Deep Learning School

Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Домашнее задание. Обучение нейронных сетей на PyTorch.

В этом домашнем задании вам предстоит предсказывать типы небесных объектов. Эту задачу вы будете решать с помощью нейронных сетей, используя библиотеку PyTorch.

Вам необходимо заполнить пропуски в ноутбуке. Кое-где вас просят сделать выводы о проделанной работе. Постарайтесь ответить на вопросы обдуманно и развёрнуто.

В этом домашнем задании мы используем новый метод проверки --- Peer Review.

Peer Review — альтернативный способ проверки ваших заданий, который подразумевает, что после сдачи задания у вас появится возможность (и даже моральная обязанность, но не строгое обязательство) проверить задания нескольких ваших однокурсников. Соответственно, и ваши работы будут проверять другие учащиеся курса. Для выставления оценки необходимо будет, чтобы вашу работу проверило по крайней мере 3 ваших однокурсника. Вы же, выступая в роли проверяющего, сможете узнать больше о выполненном задании, увидеть, как его выполняли другие.

Чем больше заданий однокурсников вы проверите, тем лучше! Но, пожалуйста, проверяйте внимательно. По нашим оценкам, на проверку одной работы у вас уйдёт 5-10 минут. Подробные инструкции для проверки заданий мы пришлём позже.

ВАЖНО! Чтобы задание было удобнее проверять, необходимо сдать на Stepik два файла: файл в формате .ipynb и файл в формате .pdf. Файл .pdf можно получить, открыв File->Print и выбрать "Save as PDF". Аналогичный способ есть и в Jupyter.

In [0]:

```
import torch
from torch import nn
from torch import functional as F
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
```

Дисклеймер про CrossEntropyLoss и NLLLoss

Обычно в PyTorch не нужно делать Softmax как последний слой модели.

- Если Вы используете NLLLoss, то ему на вход надо давать лог вероятности, то есть выход слоя LogSoftmax. (Просто результат софтмакса, к которому применен логарифм)
- Если Вы используете CrossEntropyLoss, то применение LogSoftmax уже включено внутрь лосса, поэтому ему на вход надо подавать просто выход обычного линейного слоя без активации. По сути CrossEntropyLoss = LogSoftmax + NLLLoss

Зачем такие сложности, чтобы посчитать обычную кросс энтропию, которую мы использовали как лосс еще в логистической регрессии? Дело в том, что нам в любом случае придется взять логарифм от результатов софтмакса, а если делать это одной функцией, то можно сделать более устойчивую реализацию, которая даст меньшую вычислительную погрешность.

Таким образом, если у вас в конце сети, решающей задачу классификации, стоит просто линейный слой без активации, то вам нужно использовать CrossEntropy. В этой домашке везде используется лосс CrossEntropy

Задание 1. Создайте генератор батчей.

В этот раз мы хотим сделать генератор, который будет максимально похож на то, что используется в реальном обучении.

С помощью numpy вам нужно перемешать исходную выборку и выбирать из нее батчи размером batch_size, если размер выборки не делился на размер батча, то последний батч должен иметь размер меньше batch_size и состоять просто из всех оставшихся объектов. Возвращать нужно в формате (X_batch, y_batch). Необходимо написать именно генератор, то есть вместо return использовать yield.

Хорошая статья про генераторы: https://habr.com/ru/post/132554/ (https://habr.com/ru/post/132554/)

Ответ на задание - код

In [0]:

```
def batch_generator(X, y, batch_size):

    # basic info about X
    n = len(X)
    num_full_batches = n // batch_size
    rest = n % batch_size

    # shuffle the data before yielding batches
    np.random.seed(42)
    perm = np.random.permutation(len(X))
    X_shuffled = X[perm, :]
    y_shuffled = y[perm]

    # yielding num_full_batches, each of size=batch_size
    for i in range(num_full_batches):
        yield X_shuffled[i * batch_size : (i+1) * batch_size, :], \
            y_shuffled[i * batch_size : (i+1) * batch_size]

    # yelding the last (not full) batch if needed
    if rest != 0:
        yield X_shuffled[-rest:, :], y_shuffled[-rest:]
```

