ESE 546, FALL 2020

HOMEWORK 0

SHEIL SARDA [SHEILS@SEAS],
COLLABORATORS: ARJUN GOVIND [AGOVIND@WHARTON]

Solution 1 (Time spent: 5 hour)**.** a A proposed new objective function could be:

$$\text{minimize} \frac{1}{2}||\theta||^2 + \sum_{i=1}^{n} \xi_i$$

b Support samples in an `SVM` are datapoints which are the closest to the hyperplane described by the `SVM`'s output vector w.

c We did not construct a test dataset is that this is a model with very few hyperparameters, so it will be easy to validate and tune. Thus, it does not require much data to validate and we can better utilize that data in the training set.

If the model had many hyperparameters, we would want to maintain a large test and validation set.

d The parameter C is a regularization parameter on the model weights. Its default value is 1.0. Higher values of C enable our model to have larger weights, since they get penalized less compared to lower values of C.

The parameter γ is the kernel coefficient with a default value of `scale`. It is used to determine the kernel method used by the `SVM`.

Validation accuracy and 10-class confusion matrix is copied below.

```
Classification report for classifier SVC():
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1343
           1       0.98      0.99      0.99      1600
           2       0.97      0.98      0.98      1380
           3       0.97      0.97      0.97      1433
           4       0.97      0.98      0.98      1295
           5       0.98      0.97      0.98      1273
           6       0.99      0.99      0.99      1396
           7       0.97      0.97      0.97      1503
           8       0.97      0.96      0.97      1357
           9       0.97      0.96      0.97      1420
```

```
accuracy                           0.98     14000
macro avg      0.98      0.98      0.98     14000
weighted avg   0.98      0.98      0.98     14000


Confusion matrix:
 [[1329    1    5    0    1    2    1    1    2    1]
  [   0 1588    3    3    1    0    0    2    2    1]
  [   0    3 1355    2    2    3    1    7    6    1]
  [   0    2   12 1385    2    9    2    9    7    5]
  [   1    1    2    0 1268    0    2    2    2   17]
  [   0    2    2   12    1 1239    8    1    8    0]
  [   1    0    0    0    3    5 1385    0    2    0]
  [   1    7   10    0    7    1    0 1464    1   12]
  [   2    6    9   13    4    6    5    4 1305    3]
  [   6    8    1    7   13    3    0   12    6 1364]]
```

The ratio of number of support samples to total number of training samples for the trained classifier is:

```
Training Samples: 56000

 Class   Samples (Support / Training) %
     0    1343    2.40%
     1    1600    2.86%
     2    1380    2.46%
     3    1433    2.56%
     4    1295    2.31%
     5    1273    2.27%
     6    1396    2.49%
     7    1503    2.68%
     8    1357    2.42%
     9    1420    2.54%
```

e The parameter ”shrinking” is responsible for speeding up convergence by identifying and removing some elements which are bounded during training to simplify the optimization problem.

The optimization algorithm used to fit the SVM in scikit-learn is an `SMO`-type (Sequential Minimal Optimization) decomposition method proposed in Fan et al. (2005).

f `SVC` solves the multi-class classification problem using a `one-vs-one` solution where classifiers are trained for each pair of classes ($\binom{n}{2}$ classifiers). Each binary classifier predicts a class label, and the label with the most votes is predicted by the `one-vs-one` strategy.

An alternative is the `one-vs-rest` strategy where the multi-class classification problem is split into one binary classifier for each class.

g All the hyper-parameters tested with their accuracy:

	params	mean_test_score
0	{'C': 1, 'shrinking': True}	0.9568
1	{'C': 1, 'shrinking': False}	0.9568
2	{'C': 10, 'shrinking': True}	0.9629
3	{'C': 10, 'shrinking': False}	0.9629
4	{'C': 50, 'shrinking': True}	0.9627
5	{'C': 50, 'shrinking': False}	0.9627

h Accomplished in code.

i Accomplished in code.

