Lec 23 - pytorch - optim & nn

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

Demo 2 - Using a model

A sample model

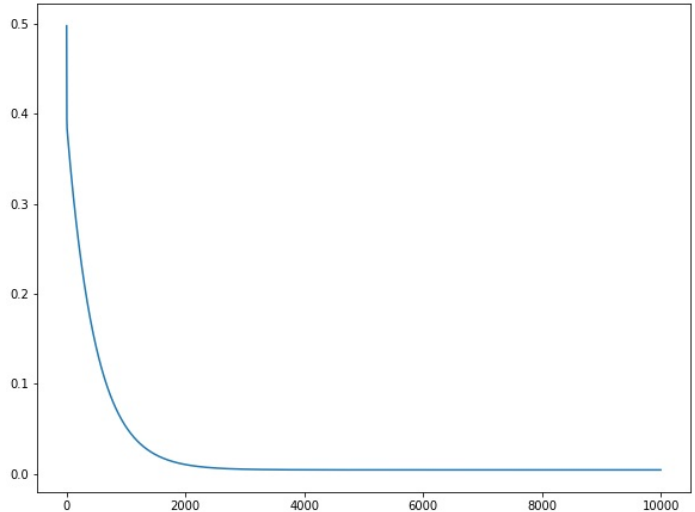
```
class Model(torch.nn.Module):
    def __init__(self, X, y, beta=None):
        super().__init__()

        self.X = X
        self.y = y

        if beta is None:
          beta = torch.zeros(X.shape[1])

        beta.requires_grad = True
        self.beta = torch.nn.Parameter(beta)

    def forward(self, X):
        return X @ self.beta

    def fit(self, opt, n=1000, loss_fn = torch.nn.MSELoss()):
      losses = []

      for i in range(n):
          loss = loss_fn(self(self.X).squeeze(), self.y.squeeze())
          loss.backward()

          opt.step()
          opt.zero_grad()
```

Fitting

```
x = torch.linspace(-math.pi, math.pi, 200)
y = torch.sin(x)

X = torch.vstack((
  torch.ones_like(x),
  x,
  x**2,
  x**3
)).T

m = Model(X, y)
opt = torch.optim.SGD(m.parameters(), lr=1e-3)
losses = m.fit(opt, n=10000)

m.beta.detach()
```

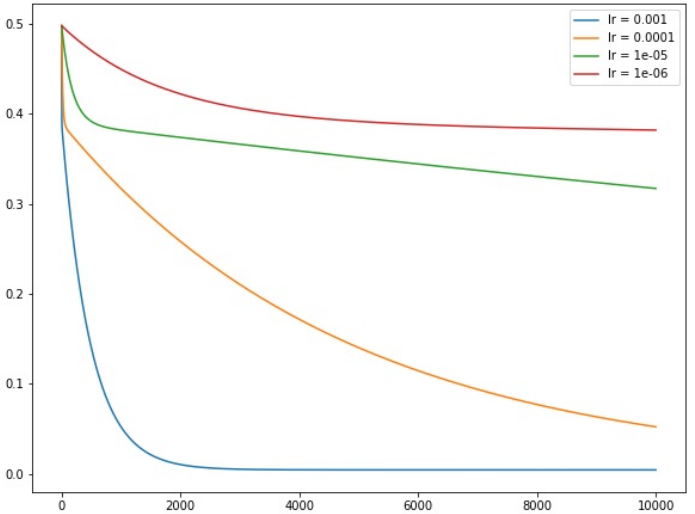
```
## tensor([-1.0707e-08,  8.5455e-01,  3.9629e-09,
```

```
plt.figure(figsize=(8,6), layout="constrained")
plt.plot(losses)
plt.show()
```



