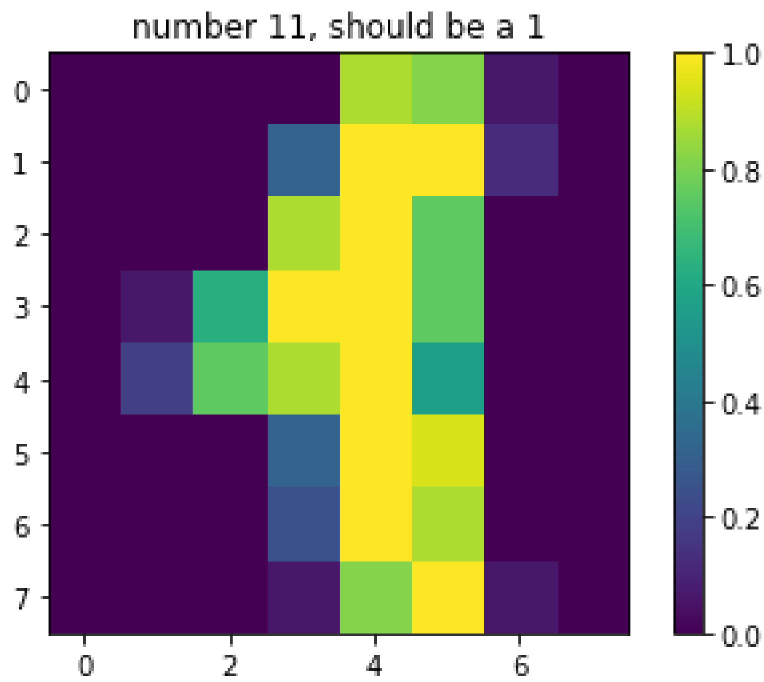

number 6, should be a 6
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

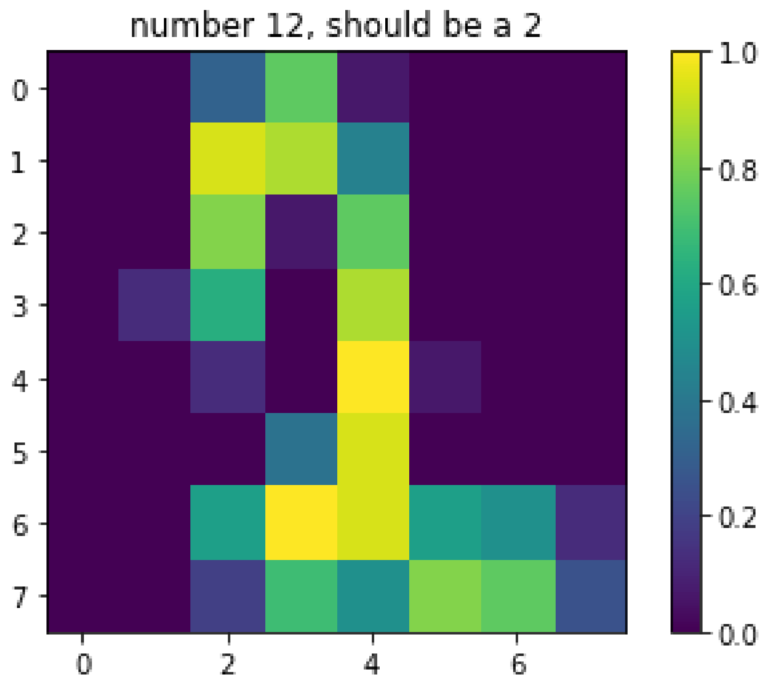

number 7, should be a 7
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

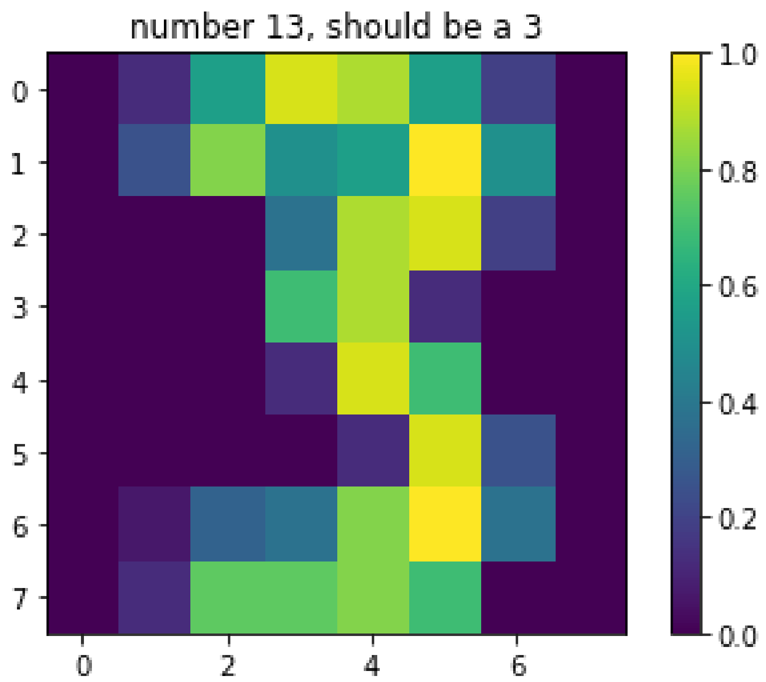

number 8, should be a 8
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

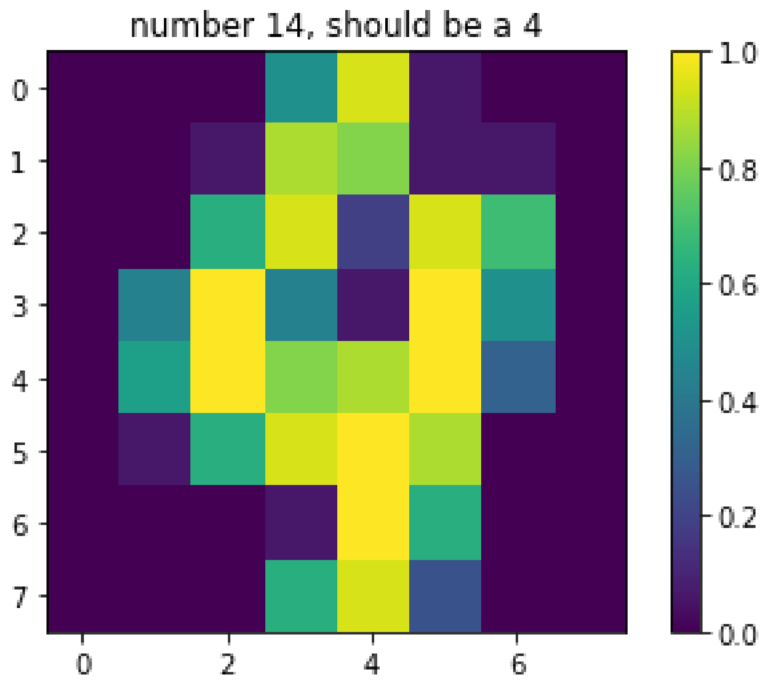

number 9, should be a 9
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

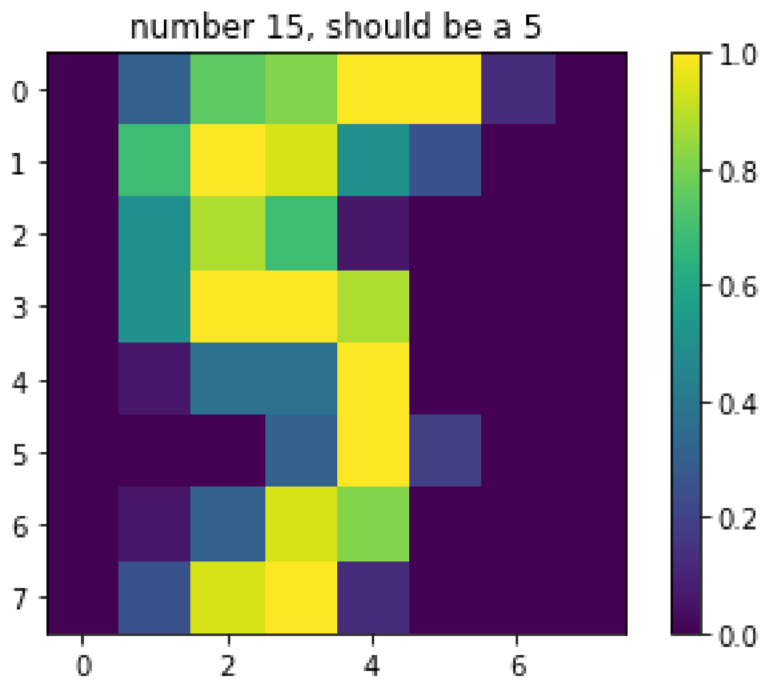

number 10, should be a 0
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

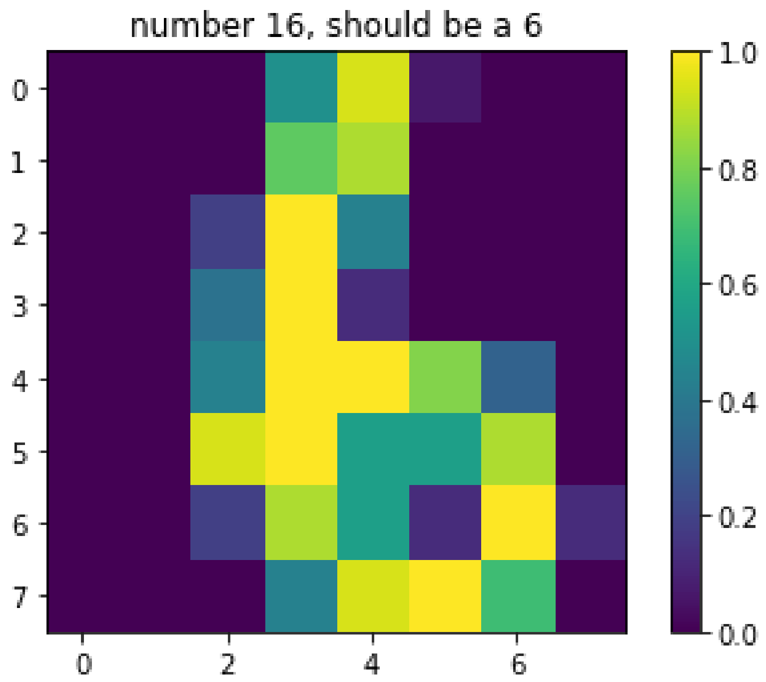

number 11, should be a 1
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

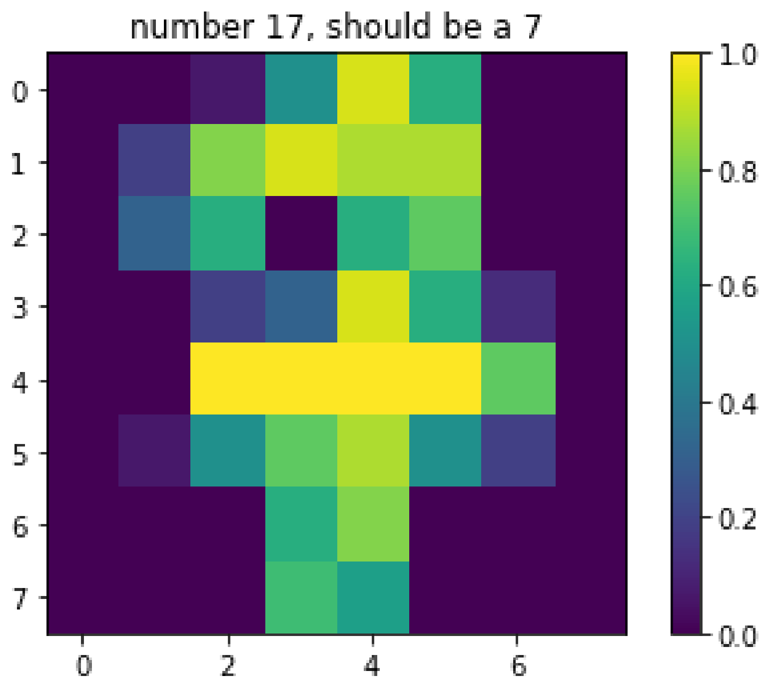

number 12, should be a 2
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

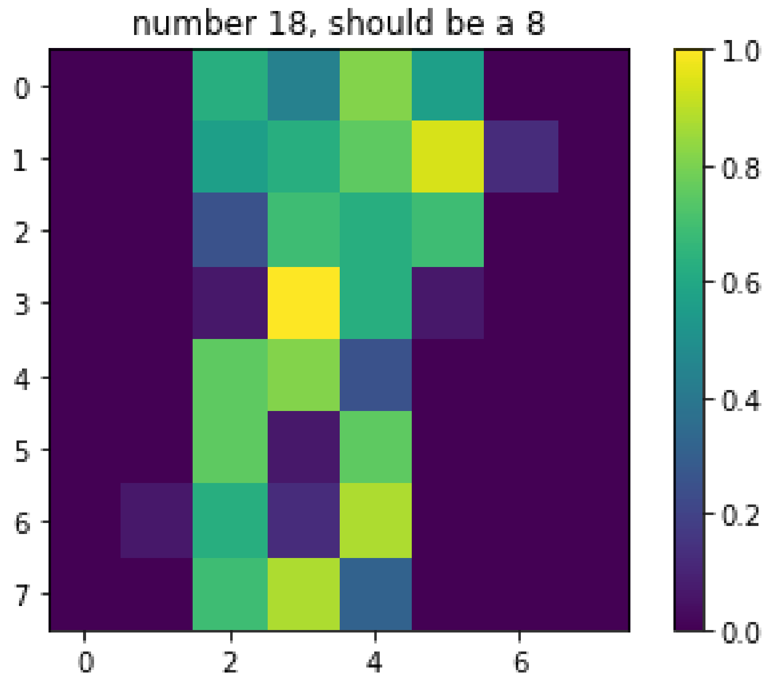

number 13, should be a 3
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0

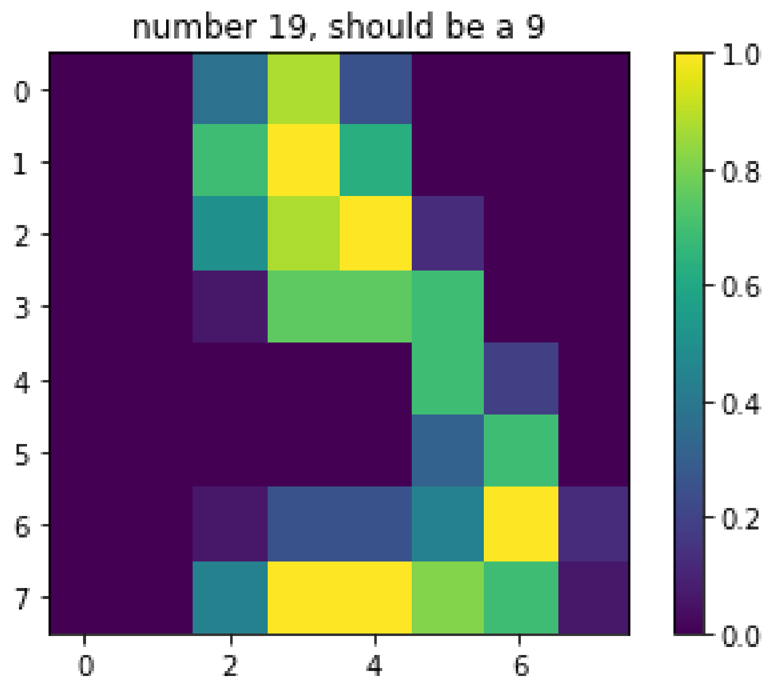

number 14, should be a 4
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0


number 15, should be a 5
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0


number 16, should be a 6
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0


number 17, should be a 7
0
1
2
3
4
5
6
7
0
2
4
6
1.0
0.8
0.6
0.4
0.2
0.0





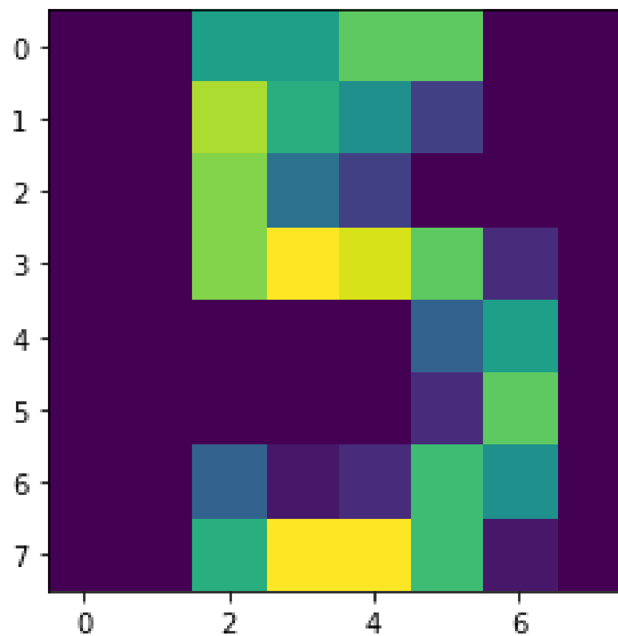
[57]:

```
n = X.shape[0]