Попробуем потестировать наш код

In [0]:

```
from inspect import isgeneratorfunction
assert isgeneratorfunction(batch_generator), "batch_generator должен \
    быть генератором! В условии есть ссылка на доки"

X = np.array([
              [1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]
])
y = np.array([
              1, 2, 3
])

# Проверим shape первого батча
iterator = batch_generator(X, y, 2)
X_batch, y_batch = next(iterator)
assert X_batch.shape == (2, 3), y_batch.shape == (2,)
assert np.allclose(X_batch, X[:2]), np.allclose(y_batch, y[:2])

# Проверим shape последнего батча (их всего два)
X_batch, y_batch = next(iterator)
assert X_batch.shape == (1, 3), y_batch.shape == (1,)
assert np.allclose(X_batch, X[2:]), np.allclose(y_batch, y[2:])

# Проверим, что итерации закончились
iter_ended = False
try:
    next(iterator)
except StopIteration:
    iter_ended = True
assert iter_ended

# Еще раз проверим то, сколько батчей создает итератор
X = np.random.randint(0, 100, size=(1000, 100))
y = np.random.randint(-1, 1, size=(1000, 1))
num_iter = 0
for _ in batch_generator(X, y, 3):
    num_iter += 1
assert num_iter == (1000 // 3 + 1)
```

Задание 2. Обучите модель для классификации звезд

Загрузите датасет из файла sky_data.csv, разделите его на train/test и обучите на нем нейронную сеть (архитектура ниже). Обучайте на батчах с помощью оптимизатора Adam, lr подберите сами, пробуйте что-то вроде 1e-2

Архитектура:

1. Dense Layer с relu активацией и 50 нейронами
2. Dropout 80% (если другой keep rate дает сходимость лучше, то можно изменить) (попробуйте 50%)
3. BatchNorm
4. Dense Layer с relu активацией и 100 нейронами
5. Dropout 80% (если другой keep rate дает сходимость лучше, то можно изменить) (попробуйте для разнообразия 50%)
6. BatchNorm
7. Выходной Dense слой с количеством нейронов, равному количеству классов

Лосс - CrossEntropy.

В датасете классы представлены строками, поэтому классы нужно закодировать. Для этого в строчке ниже объявлен dict, с помощью него и функции map превратите столбец с таргетом в целое число. Кроме того, за вас мы выделили признаки, которые нужно использовать.

Загрузка и обработка данных

In [0]:

```
feature_columns = ['ra', 'dec', 'u', 'g', 'r', 'i', 'z', 'run', 'camcol', 'fiel
d']
target_column = 'class'

target_mapping = {
    'GALAXY': 0,
    'STAR': 1,
    'QSO': 2
}
```

In [54]:

```
data = pd.read_csv('https://drive.google.com/uc?id=1K-8CtATw6Sv7k2dXco1fL5MAhTbK
tIH3')
data['class'].value_counts()
```

Out[54]:

```
GALAXY    4998
STAR      4152
QSO        850
Name: class, dtype: int64
```

In [55]:

```
data.head()
```

Out[55]:

	objid	ra	dec	u	g	r	i	z	run
0	1.237650e+18	183.531326	0.089693	19.47406	17.04240	15.94699	15.50342	15.22531	752
1	1.237650e+18	183.598371	0.135285	18.66280	17.21449	16.67637	16.48922	16.39150	752
2	1.237650e+18	183.680207	0.126185	19.38298	18.19169	17.47428	17.08732	16.80125	752
3	1.237650e+18	183.870529	0.049911	17.76536	16.60272	16.16116	15.98233	15.90438	752
4	1.237650e+18	183.883288	0.102557	17.55025	16.26342	16.43869	16.55492	16.61326	752

In [56]:

```
# Extract Features
X = data[feature_columns]
# Extract target
y = data[target_column].copy()

# encode target with target_mapping
y.replace(target_mapping, inplace=True)
y.value_counts() # check the results
```

Out[56]:

```
0    4998
1    4152
2     850
Name: class, dtype: int64
```

Нормализация фичей

In [57]:

```
print("Mean by col: \n", X.mean(axis=0), "\n")
print("Std by col: \n", X.std(axis=0))
```

```
Mean by col:
 ra         175.529987
dec         14.836148
u           18.619355
g           17.371931
r           16.840963
i           16.583579
z           16.422833
run        981.034800
camcol       3.648700
field      302.380100
dtype: float64

Std by col:
 ra          47.783439
dec         25.212207
u            0.828656
g            0.945457
r            1.067764
i            1.141805
z            1.203188
run        273.305024
camcol       1.666183
field      162.577763
dtype: float64
```

