Machine Learning HW2

Q1)

```python
class CustomFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        return grad_output * 0.5 * 3 *(5 * input **2 - 1)



dtype = torch.float
device = torch.device("cuda:0")


x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
y = torch.sin(x) # We approximate this sine function.
```

```python
# In our model, we have 4 weights to train: y = a + b * P3(c + d * x).
# These weights need to be initialized.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
a = torch.full((), 0.0, device=device, dtype=dtype, requires_grad=True)
b = torch.full((), -1.0, device=device, dtype=dtype, requires_grad=True)
c = torch.full((), 0.0, device=device, dtype=dtype, requires_grad=True)
d = torch.full((), 0.3, device=device, dtype=dtype, requires_grad=True)

learning_rate = 5e-6
for t in range(2000):
    P3 = CustomFunction.apply

    # Forward pass: predict y.
    # P3 using our custom backward function.
    y_pred = a + b * P3(c + d * x)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass.
    loss.backward()

    # Update weights using gradient descent
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

```python
        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} * P3({c.item()} + {d.item()} x)')
```

Description:

For the first question, we are to approximate sine function using 4 parameters (a,b,c,d). This involves implementing backward pass for the given custom function. In the backward pass, we receive grad_output containing the gradient of the loss w.r.t. the output. Using this, we compute the gradient of the loss w.r.t. the input. In the training loop, we use autograd on our loss which automatically computes gradient for all its tracked operations by calling backward(). We obtain the resulting gradient for each variable and perform gradient descent to update the variables by some learning rate value. This basically moves the variables we are estimating in the direction that minimizes the loss value. We assign None value to each gradients after each update.

Results:

```
99 209.9583282470703
199 144.66018676757812
299 100.70249938964844
399 71.03519439697266
499 50.97850799560547
599 37.403133392333984
699 28.206865310668945
799 21.97318458557129
899 17.7457275390625
999 14.877889633178711
1099 12.931764602661133
1199 11.610918045043945
1299 10.714248657226562
1399 10.105474472045898
1499 9.692106246948242
1599 9.411375045776367
1699 9.220745086669922
1799 9.091285705566406
1899 9.003361701965332
1999 8.943639755249023
Result: y = -5.733219801684619e-11 + -2.208526849746704 * P3(-2.4259058650777376e-10 + 0.2554861009120941 x)
```

Q2)

Description:

#	Name	equation
1	Linear	$k\left(x_i, x_j\right) = x_i \cdot x_j + c$
2	Poly	$k\left(x_i, x_j\right) = \left(x_i \cdot x_j + c\right)^d$
3	Sigmoid	$k\left(x_i, x_j\right) = tanh(\kappa x_i, x_j + c)$, for some (not every) $\kappa > 0$ and $c < 0$.
4	Log	$k\left(x_i, x_j\right) = -log \left\| x_i - x_j \right\|^d + 1$.
5	Multiquadric	$k\left(x_i, x_j\right) = sqrt\left(\left\| x_i - x_j \right\|^2 + c^2\right)$
6	Rbf	$k\left(x_i, x_j\right) = exp(-\gamma \left\| x_i - x_j \right\|^2)$, for $\gamma > 0$.
7	Fourier	$k\left(x_i, x_j\right) = \left(1 - q^2\right) / \left(2\left(1 - 2q\cos\left(x_i - x_j\right) + q^2\right)\right)$
8	Tstudent	$k\left(x_i, x_j\right) = 1 / \left(1 + \left\| x_i - x_j \right\|^d\right)$
9	thinplate	$k\left(x_i, x_j\right) = \left\| x_i - x_j \right\|^{2n+1}$
10	cosine	$k\left(x_i, x_j\right) = x_i \cdot x_j / \left(\left\| x_i \right\| \cdot \| x_j \|\right)$
11	wave	$k\left(x_i, x_j\right) = (\theta / \left\| x_i - x_j \right\|) sin(\left\| x_i - x_j \right\| / \theta)$

Implementation:

```
# 1) Linear Kernel
def linearK(x1, x2, C=0.001):
  return np.dot(x1, x2.T) + C

# 2) Poly Kernel
def polyK(x1, x2, d=2):
  return np.power(np.dot(x1, x2.T),d)

# 3) Sigmoid Kernel
def sigmoidK(x1, x2, alpha=0.001, C=1):
  alpha = 1/x1.shape[1]
  return np.tanh(alpha*np.dot(x1, x2.T)+C)