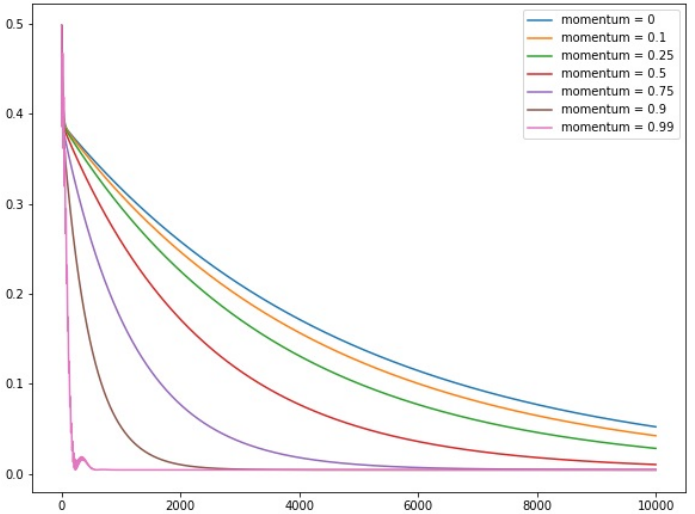
Learning rate and convergence

```
plt.figure(figsize=(8,6), layout="constrained")

for lr in [1e-3, 1e-4, 1e-5, 1e-6]:
  m = Model(X, y)
  opt = torch.optim.SGD(m.parameters(), lr=lr)
  losses = m.fit(opt, n=10000)

  plt.plot(losses, label=f"lr = {lr}")

plt.legend()
plt.show()
```



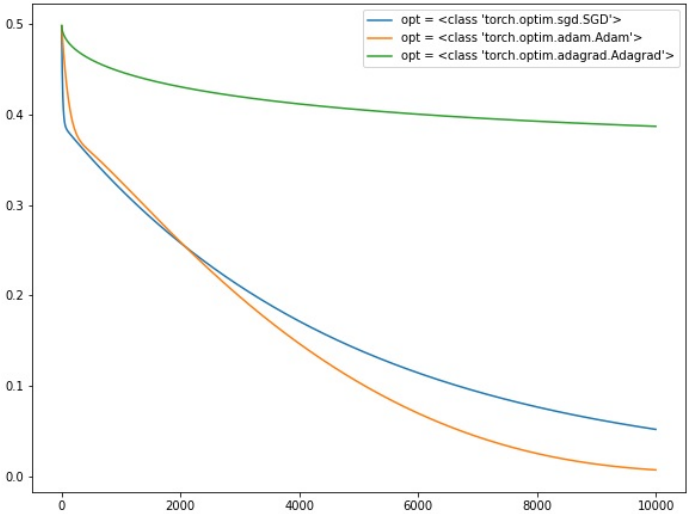
Momentum and convergence

```
plt.figure(figsize=(8,6), layout="constrained")

for mt in [0, 0.1, 0.25, 0.5, 0.75, 0.9, 0.99]:
  m = Model(X, y)
  opt = torch.optim.SGD(m.parameters(), lr=1e-4,
  losses = m.fit(opt, n=10000)

  plt.plot(losses, label=f"momentum = {mt}")

plt.legend()
plt.show()
```



Optimizers and convergence

```
plt.figure(figsize=(8,6), layout="constrained")

opts = (torch.optim.SGD,
        torch.optim.Adam,
        torch.optim.Adagrad)

for opt_fn in opts:
  m = Model(X, y)
  opt = opt_fn(m.parameters(), lr=1e-4)
  losses = m.fit(opt, n=10000)

  plt.plot(losses, label=f"opt = {opt_fn}")

plt.legend()
plt.show()
```



MNIST & Logistic models

MNIST handwritten digits - simplified

```
from sklearn.datasets import load_digits

digits = load_digits()
```

```
X = digits.data
X.shape
```

```
## (1797, 64)
```

```
X[0:3]
```

```
## array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13.,
##         11.,  8.,  0.,  0.,  4., 12.,  0.,  0.,  8.,  8.,  0.,  0.,
##          8.,  0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,
##         10.,  0.,  0.,  0.],
##        [ 0.,  0.,  0., 12., 13.,  5.,  0.,  0.,  0.,  0.,  0.,
##          6.,  0.,  0.,  0.,  7., 15., 16., 16.,  2.,  0.,  0.,  0.,  0
##          0.,  0.,  0.,  0.,  1., 16., 16.,  6.,  0.,  0.,  0.,
##         16., 10.,  0.,  0.],
##        [ 0.,  0.,  0.,  4., 15., 12.,  0.,  0.,  0.,  0.,  3.,
##         16.,  0.,  0.,  0.,  0.,  1.,  6., 15., 11.,  0.,  0.,  0.,
##          0.,  0.,  0.,  9., 16., 16.,  5.,  0.,  0.,  0.,  0.,
##         11., 16.,  9.,  0.]])
```