perm = list(range(n))
```

```
random.shuffle(perm)
(X, Y) = ({'tr': X[perm[:n//2], ...], 'te': X[perm[n//2:], ...]}, {'tr':Y[perm[:
 ↪n//2]], 'te':Y[perm[n//2:]]})
```

[58]:
```
plt.imshow(X['tr'][0, ...].view(-1,8))
```

[58]: <matplotlib.image.AxesImage at 0x7fb4d6eb48b0>



[59]:
```
width = 64
mb_sz = 128

net1 = torch.nn.Sequential(
    torch.nn.Linear(X['tr'].shape[1], width),
    torch.nn.ReLU(),
    torch.nn.Linear(width, len(sk_digits.target_names)),
)

class Net2(torch.nn.Module):
    def __init__(self, width):
        super(Net2, self).__init__() # boilerplate
        self.conv = torch.nn.Conv2d(1, width, kernel_size = 3)
        self.relu = torch.nn.ReLU()
        self.fc = torch.nn.Linear(6 * 6 * width, 10)

    def forward(self, x):
        x = x.view(x.shape[0], 1, 8, 8)
```

```
        x = self.conv(x)
        x = self.relu(x)
        x = x.view(x.shape[0], -1)
        return self.fc(x)
net2 = Net2(width)

for (net_i, (net_s, net, stepsize)) in enumerate([
    ('dense', net1, 0.1),
    ('conv',  net2, 0.1),
]):
    losses = { 'tr' : [], 'te' : [] }
    for i in range(4000):
        net.zero_grad()
        idxs = random.sample(range(X['tr'].shape[0]), mb_sz)
        (x, y) = (X['tr'][idxs, ...], Y['tr'][idxs])
        yhat = net(x)
        # bakes in ".mean()":
        loss = torch.nn.CrossEntropyLoss()(yhat, y)
        loss.backward()
        with torch.no_grad():