Solution 2 (Time spent: 1 hour)**.**

$$
\begin{aligned}
&\mathop{\mathbb{E}}_{X}[\varphi(X)] && \geq \varphi(\mu) \\
\implies & p_1\varphi(x_1) + p_2\varphi(x_2) + \dots p_k\varphi(x_k) \\
=&(p_1+p_2)\left(\left(\frac{p_1}{p_1+p_2}\right)\varphi(x_1) + \left(\frac{p_2}{p_1+p_2}\right)\varphi(x_2)\right) + \ldots + p_k\varphi(x_k) \\
\leqslant&(p_1+p_2)\varphi\left(\left(\frac{p_1}{p_1+p_2}\right)x_1 + \left(\frac{p_2}{p_1+p_2}\right)x_2\right) + \ldots + p_k\varphi(x_k) \\
\leqslant&\varphi\left((p_1+p_2)\left(\left(\frac{p_1}{p_1+p_2}\right)x_1 + \left(\frac{p_2}{p_1+p_2}\right)x_2\right) + \ldots + p_kx_k\right) \\
=&\varphi(p_1x_1 + p_2x_2 + \ldots + p_kx_k) \\
=&\varphi(\mu)
\end{aligned}
$$

Solution 3 (Time spent: 8 hours)**.** a Images plotted in code.

b Linear layer:

```
class linear_t:

def __init__(self):
  self.input_size = 784
  self.classes = 10

  # initialize to appropriate sizes, fill with Gaussian entries
  # normalize to make Frobenius norm of w, b equal to 1
  self.w = np.random.normal(loc=0, scale=1, size=(self.classes, self.input_
  self.w = self.w / np.linalg.norm(self.w)

  self.b = np.random.normal(loc=0, scale=1, size=(self.classes, 1))
  self.b = self.b / np.linalg.norm(self.b)

  self.dw = np.zeros_like(self.w)
  self.db = np.zeros_like(self.b)

def forward(self, h_l):
  h_l1 = np.matmul(h_l, np.transpose(self.w)) + self.b.T

  # cache h^1 in forward because we will need it to compute
  # dw in backward
  self.hl = h_l
  return h_l1

def backward(self, dh_l1):
  dh_l = np.matmul(dh_l1, self.w)
  dw = np.matmul(dh_l1.T, self.hl)
  db = np.matmul(dh_l1.T, np.ones([self.hl.shape[0], 1]))

  self.dw, self.db = dw, db
  return dh_l

def zero_grad(self):
  # useful to delete stored backprop gradients of
  # previous mini-batch before you start a new mini-batch
  self.dw , self.db = 0*self.dw, 0*self.db
```

c Rectified Linear Unit Layer:

```
class relu_t:

def forward(self, h_l):
  return np.maximum(0, h_l)

def backward(self, dh_l1):
  x = np.array(dh_l1, copy=True)
  x[x <= 0] = 0
  x[x  > 0] = 1
  return x

def zero_grad(self):
  pass
```

d
```
class softmax_cross_entropy_t:

def __init__(self):
  # no parameters, nothing to initialize
  self.h_l1 = None
  self.y = None

def forward(self, h_l, y):
  expd = np.exp(h_l)
  self.h_l1 = expd / np.sum(expd, axis=1, keepdims=True)
  self.y = y
  # compute average loss ell(y) over a mini-batch
  batch = self.h_l1.shape[0]
  ell = np.sum(-np.log(self.h_l1[range(batch), self.y])) / batch
  error = np.sum(np.not_equal(self.y, np.argmax(self.h_l1, axis=1)) / batch
  return ell, error

def backward(self):
  # as we saw in notes, backprop input to the
  # loss layer is 1, so this function does not
  # take any arguments
  batch = self.y.shape[0]
  self.h_l1[range(batch), self.y] -= 1
  dh_l = self.h_l1 / batch
  return dh_l
```

```
  def zero_grad(self):
    pass
```

e

f Done in code.

g Done in code.

h