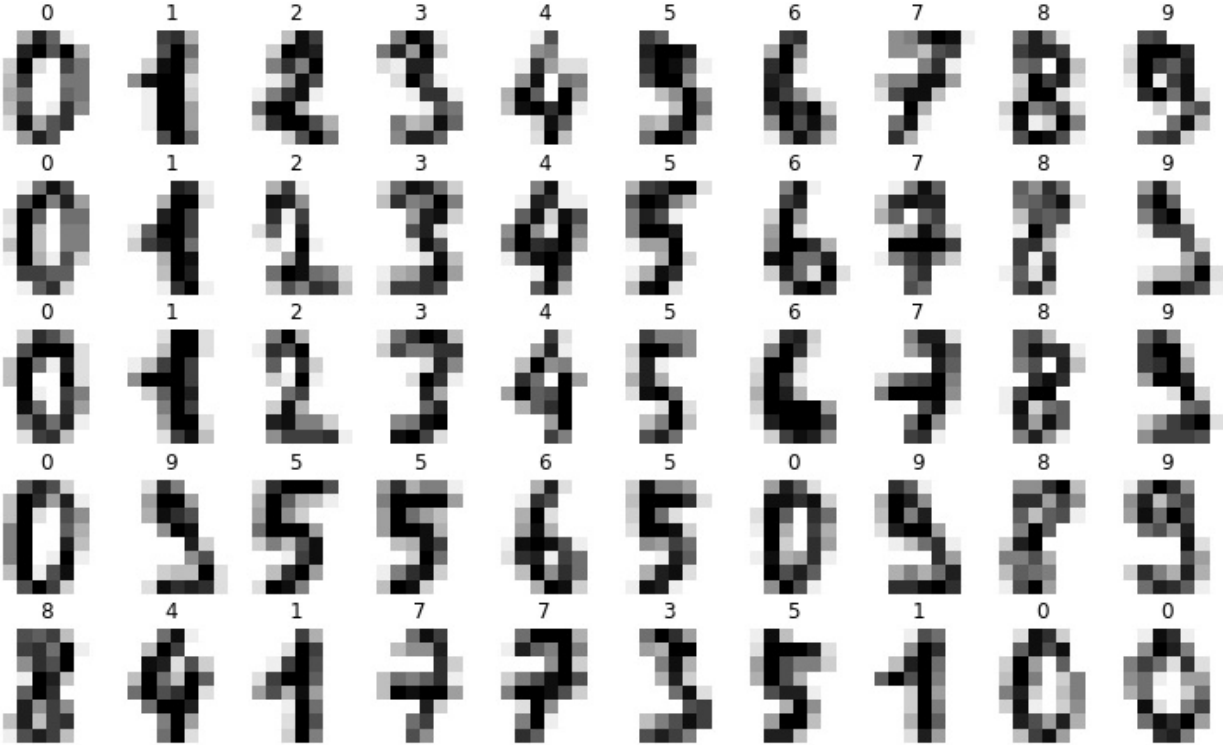
```
y = digits.target
y.shape
```

```
## (1797,)
```

```
y[0:10]
```

```
## array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Example digits



Test train split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, shuffle=True, random_state=1234
)
```

```
X_train.shape
```

```
## (1437, 64)
```

```
y_train.shape
```

```
## (1437,)
```

```
X_test.shape
```

```
## (360, 64)
```

```
y_test.shape
```

```
## (360,)
```

```
from sklearn.linear_model import LogisticRegressi
from sklearn.metrics import accuracy_score

lr = LogisticRegression(penalty='none').fit(X_tra

accuracy_score(y_train, lr.predict(X_train))
```

```
## 1.0
```

```
accuracy_score(y_test, lr.predict(X_test))
```

```
## 0.9583333333333334
```