            if (i + 1) % 50 == 0:
                yhat2 = net(X['te'])
                loss2 = torch.nn.CrossEntropyLoss()(yhat2, Y['te'])
                print(f"{i} {loss:.3f} {loss2:.3f}")
                losses['tr'].append(loss.detach())
                losses['te'].append(loss2.detach())

            for P in net.parameters():
                P -= stepsize * P.grad
    for s in ['tr', 'te']:
        plt.figure(1)
        plt.plot(range(len(losses[s])), losses[s],
                  label = f"{net_s} {s}")

    with torch.no_grad():
        yhat2 = net(X['te'])
        loss2 = torch.nn.CrossEntropyLoss(reduction = 'none')(yhat2, Y['te'])
        loss2_sort = loss2.sort()
        for j in range(5):
            plt.figure(2 + 10 * net_i + 2 * j)
            idx = loss2_sort.indices[j]
            yhat2_single = yhat2[idx, ...].argmax().item()
            plt.imshow(X['te'][idx, ...].view(8,8))
            plt.title(f"{j}th least confusing te for {net_s}, yhat␣
 ↪{yhat2_single} y {Y['te'][idx]}")
            plt.figure(2 + 10 * net_i + 2 * j + 1)
            idx = loss2_sort.indices[-1-j]
```

```
            yhat2_single = yhat2[idx, ...].argmax().item()
            plt.imshow(X['te'][idx, ...].view(8,8))
            plt.title(f"{j}th most confusing te for {net_s}, yhat␣
 ↪{yhat2_single} y {Y['te'][idx]}")

plt.figure(1)
plt.title("risk curves")
plt.legend()
plt.show()
```