In [0]:

```
# Просто вычтите среднее и поделите на стандартное отклонение (с помощью пандас).
# Также преобразуйте всё в np.array
X = ((X - X.mean(axis=0)) / X.std(axis=0, ddof=0)).to_numpy()
y = y.to_numpy()
```

In [59]:

```
print("Mean by col: \n", np.round(X.mean(axis=0), 5), "\n")
print("Std by col: \n", X.std(axis=0))
```

```
Mean by col:
 [ 0. -0.  0.  0.  0.  0. -0. -0.  0. -0.]

Std by col:
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

In [0]:

```
assert type(X) == np.ndarray and \
    type(y) == np.ndarray, 'Проверьте, что получившиеся массивы являются np.ndarray'
assert np.allclose(y[:5], [1,1,0,1,1])
assert X.shape == (10000, 10)
assert np.allclose(X.mean(axis=0), np.zeros(10)) and \
    np.allclose(X.std(axis=0), np.ones(10)), 'Данные не отнормированы'
```

Обучение

In [0]:

```
# Split train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
# Превратим данные в тензоры, чтобы потом было удобнее
X_train = torch.FloatTensor(X_train)
y_train = torch.LongTensor(y_train)
X_test = torch.FloatTensor(X_test)
y_test = torch.LongTensor(y_test)
```

In [62]:

```
# initial number of features
len(feature_columns)
```

Out[62]:

```
10
```

In [63]:

```
# important to know the data types of the objects we will be working with
y_test.dtype, X_test.dtype
```

Out[63]:

```
(torch.int64, torch.float32)
```

Хорошо, данные мы подготовили, теперь надо объявить модель.

Обучайте на батчах с помощью оптимизатора Adam, lr подберите сами, пробуйте что-то вроде 1e-2

Архитектура:

1. Dense Layer с relu активацией и 50 нейронами
2. Dropout 80% (если другой keep rate дает сходимость лучше, то можно изменить) (попробуйте 50%)
3. BatchNorm
4. Dense Layer с relu активацией и 100 нейронами
5. Dropout 80% (если другой keep rate дает сходимость лучше, то можно изменить) (попробуйте для разнообразия 50%)
6. BatchNorm
7. Выходной Dense слой с количеством нейронов, равному количеству классов

Лосс - CrossEntropy.

In [0]:

```
torch.manual_seed(42)
np.random.seed(42)
model = nn.Sequential(
    nn.Linear(in_features=len(feature_columns), out_features=50),
    nn.ReLU(),
    nn.Dropout(p=0.2),
    nn.BatchNorm1d(50),

    nn.Linear(in_features=50, out_features=100),
    nn.ReLU(),
    nn.Dropout(p=0.2),
    nn.BatchNorm1d(num_features=100),

    nn.Linear(in_features=100, out_features=3)
)

loss_fn = nn.CrossEntropyLoss()

learning_rate = 1e-2
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Обучающий цикл

In [0]:

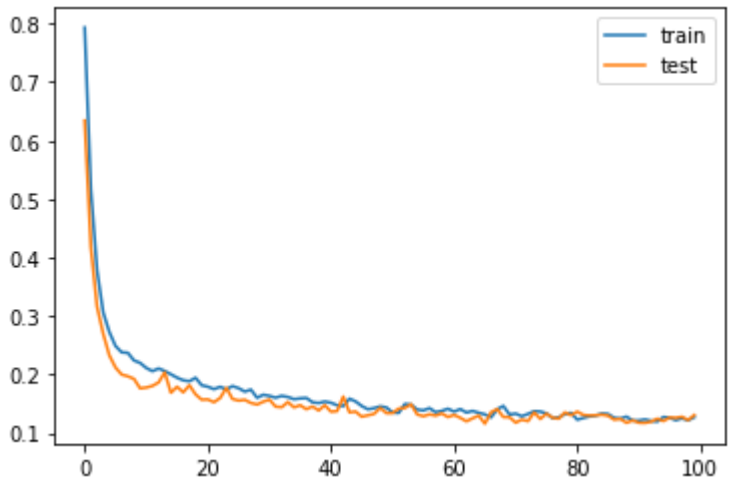
```
def train(X_train, y_train, X_test, y_test, num_epoch):
    train_losses = []
    test_losses = []

    for i in range(num_epoch):
        epoch_train_losses = []

        for X_batch, y_batch in batch_generator(X_train, y_train, 500):
            # Дропаут работает по-разному во время обучения и реального предсказания
            # Поэтотму нужно включать и выключать режим обучения (команда ниже)
            model.train(True)

            # Посчитаем предсказание и лосс
            pred_batch = model(X_batch)
            batch_loss = loss_fn(pred_batch, y_batch)

            # зануляем градиент
            optimizer.zero_grad()

            # backward
            batch_loss.backward()

            # ОБНОВЛЯЕМ веса
            optimizer.step()

            # Запишем число (не тензор) в наши батчевые лоссы
            epoch_train_losses.append(batch_loss.item())

        train_losses.append(np.mean(epoch_train_losses))

        # Теперь посчитаем лосс на тесте
        model.train(False)
        with torch.no_grad():
            # Сюда опять же надо положить именно число равное лоссу на всем тест датасет
e
            pred_test = model(X_test)
            test_loss = loss_fn(pred_test, y_test)

            test_losses.append(test_loss.item())

    return train_losses, test_losses
```