As Tensors

```
X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train)
X_test = torch.from_numpy(X_test).float()
y_test = torch.from_numpy(y_test)
```

```
X_train.shape
```

```
## torch.Size([1437, 64])
```

```
y_train.shape
```

```
## torch.Size([1437])
```

```
X_test.shape
```

```
## torch.Size([360, 64])
```

```
y_test.shape
```

```
## torch.Size([360])
```

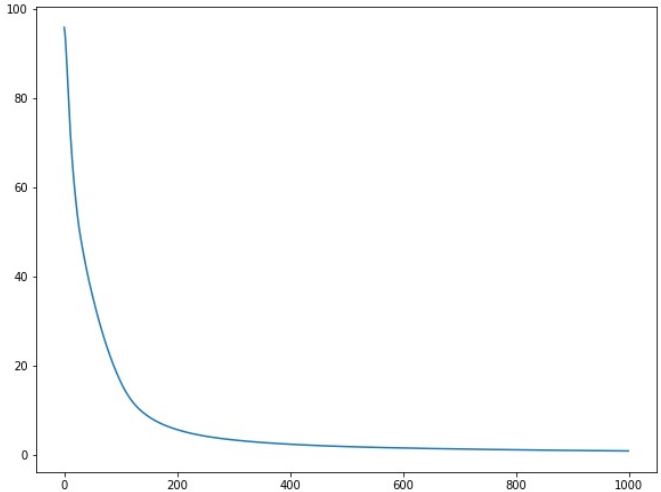
PyTorch Model

```
class mnist_model(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.beta = torch.nn.Parameter(
          torch.randn(input_dim, output_dim, requires_grad=True)
        )
        self.intercept = torch.nn.Parameter(
          torch.randn(output_dim, requires_grad=True)
        )

    def forward(self, X):
        return (X @ self.beta + self.intercept).squeeze()

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses = []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()
```

Cross entropy loss

```
m = mnist_model(64, 10)
l = m.fit(X_train, y_train, X_test, y_test)
```

```
plt.figure(figsize=(8,6), layout="constrained")
plt.plot(l)
plt.show()
```



Out of sample accuracy

```
m(X_test)
```

```
## tensor([[-60.8759, -17.2256,  -9.1371,  ...,  3
##         [ -1.5938,  72.1707, -43.8905,  ..., -1
##         [-68.0310, -44.0990, -16.7110,  ...,  5
##         ...,
##         [ 53.7474,  35.9603,  -1.8724,  ..., -5
##         [-63.3765,  11.4854, -83.6133,  ...,  3
##         [-34.7887,  -6.8412,   0.2186,  ..., -1
##        grad_fn=<SqueezeBackward0>)
```

```
val, index = torch.max(m(X_test), dim=1)
```

```
index
```

```
## tensor([7, 1, 7, 6, 0, 2, 6, 3, 6, 3, 7, 8, 7,
##         3, 6, 6, 0, 5, 4, 1, 8, 1, 2, 3, 2, 7,
##         4, 1, 4, 1, 7, 6, 8, 2, 9, 9, 8, 0, 8,
##         9, 5, 2, 1, 9, 2, 1, 3, 8, 7, 3, 3, 1,
##         5, 2, 2, 4, 5, 4, 7, 6, 5, 7, 2, 4, 1,
##         2, 7, 6, 4, 3, 2, 1, 1, 6, 4, 6, 2, 7,
##         3, 8, 3, 2, 0, 4, 0, 8, 5, 4, 6, 1, 1,
##         3, 8, 6, 4, 7, 1, 5, 7, 4, 7, 4, 3, 2,
```

```
(index == y_test).sum()
```

```
## tensor(324)
```

```
(index == y_test).sum() / len(y_test)
```

```
## tensor(0.9000)
```