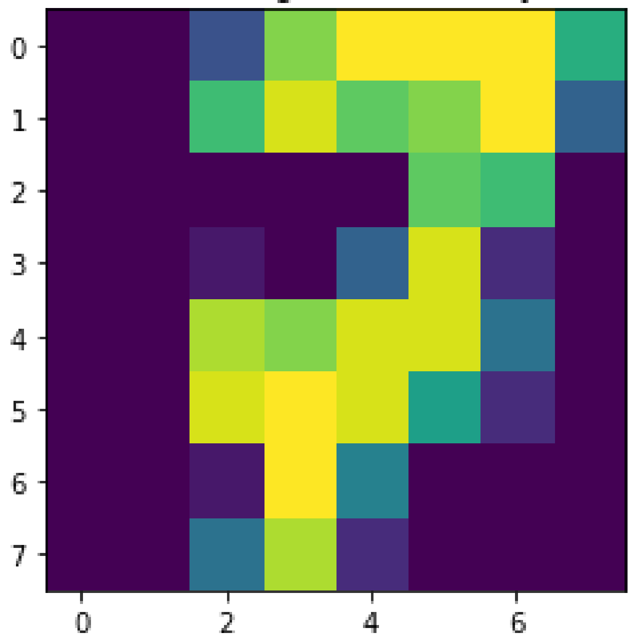
```
49 1.849 1.896
99 1.078 1.147
149 0.639 0.689
199 0.373 0.484
249 0.375 0.379
299 0.301 0.315
349 0.207 0.279
399 0.170 0.248
449 0.208 0.228
499 0.140 0.215
549 0.167 0.201
599 0.119 0.191
649 0.116 0.181
699 0.092 0.175
749 0.131 0.172
799 0.098 0.165
849 0.103 0.161
899 0.119 0.161
949 0.107 0.155
999 0.083 0.152
1049 0.075 0.149
1099 0.112 0.144
1149 0.091 0.143
1199 0.052 0.143
1249 0.092 0.140
1299 0.055 0.139
1349 0.039 0.140
1399 0.033 0.137
1449 0.033 0.135
1499 0.046 0.134
1549 0.070 0.132
1599 0.036 0.134
1649 0.063 0.130
1699 0.064 0.132
1749 0.057 0.132
1799 0.031 0.131
1849 0.039 0.130
```

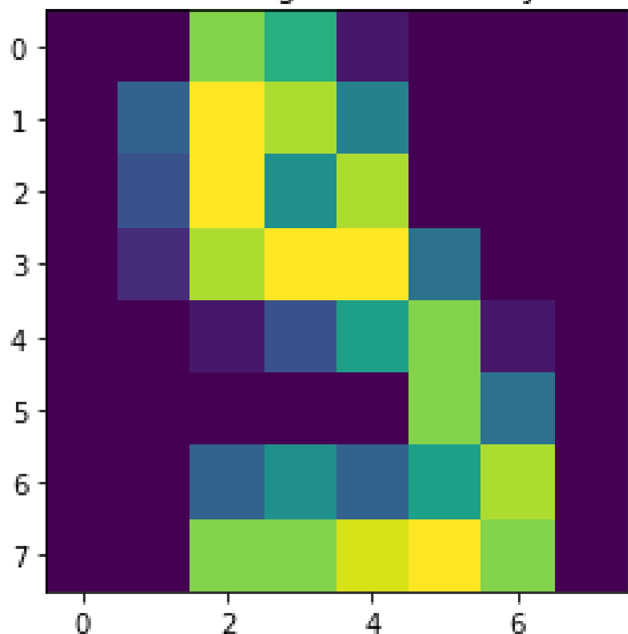
1899 0.052 0.129
1949 0.039 0.129
1999 0.024 0.127
2049 0.031 0.126
2099 0.039 0.126
2149 0.032 0.128
2199 0.037 0.126
2249 0.033 0.126
2299 0.024 0.125
2349 0.033 0.127
2399 0.031 0.125
2449 0.027 0.126
2499 0.023 0.124
2549 0.032 0.124
2599 0.025 0.123
2649 0.017 0.124
2699 0.024 0.124
2749 0.026 0.123
2799 0.030 0.124
2849 0.022 0.124
2899 0.028 0.124
2949 0.016 0.123
2999 0.026 0.123
3049 0.016 0.122
3099 0.013 0.124
3149 0.024 0.123
3199 0.026 0.123
3249 0.028 0.123
3299 0.012 0.124
3349 0.018 0.123
3399 0.018 0.124
3449 0.019 0.122
3499 0.022 0.123
3549 0.022 0.123
3599 0.017 0.124
3649 0.017 0.123
3699 0.014 0.122
3749 0.015 0.123
3799 0.017 0.123
3849 0.017 0.122
3899 0.013 0.122
3949 0.015 0.122
3999 0.015 0.122
49 0.315 0.404
99 0.138 0.254
149 0.119 0.203
199 0.097 0.184
249 0.108 0.181

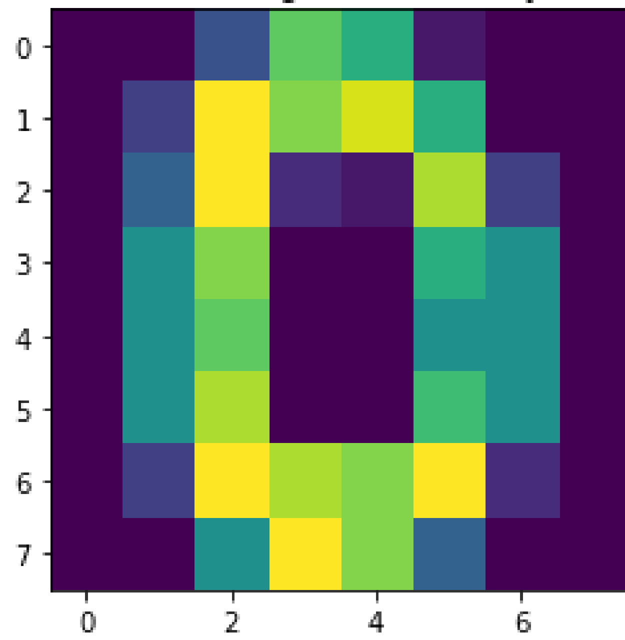
299 0.091 0.179
349 0.062 0.154
399 0.033 0.150
449 0.051 0.141
499 0.054 0.148
549 0.071 0.142
599 0.040 0.138
649 0.036 0.134
699 0.053 0.138
749 0.035 0.130
799 0.041 0.131
849 0.039 0.133
899 0.037 0.130
949 0.034 0.130
999 0.036 0.139
1049 0.030 0.130
1099 0.029 0.133
1149 0.032 0.133
1199 0.035 0.128
1249 0.026 0.130
1299 0.012 0.129
1349 0.020 0.132
1399 0.012 0.128
1449 0.011 0.130
1499 0.017 0.128
1549 0.010 0.133
1599 0.024 0.133
1649 0.019 0.132
1699 0.013 0.134
1749 0.014 0.128
1799 0.012 0.131
1849 0.025 0.136
1899 0.015 0.136
1949 0.018 0.131
1999 0.015 0.132
2049 0.009 0.135
2099 0.013 0.132
2149 0.009 0.136
2199 0.013 0.132
2249 0.013 0.135
2299 0.009 0.134
2349 0.018 0.133
2399 0.010 0.137
2449 0.004 0.136
2499 0.009 0.136
2549 0.008 0.136
2599 0.007 0.138
2649 0.006 0.136