In [0]:

```
def check_loss_decreased():
    print("На графике сверху, точно есть сходимость? Точно-точно? [Да/Нет]")
    s = input()
    if s.lower() == 'да':
        print("Хорошо!")
    else:
        raise RuntimeError("Можно уменьшить дропаут, уменьшить lr, \
                            поправить архитектуру, etc")
```

In [67]:

```
#Подберите количество эпох так, чтобы график loss сходился
train_losses, test_losses = train(X_train, y_train,
                                  X_test, y_test, num_epoch=100)
plt.plot(range(len(train_losses)), train_losses, label='train')
plt.plot(range(len(test_losses)), test_losses, label='test')
plt.legend()
plt.show()

check_loss_decreased()
assert train_losses[-1] < 0.3 and test_losses[-1] < 0.3
```



На графике сверху, точно есть сходимость? Точно-точно? [Да/Нет]
Да
Хорошо!

Вычислите accuracy получившейся модели на train и test

In [0]:

```
# setting the model into the evaluation mode (for Dropout and BatchNorm)
model.eval()

# saving the preidcted labels for train and test sets
# torch.max returns a tuple (values, indices), so we subset the indices with [1]
train_pred_labels = torch.max(model.forward(X_train).detach(), 1)[1]
test_pred_labels = torch.max(model.forward(X_test).detach(), 1)[1]
```

In [69]:

```
train_pred_labels
```

Out[69]:

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

In [70]:

```
y_train
```

Out[70]:

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

In [0]:

```
from sklearn.metrics import accuracy_score
```

In [72]:

```
train_acc = accuracy_score(y_true=y_train, y_pred=train_pred_labels)
test_acc = accuracy_score(y_true=y_test, y_pred=test_pred_labels)

assert train_acc > 0.9, "Если уж классифицировать звезды, которые уже видел, \
    то не хуже, чем в 90% случаев"
assert test_acc > 0.9, "Новые звезды тоже надо классифицировать хотя бы в 90% случаев"

print("Train accuracy: {}\nTest accuracy: {}".format(train_acc, test_acc))
```

```
Train accuracy: 0.9697333333333333
Test accuracy: 0.9608
```

Задание 3. Исправление ошибок в архитектуре

Только что вы обучили полносвязную нейронную сеть. Теперь вам предстоит проанализировать архитектуру нейронной сети ниже, исправить в ней ошибки и обучить её с помощью той же функции train. Пример исправления ошибок есть в семинаре Григория Лелейтнера.

Будьте осторожнее и убедитесь, что перед запуском train вы вновь переопределили все необходимые внешние переменные (train обращается к глобальным переменным, в целом так делать не стоит, но сейчас это было оправдано, так как иначе нам пришлось бы передавать порядка 7-8 аргументов).

Чтобы у вас получилась такая же архитектура, как у нас, и ответы совпали, давайте определим некоторые правила, как исправлять ошибки:

1. Если вы видите лишний нелинейный слой, который стоит не на своем месте, просто удалите его. (не нужно добавлять новые слои, чтобы сделать постановку изначального слоя разумной. Удалять надо самый последний слой, который все портит. Для линейных слоев надо что-то исправить, а не удалить его)
2. Если у слоя нет активации, то добавьте ReLU или другую подходящую активацию
3. Если что-то не так с learning_rate, то поставьте 1e-2
4. Если что-то не так с параметрами, считайте первый параметр, который появляется, как верный (т.е. далее в сети должен использоваться он).
5. Ошибки могут быть и в полносвязных слоях.
6. Любые другие проблемы решаются более менее однозначно, если же у вас есть серьезные сомнения, то напишите в беседу в телеграме и пинганите меня @runfme

Задача все та же - классификация небесных объектов на том же датасете. После исправления сети вам нужно обучить ее.