Calculating Accuracy

```
class mnist_model(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.beta = torch.nn.Parameter(
          torch.randn(input_dim, output_dim, requires_grad=True)
        )
        self.intercept = torch.nn.Parameter(
          torch.randn(output_dim, requires_grad=True)
        )

    def forward(self, X):
        return (X @ self.beta + self.intercept).squeeze()

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()
          losses.append(loss.item())

          if (i+1) % acc_step == 0:
            val, train_pred = torch.max(self(X_train), dim=1)
            val, test_pred  = torch.max(self(X_test), dim=1)

            train_acc.append( (train_pred == y_train).sum() / len(y_train) )
            test_acc.append( (test_pred == y_test).sum() / len(y_test) )
```

Performance

```
loss, train_acc, test_acc = mnist_model(64, 10).fit(X_train, y_train, X_test, y_test,acc_step=10, n=3000)
```

```
plt.figure(figsize=(8,6), layout="constrained")
plt.plot(train_acc, label="train accuracy")
plt.plot(test_acc, label="test accuracy")
plt.legend()
plt.show()
```

NN Layers

```
class mnist_nn_model(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, X):
        return self.linear(X)

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()
          losses.append(loss.item())

          if (i+1) % acc_step == 0:
            val, train_pred = torch.max(self(X_train), dim=1)
            val, test_pred  = torch.max(self(X_test), dim=1)

            train_acc.append( (train_pred == y_train).sum() / len(y_train) )
```

Linear layer parameters

```
m = mnist_nn_model(64, 10)

m.parameters()
```

```
## <generator object Module.parameters at 0x2a5a1cba0>
```

```
len(list(m.parameters()))
```

```
## 2
```

```
list(m.parameters())[0].shape
```

```
## torch.Size([10, 64])
```

```
list(m.parameters())[1].shape
```

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

Applies a linear transform to the incoming data:

$$y = xA^T + b$$

Performance

```
loss, train_acc, test_acc = m.fit(X_train, y_train, X_test, y_test, n=1500)
```

Feedforward Neural Network

FNN Model

```
class mnist_fnn_model(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nl_step = torch.nn.ReLU(), seed=1234):
        super().__init__()
        self.l1 = torch.nn.Linear(input_dim, hidden_dim)
        self.nl = nl_step
        self.l2 = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, X):
        out = self.l1(X)
        out = self.nl(out)
        out = self.l2(out)
        return out

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
          loss = torch.nn.CrossEntropyLoss()(self(X_train), y_train)
          loss.backward()
          opt.step()

          losses.append(loss.item())

          if (i+1) % acc_step == 0:
            val, train_pred = torch.max(self(X_train), dim=1)
            val, test_pred  = torch.max(self(X_test), dim=1)

            train_acc.append( (train_pred == y_train).sum() / len(y_train) )
            test_acc.append( (test_pred == y_test).sum() / len(y_test) )
```

Model parameters

```
m = mnist_fnn_model(64,64,10)

len(list(m.parameters()))
```

```
## 4
```

```
for i, p in enumerate(m.parameters()):
  print("Param", i, p.shape)
```

```
## Param 0 torch.Size([64, 64])
## Param 1 torch.Size([64])
## Param 2 torch.Size([10, 64])
## Param 3 torch.Size([10])
```

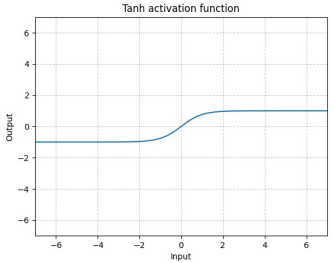
Performance - ReLU

```
loss, train_acc, test_acc = mnist_fnn_model(64,64,10).fit(
  X_train, y_train, X_test, y_test, n=2000
)
test_acc[-5:]
```

```
## [tensor(0.9750), tensor(0.9750), tensor(0.9750), tensor(0.9750), tensor(0.9750)]
```