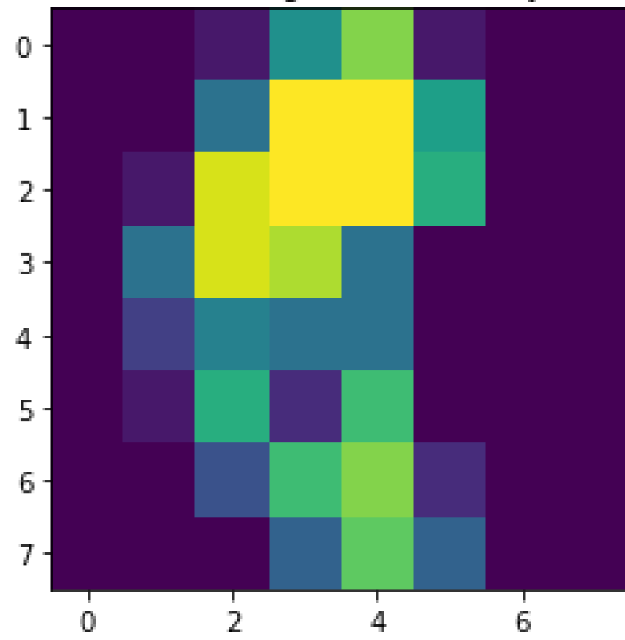
```
2699 0.005 0.136
2749 0.009 0.134
2799 0.013 0.136
2849 0.007 0.133
2899 0.008 0.134
2949 0.006 0.137
2999 0.010 0.134
3049 0.007 0.135
3099 0.009 0.138
3149 0.008 0.134
3199 0.009 0.135
3249 0.004 0.136
3299 0.007 0.135
3349 0.005 0.139
3399 0.006 0.137
3449 0.004 0.134
3499 0.005 0.138
3549 0.004 0.139
3599 0.006 0.142
3649 0.004 0.140
3699 0.005 0.139
3749 0.006 0.138
3799 0.004 0.138
3849 0.007 0.140
3899 0.007 0.139
3949 0.004 0.136
3999 0.005 0.140
```

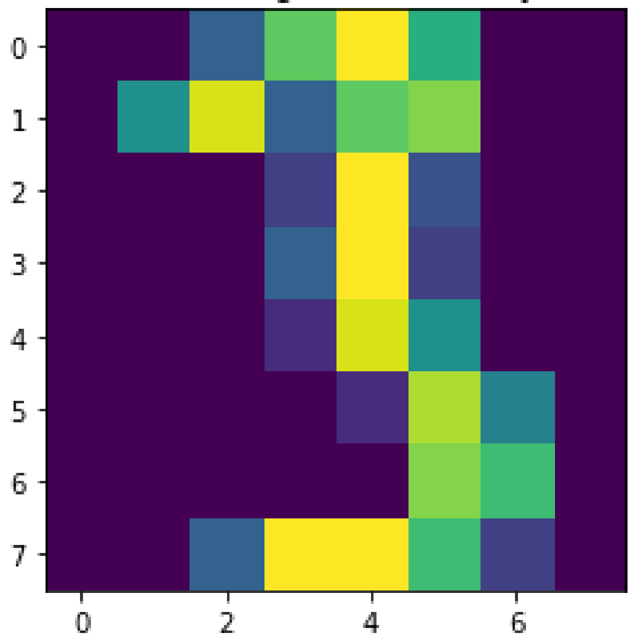
```
/tmp/ipykernel_383958/918564732.py:63: RuntimeWarning: More than 20 figures have
been opened. Figures created through the pyplot interface
(`matplotlib.pyplot.figure`) are retained until explicitly closed and may
consume too much memory. (To control this warning, see the rcParam
`figure.max_open_warning`).
  plt.figure(2 + 10 * net_i + 2 * j + 1)
```