Ответ на задачу - средний лосс на тестовом датасете

In [0]:

```
torch.manual_seed(42)
np.random.seed(42)
# WRONG ARCH
model = nn.Sequential(
    nn.Dropout(p=0.5),
    nn.Linear(6, 50),
    nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(100, 200),
    nn.Softmax(),
    nn.Linear(200, 200),
    nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(200, 3),
    nn.Dropout(p=0.5)
)


loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-100)
```

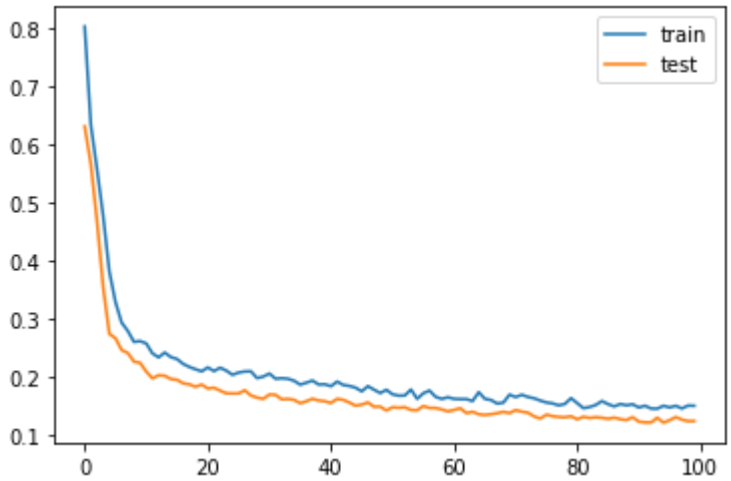
In [0]:

```
# RIGHT ARCH
torch.manual_seed(42)
np.random.seed(42)

model = nn.Sequential(
    nn.Linear(len(feature_columns), 100),
    nn.ReLU(),
    nn.Dropout(p=0.5),

    nn.Linear(100, 200),
    nn.ReLU(),

    nn.Linear(200, 200),
    nn.ReLU(),
    nn.Dropout(p=0.5),

    nn.Linear(200, 3),
)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
```

Обучите и протестируйте модель так же, как вы это сделали в задаче 2. Вычислите accuracy.

In [75]:

```
#Подберите количество эпох так, чтобы график loss сходился
train_losses, test_losses = train(X_train, y_train,
                                  X_test, y_test, num_epoch=100)
plt.plot(range(len(train_losses)), train_losses, label='train')
plt.plot(range(len(test_losses)), test_losses, label='test')
plt.legend()
plt.show()

check_loss_decreased()
assert train_losses[-1] < 0.3 and test_losses[-1] < 0.3
```



На графике сверху, точно есть сходимость? Точно-точно? [Да/Нет]
Да
Хорошо!

In [0]:

```
# setting the model into the evaluation mode (for Dropout and BatchNorm)
model.eval()

# saving the preidcted labels for train and test sets
# torch.max returns a tuple (values, indices) = (max, argmax), so we take the
# indices by subsetting to [1]
train_pred_labels = torch.max(model.forward(X_train).detach(), 1)[1]
test_pred_labels = torch.max(model.forward(X_test).detach(), 1)[1]
```

In [77]:

```
from sklearn.metrics import accuracy_score

train_acc = accuracy_score(y_true=y_train, y_pred=train_pred_labels)
test_acc = accuracy_score(y_true=y_test, y_pred=test_pred_labels)

assert train_acc > 0.9, "Если уж классифицировать звезды, которые уже видел, \
    то не хуже, чем в 90% случаев"
assert test_acc > 0.9, "Новые звезды тоже надо классифицировать хотя бы в 90% случаев"

print("Train accuracy: {}\nTest accuracy: {}".format(train_acc, test_acc))
```

```
Train accuracy: 0.9670666666666666
Test accuracy: 0.9588
```

Ответ на задачу - средний лосс на тестовом датасете:

In [78]:

```
# средний лосс на тестовом датасете за все эпохи обучения
np.mean(test_losses)
```

Out[78]:

```
0.1717549003660679
```

In [79]:

```
# средний лосс на тестовом датасете после обучения
test_losses[-1]
```

Out[79]:

```
0.12537933886051178
```

Задание 4. Stack layers

Давайте посмотрим, когда добавление перестает улучшать метрики. Увеличивайте блоков из слоев в сети, пока минимальный лосс на тестовом датасете за все время обучения не перестанет уменьшаться (20 эпох).