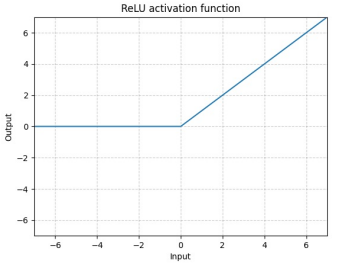
Non-linear activation functions

$$\text{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$



$$\text{Sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

$$\text{ReLU}(x) = \max(0, x)$$



Performance - tanh

```
loss, train_acc, test_acc = mnist_fnn_model(64,64,10, nl_step=torch.nn.Tanh()).fit(
  X_train, y_train, X_test, y_test, n=2000
)
test_acc[-5:]
```

```
## [tensor(0.9694), tensor(0.9694), tensor(0.9694), tensor(0.9694), tensor(0.9694)]
```

Performance - Sigmoid

```
loss, train_acc, test_acc = mnist_fnn_model(64,64,10, nl_step=torch.nn.Sigmoid()).fit(
  X_train, y_train, X_test, y_test, n=2000
)
test_acc[-5:]
```

```
## [tensor(0.9556), tensor(0.9556), tensor(0.9556), tensor(0.9556), tensor(0.9556)]
```

Multilayer FNN Model

```
class mnist_fnn2_model(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nl_step = torch.nn.ReLU(), seed=1234):
        super().__init__()
        self.l1 = torch.nn.Linear(input_dim, hidden_dim)
        self.nl = nl_step
        self.l2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.nl = nl_step
        self.l3 = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, X):
        out = self.l1(X)
        out = self.nl(out)
        out = self.l2(out)
        out = self.nl(out)
        out = self.l3(out)
        return out

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      loss_fn = torch.nn.CrossEntropyLoss()
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
          loss = loss_fn(self(X_train), y_train)
          loss.backward()
          opt.step()

          losses.append(loss.item())

          if (i+1) % acc_step == 0:
```

Performance

```
loss, train_acc, test_acc = mnist_fnn2_model(64,64,10, nl_step=torch.nn.ReLU()).fit(
  X_train, y_train, X_test, y_test, n=1000
)
test_acc[-5:]
```

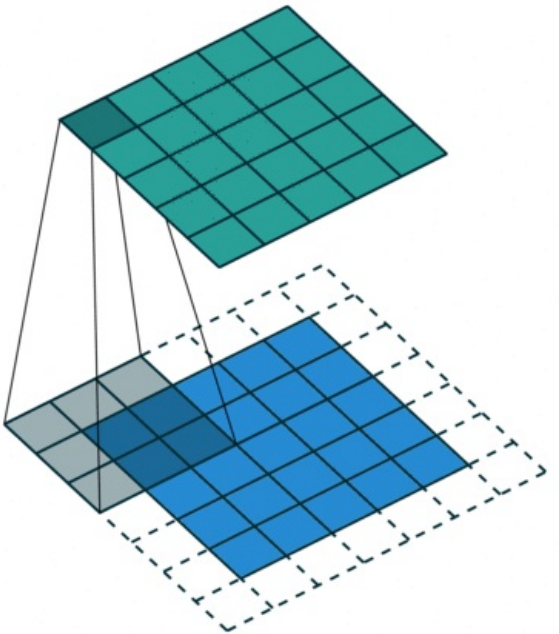
```
## [tensor(0.9611), tensor(0.9611), tensor(0.9611), tensor(0.9611), tensor(0.9611)]
```

Convolutional NN

2d convolutions

3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0



Source

nn.Conv2d()

```
cv = torch.nn.Conv2d(
  in_channels=1, out_channels=4,
  kernel_size=3,
  stride=1, padding=1
)
```

```
list(cv.parameters())
```

```
## [Parameter containing:
## tensor([[[[ 0.3185, -0.1251,  0.0127],
##           [ 0.2319,  0.1078,  0.0828],
##           [-0.2651,  0.1484, -0.0063]]],
##
##
##         [[[-0.3191,  0.1816, -0.0369],
##           [ 0.0180,  0.0299, -0.0892],
##           [ 0.2278, -0.2236,  0.2685]]],
##
##
##         [[[ 0.0027, -0.3022,  0.0216],
##           [-0.0097, -0.1375,  0.0558],
##           [ 0.0894,  0.1148, -0.2137]]],
##
```