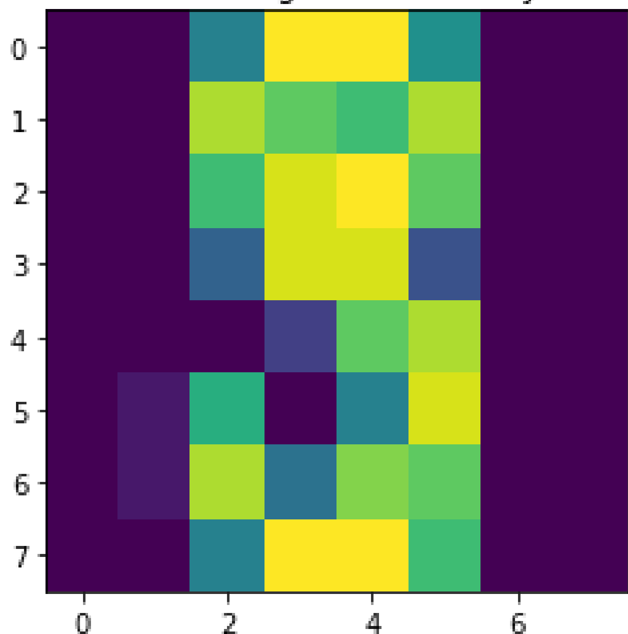

0th least confusing te for dense, yhat 7 y 7
0
1
2
3
4
5
6
7
0
2
4
6

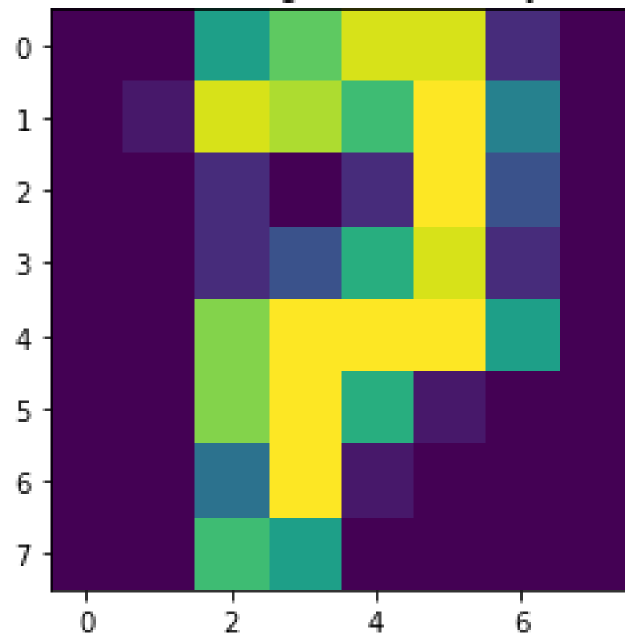

0th most confusing te for dense, yhat 5 y 9
0
1
2
3
4
5
6
7
0
2
4
6

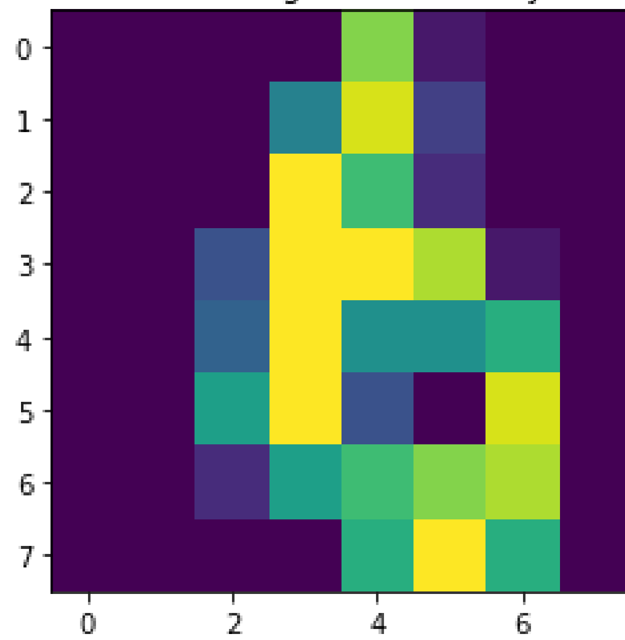

1th least confusing te for dense, yhat 0 y 0
0
1
2
3
4
5
6
7
0
2
4
6

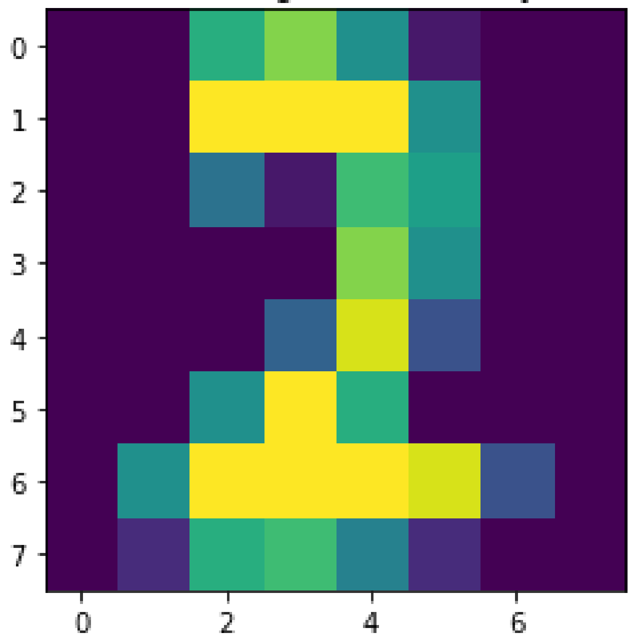

1th most confusing te for dense, yhat 1 y 8
0
1
2
3
4
5
6
7
0
2
4
6

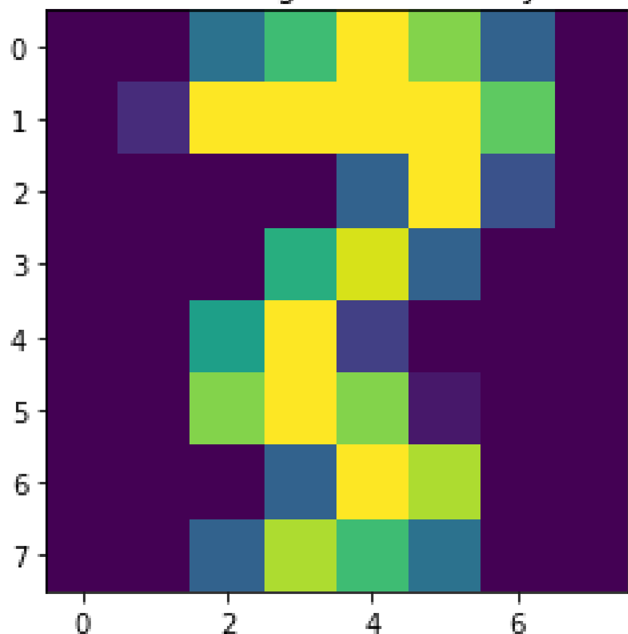

2th least confusing te for dense, yhat 3 y 3
0
1
2
3
4
5
6
7
0
2
4
6

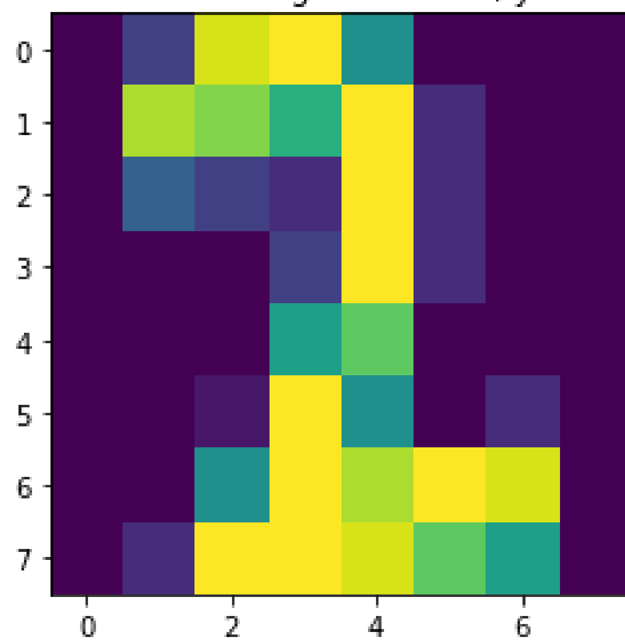

2th most confusing te for dense, yhat 3 y 9
0
1
2
3
4
5
6
7
0
2
4
6

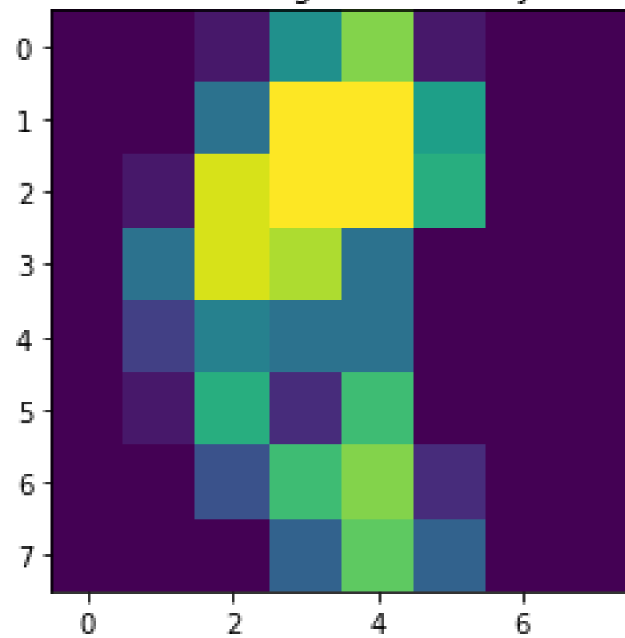

3th least confusing te for dense, yhat 7 y 7