# 4) Log Kernel
def logK(x1, x2, d=2):
  euc = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      euc[i][j] = np.sqrt(np.sum(np.square(np.subtract(a,b))))

  return -np.log(np.power(euc, d)+1)

# 5) Multiquadric Kernel
def multiquadricK(x1,x2, C=1000):
  euc = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      euc[i][j] = np.sqrt(np.sum(np.square(np.subtract(a,b))))
  val = np.square(euc) + C**2
  return np.sqrt(val)
```

```python
# 6) rbf Kernel
def rbfK(x1, x2, r=0.1):
  euc = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      euc[i][j] = np.sqrt(np.sum(np.square(np.subtract(a,b))))

  return np.exp(-r*np.square(euc))

# 7) Fourier Kernel
def fourierK(x1, x2, q=0.9):
  euc = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      euc[i][j] = np.sqrt(np.sum(np.square(np.subtract(a,b))))

  denom = 2*(1 - 2*q*np.cos(euc) + q**2)
  return (1-q**2)/denom

# 8) Tstudent Kernel
def tstudentK(x1, x2, d=100):
  euc = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      euc[i][j] = np.sqrt(np.sum(np.square(np.subtract(a,b))))

  val = 1+ np.power(euc, d)
  return 1/val
```

```python
# 9) Thin plate Kernel
def thinplateK(x1, x2, n=6):
  euc = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      euc[i][j] = np.sqrt(np.sum(np.square(np.subtract(a,b))))
  return np.power(euc,2*n+1)

# 10) Cosine Kernel
def cosineK(x1, x2):
  prod = np.dot(x1, x2.T)
  denom = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      denom[i][j] = np.dot(np.linalg.norm(a),np.linalg.norm(b))
  return prod/denom

# 11) Wave Kernel
def waveK(x1, x2, sigma=2):
  euc = np.empty((x1.shape[0],x2.shape[0]), dtype=np.float64)
  for i,a in enumerate(x1):
    for j,b in enumerate(x2):
      euc[i][j] = np.sqrt(np.sum(np.square(np.subtract(a,b))))
  return (sigma/euc)*(np.sin(euc/sigma))
```

Screen capture:

1) Linear Kernel

```python
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=linearK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)
```

```
0.375
0.323867478025693
```

2) Poly Kernel

```python
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=polyK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)
```

```
0.725
0.6925227113906359
```

3) Sigmoid Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=sigmoidK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

0.45
0.42708333333333337
```

4) Log Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=logK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760: DataConve
  y = column_or_1d(y, warn=True)
0.5
0.4791666666666667
```

5) Multiquadric Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=multiquadricK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
  y = column_or_1d(y, warn=True)
0.5
0.4884910485933504
```

6) Rbf Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=rbfK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
  y = column_or_1d(y, warn=True)
0.625
0.5943204868154158
```

7) Fourier Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=fourierK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
  y = column_or_1d(y, warn=True)
0.575
0.5523370638578011
```

8) Tstudent Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=tstudentK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
  y = column_or_1d(y, warn=True)
0.625
0.5943204868154158
```

9) Thinplate Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=thinplateK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
  y = column_or_1d(y, warn=True)
0.6
0.5238095238095237
```

10) Cosine Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=cosineK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
  y = column_or_1d(y, warn=True)
0.5
0.494949494949495
```

11) Wave Kernel

```
#You must use a random state of 2011 for this homework.
clf = SVC(random_state=2011, kernel=waveK)
clf.fit(X, Y)
yp = clf.predict(XTe)
print(accuracy_score(YTe, yp))
print(f1_score(YTe, yp, average='macro'))

# The version of sklearn should be "0.22.2.post1" for reproducibility.
print(sklearn.__version__)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
  y = column_or_1d(y, warn=True)
0.35
0.25925925925925924
```

Results:

Kernel	*accuracy*	*F1 score*
Linear	0.375	0.323867478025693
Poly	0.725	0.6925227113906359
Sigmoid	0.45	0.42708333333333337
Log	0.5	0.4791666666666667
Multiquadric	0.5	0.48849104859335 04
Rbf	0.625	0.5943204868154158
Fourier	0.575	0.5523370638578011
Tstudent	0.625	0.5943204868154158
Thin-plate	0.6	0.5238095238095237
Cosine	0.5	0.494949494949495
Wave	0.35	0.25925925925925924