Стоит помнить, что нельзя переиспользовать слои с предыдущих обучений, потому что они уже будут с подобранными весами.

Чтобы получить воспроизводимость и идентичный нашему ответ, надо объявлять все слои в порядке, в котором они применяются внутри модели. Это важно, если вы будете собирать свою модель из частей. Перед объявлением этих слоев по порядку напишите

```
torch.manual_seed(42)
np.random.seed(42)
```

При чем каждый раз, когда вы заново создаете модель, перезадавайте random seeds

Опитимизатор - Adam(lr=1e-2)

In [0]:

```
# МОДЕЛЬ ДЛЯ ПРИМЕРА, НА САМОМ ДЕЛЕ ВАМ ПРИДЕТСЯ СОЗДАВАТЬ НОВУЮ
# МОДЕЛЬ ДЛЯ КАЖДОГО КОЛИЧЕСТВА БЛОКОВ
model = nn.Sequential(
    nn.Linear(len(feature_columns), 100),
    nn.ReLU(),
    nn.Dropout(p=0.5),
    # Начало блока, который надо вставалять много раз
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.BatchNorm1d(100),
    # Конец блока
    nn.Linear(100, 3)
    # Блока Softmax нет, поэтому нам нужно использовать лосс - CrossEntropyLoss
)
```

In [0]:

```
# Вы уже многое умеете, поэтому теперь код надо написать самому
# Идея - разделить модель на части.
# Вначале создать head часть как Sequential модель, потом в цикле создать Sequential
# модели, которые представляют из себя блоки, потом создать tail часть тоже как Sequenti
al,
# а потом объединить их в одну Sequential модель
# вот таким кодом: nn.Sequential(header, *blocks, footer)

# Важная идея тут состоит в том, что модели могут быть частями других моделей)
```

In [0]:

```
# choose up till which number of blocks to try in the model
max_num_blocks = 10
```

Решение с конструкторами

In [0]:

```
def head_constructor():
    head = nn.Sequential(
        nn.Linear(len(feature_columns), 100),
        nn.ReLU(),
        nn.Dropout(p=0.5),
    )
    return head

# constracting one block to reuse
def block_constructor():
    block = nn.Sequential(
        nn.Linear(100, 100),
        nn.ReLU(),
        nn.BatchNorm1d(100),
    )
    return block

# constracting the tail
def tail_constructor():
    return nn.Sequential(nn.Linear(100, 3))
```

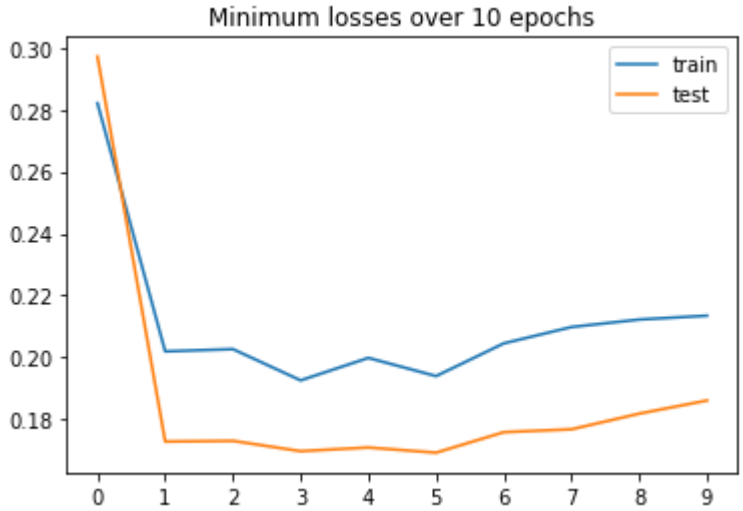
In [84]:

```
# record the losses here
min_test_losses = []
min_train_losses = []

for num_blocks in range(max_num_blocks):

    print("Fitting the model with {} blocks".format(num_blocks))

    # creating the model
    torch.manual_seed(42)
    np.random.seed(42)
    model = nn.Sequential(head_constructor(),
                          *[block_constructor() for _ in range(num_blocks)],
                          tail_constructor())

    # loss and optimimzer
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)

    # training the model
    train_losses, test_losses = train(X_train, y_train,
                                      X_test, y_test, num_epoch=20)

    min_test_losses.append(np.min(test_losses))
    min_train_losses.append(np.min(train_losses))

    # # plotting train and test loss curves for each model
    # plt.plot(range(len(train_losses)), train_losses, label='train')
    # plt.plot(range(len(test_losses)), test_losses, label='test')
    # plt.title("Loss curves for {} blocks".format(num_blocks))
    # plt.legend()
    # plt.show()
```

```
Fitting the model with 0 blocks
Fitting the model with 1 blocks
Fitting the model with 2 blocks
Fitting the model with 3 blocks
Fitting the model with 4 blocks
Fitting the model with 5 blocks
Fitting the model with 6 blocks
Fitting the model with 7 blocks
Fitting the model with 8 blocks
Fitting the model with 9 blocks
```