0
1
2
3
4
5
6
7
0
2
4
6

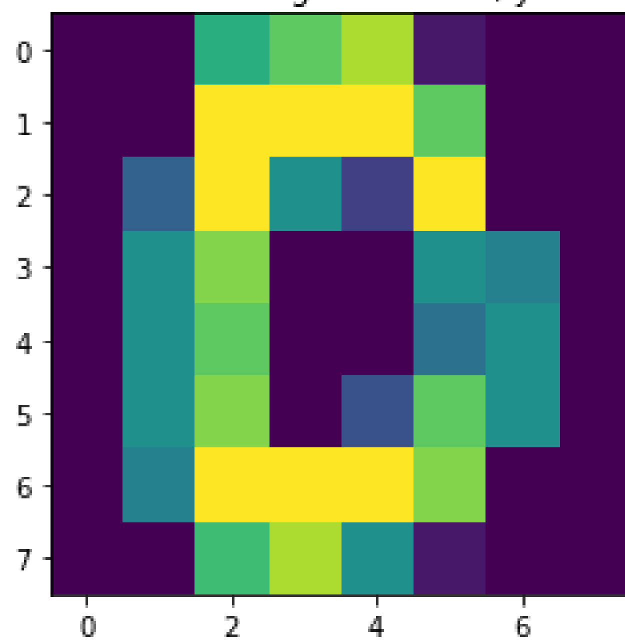

3th most confusing te for dense, yhat 1 y 6
0
1
2
3
4
5
6
7
0
2
4
6

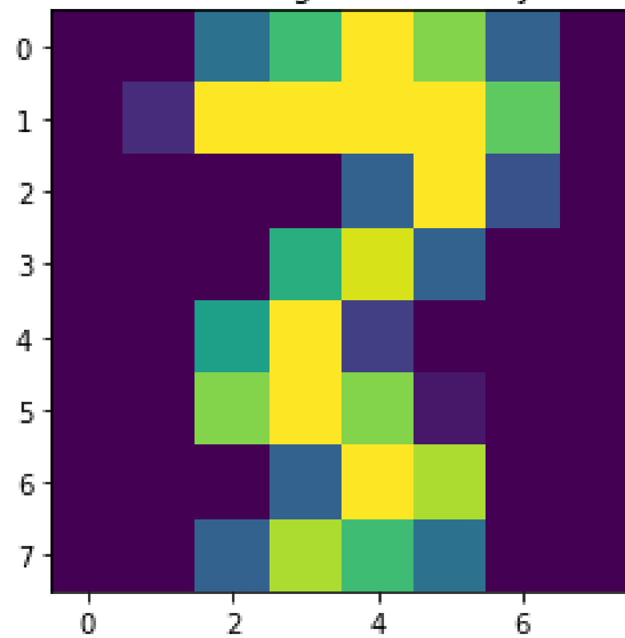

4th least confusing te for dense, yhat 2 y 2
0
1
2
3
4
5
6
7
0
2
4
6

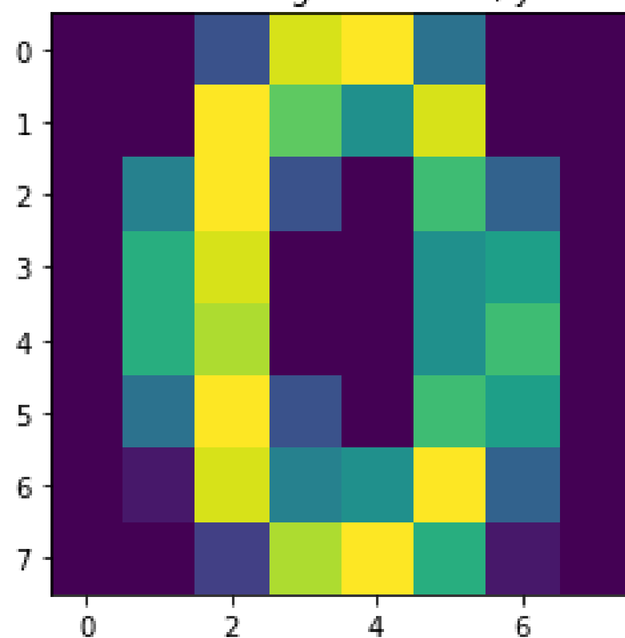

4th most confusing te for dense, yhat 2 y 3
0
1
2
3
4
5
6
7
0
2
4
6

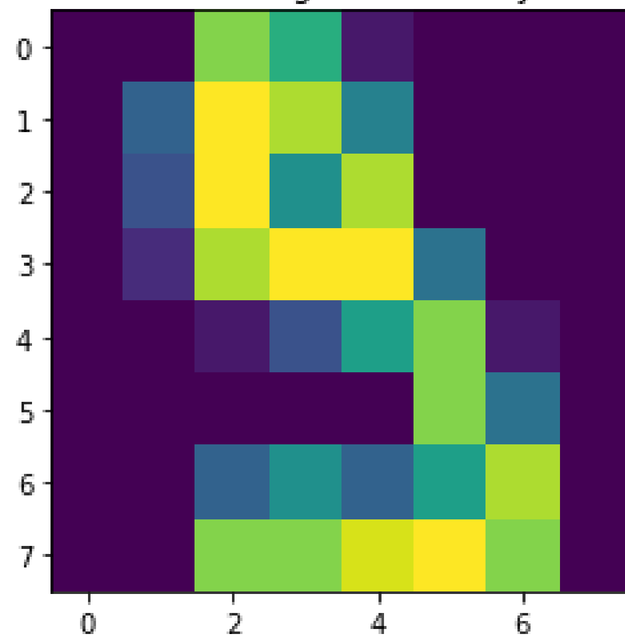

0th least confusing te for conv, yhat 2 y 2
0
1
2
3
4
5
6
7
0
2
4
6

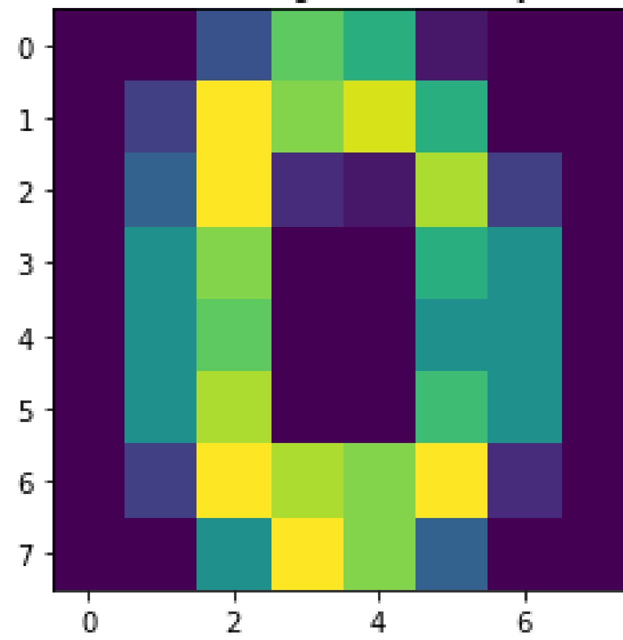

0th most confusing te for conv, yhat 1 y 8
0
1
2
3
4
5
6
7
0
2
4
6

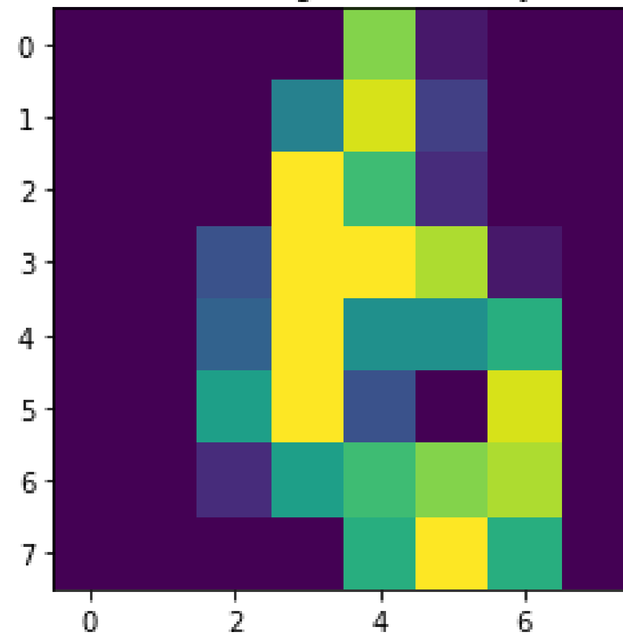

1th least confusing te for conv, yhat 0 y 0
0
1
2
3
4
5
6
7
0
2
4
6

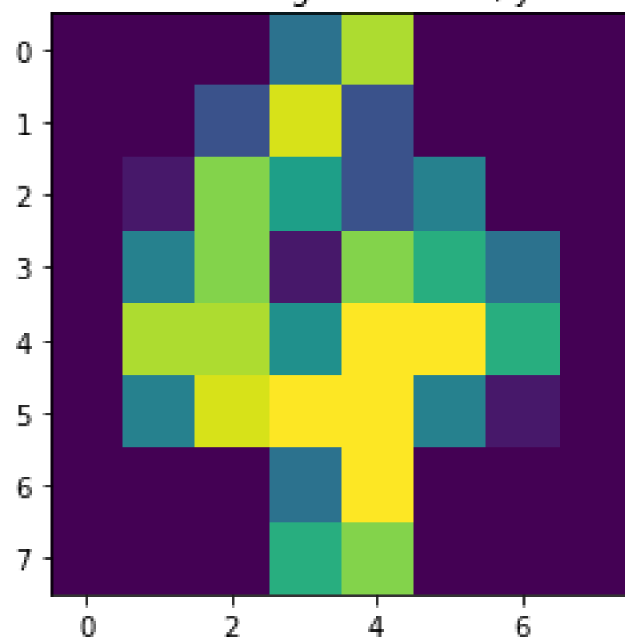