Applying Conv2d()

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

```
## tensor([[ 0.,  0.,  0., 10., 11.,  0.,  0.,  0.,  0.,  0.,  9., 16.,  6.,  0.,
##           0.,  0.,  0.,  0., 15., 13.,  0.,  0.,  0.,  0.,  0.,  0., 14., 10.,
##           0.,  0.,  0.,  0.,  0.,  1., 15., 12.,  8.,  2.,  0.,  0.,  0.,  0.,
##          12., 16., 16., 16., 10.,  1.,  0.,  0.,  7., 16., 12., 12., 16.,  4.,
##           0.,  0.,  0.,  9., 15., 12.,  5.,  0.]])
```

```
X_train[[0]].shape
```

```
## torch.Size([1, 64])
```

```
cv(X_train[[0]])
```

```
## RuntimeError: Expected 3D (unbatched) or 4D (batched) input to conv2d, but got input of size: [1, 64]
```

```
cv(X_train[[0]].view(1,8,8))
```

```
## tensor([[[ 1.4635e-01,  8.9731e-02,  2.2091e+00,  2.0856e+00,  3.0008e-01,
##            1.1069e+00,  1.4635e-01,  1.4635e-01],
##          [ 1.4635e-01,  7.9740e-01,  4.7118e+00,  1.2956e+00,  2.8663e+00,
##            5.0414e+00,  1.4635e-01,  1.4635e-01],
##          [ 1.4635e-01,  1.4148e+00,  3.9304e+00,  3.7392e+00,  4.8559e+00,
```

Pooling

```
x = torch.tensor(
  [[[0,0,0,0],
    [0,1,2,0],
    [0,3,4,0],
    [0,0,0,0]]],
  dtype=torch.float
)
x.shape
```

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

```
p = torch.nn.MaxPool2d(kernel_size=2, stride=1)
p(x)
```

```
## tensor([[[1., 2., 2.],
##          [3., 4., 4.],
##          [3., 4., 4.]]])
```

```
p = torch.nn.MaxPool2d(kernel_size=3, stride=1, p
p(x)
```

```
## tensor([[[1., 2., 2., 2.],
```

```
p = torch.nn.AvgPool2d(kernel_size=2)
p(x)
```

```
## tensor([[[0.2500, 0.5000],
##          [0.7500, 1.0000]]])
```

```
p = torch.nn.AvgPool2d(kernel_size=2, padding=1)
p(x)
```

```
## tensor([[[0.0000, 0.0000, 0.0000],
##          [0.0000, 2.5000, 0.0000],
##          [0.0000, 0.0000, 0.0000]]])
```

Convolutional model

```
class mnist_conv_model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn  = torch.nn.Conv2d(
          in_channels=1, out_channels=8,
          kernel_size=3, stride=1, padding=1
        )
        self.relu = torch.nn.ReLU()
        self.pool = torch.nn.MaxPool2d(kernel_size=2)
        self.lin  = torch.nn.Linear(8 * 4 * 4, 10)

    def forward(self, X):
        out = self.cnn(X.view(-1, 1, 8, 8))
        out = self.relu(out)
        out = self.pool(out)
        out = self.lin(out.view(-1, 8 * 4 * 4))
        return out

    def fit(self, X_train, y_train, X_test, y_test, lr=0.001, n=1000, acc_step=10):
      opt = torch.optim.SGD(self.parameters(), lr=lr, momentum=0.9)
      losses, train_acc, test_acc = [], [], []

      for i in range(n):
          opt.zero_grad()
```

Performance

```
loss, train_acc, test_acc = mnist_conv_model().fit(
  X_train, y_train, X_test, y_test, n=1000
)
test_acc[-5:]
```

```
## [tensor(0.9667), tensor(0.9667), tensor(0.9667), tensor(0.9667), tensor(0.9667)]
```