In [86]:

```
# choose up to which number of losses to plot to
# needed not to retrain the model just to get a different plot
up_to = min(max_num_blocks, 15)

plt.plot(np.arange(up_to), min_train_losses[:up_to], label='train')
plt.plot(np.arange(up_to), min_test_losses[:up_to], label='test')

plt.xticks(np.arange(up_to))
plt.title("Minimum losses over {} epochs".format(up_to))
plt.legend()
plt.show()

print("min_train_losses: \n", np.round(min_train_losses[:up_to], 4))
print("Number of blocks for minimal train loss: ",
      min_train_losses.index(min(min_train_losses)), '\n')

print("min_test_losses: \n", np.round(min_test_losses[:up_to], 4))
print("Number of blocks for minimal test loss: ",
      min_test_losses.index(min(min_test_losses)))
```




```
min_train_losses:
 [0.282  0.202  0.2027 0.1926 0.1998 0.194  0.2045 0.2098 0.2122 0.2
135]
Number of blocks for minimal train loss:  3

min_test_losses:
 [0.2971 0.1729 0.1731 0.1698 0.1709 0.1693 0.1759 0.1768 0.1819 0.1
861]
Number of blocks for minimal test loss:  5
```

Задание 5. Сделайте выводы

Начиная с какого количества блоков минимальный лосс за время обучения увеличивается?
Почему лишнее количество блоков не помогает модели?

Заметим, при увеличении числа блоков модель становится сложнее и сильнее.

Начиная с какого количества блоков минимальный лосс за время обучения увеличивается?

Модель с 5 блоками на 20 эпохах показывает лчший результат на тестовой выборке, начиная с 5 блоков минимальный лосс за время обучения чаще всего увеличивается.

(С 1 блока до 2 блоков, и с 3 блоков до 4 блоков идет небольшое увеличение лосса, но минимальный все же достигается на 5 блоках).

Почему лишнее количество блоков не помогает модели?

Это может происходить по 2 причинам:

1. Более сильная модель может переобучиться на train set, что приводит к увеличению ошибки на test set. Но эта причина не ввыдерживает критики, когда мы смотрим на график min_train_losses, потому что train loss показывает лучший результат с тремя блоками, и начинает расти.
2. Вторая возможная причина, это то, что при увеличении количества блоков, наша модель становится сильнее и ей уже не хватает 20 эпох для обучения. При увеличении количества эпох до 100, для test set и для train set минимальный лосс наблюдается для моделей с 2, и 3 блоками соответственно. При этом лосс падает в принципе: колебания лосса теперь между 0.11 и 0.14 для моделей с 1-10 блоками по сравнению с колебаниями между 0.16 и 0.18 для этих е моделей треннированных на 20 эпохах.

Вывод: Сложность данных соответствует сложности модели с 2-5 такими блоками, как даны в задании. При увеличении количества блоков больше 5 мы не добавляем модели ничего полезного, а только увеличивем количество эпох, нужных чтобы обучить эти модели до такого-же результата (большие и сложные модели нужно дольше учить). Для улучшения качества можно попробовать изменить блок (добавить Dropout или изменить количество нейронов в слое).