1th most confusing te for conv, yhat 2 y 3
0
1
2
3
4
5
6
7
0
2
4
6

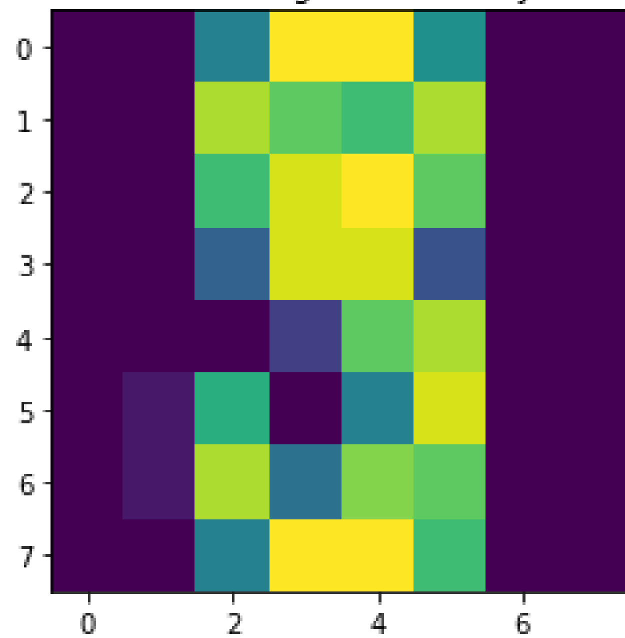

2th least confusing te for conv, yhat 0 y 0
0
1
2
3
4
5
6
7
0
2
4
6

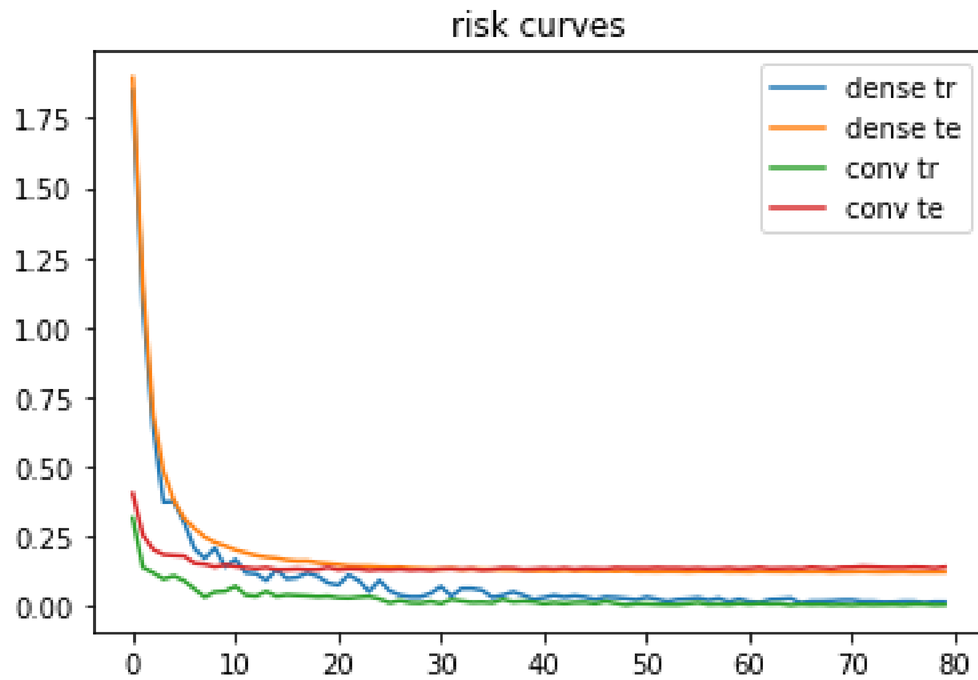

2th most confusing te for conv, yhat 5 y 9
0
1
2
3
4
5
6
7
0
2
4
6


3th least confusing te for conv, yhat 0 y 0
0
1
2
3
4
5
6
7
0
2
4
6


3th most confusing te for conv, yhat 1 y 6
0
1
2
3
4
5
6
7
0
2
4
6


4th least confusing te for conv, yhat 4 y 4
0
1
2
3
4
5
6
7
0
2
4
6


4th most confusing te for conv, yhat 3 y 9
0
1
2
3
4
5
6
7
0
2
4
6



[60]:
```
for P in net2.parameters():
    print(P.shape)
```

```
torch.Size([64, 1, 3, 3])
torch.Size([64])
torch.Size([10, 2304])
torch.Size([10])
```