CS5242_Assignment_1

February 14, 2018

1 Introduction

ASSIGNMENT DEADLINE: 4 MAR 2018 (SUN) 17:00PM

In this assignment we will be coding the building blocks for the convolutional neural network and putting them together to train a CNN on the MNIST dataset.

Attention: Only python3 will be allowed to use in this assignment. And we use `numpy` to store and caculate data and parameters. You do not need a GPU to for this assignment. CPU is enough. To run this Jupyter notebook, you need to install the depedent libraries in requiremets.txt via pip (or pip3). Note: keras version should be >=2.1.2. Please do not run this whole file before you implement all the codes. Otherwise it will occur some error.

For each layer we will implement a forward and a backward function. The forward function will receive inputs and will return the outputs of this layer(loss layer will be a little different), like this:

```
def forward(self, inputs):
  """ Receive inputs and return output"""
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  outputs = # the outputs

  return outputs
```

The backward pass will receive upstream derivatives and inputs, and will return gradients with respect to the inputs. Gradients for weights or bias will be stored in parameters in this layer , like this:

```
def backward(self, in_grads, inputs):
  """
  Receive derivative of loss with respect to outputs,
  and compute derivative with respect to inputs.
  """
  # Use values in cache to compute derivatives
  out_grads = # Derivative of loss with respect to inputs
  self.w_grad = # Derivative of loss with respect to self.weights
  self.b_grad = # Derivative of loss with respect to self.bias

  return out_grads
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

This iPython notebook serves to:

- explain the questions
- explain the function APIs and implementation examples (like `ReLU`)
- provide helper functions to piece functions together and check your code

2 ReLU layer

A convolution layer is usually followed by a non-linear activation function. We will provide the functions `forward` and `backward` of class `ReLU` in `layers.py` as an implementation example. Read through the function code and make sure you understand the derivation. Besides, we will explain the implementation of `ReLU` after `Convolution` using formula. You need to write down other layers' formulations in your reports.

2.1 Forward Formulation

Given input $x \in R^{B\times C\times H\times W}$ (B:batch size, C: number of channel, H: input height, W: input width), output $y \in R^{B\times C\times H\times W}$ will be caculated like this:

$$y = indicator(x) \times x$$

Here, $indicator(x)$ return the same size of input x, comparing x with 0 element-wisely. If $x_{i,j,k,l} \geq 0$ return $z_{i,j,k,l} = 1$. And the multiplication is also element-wise. If the input x has only 2 dimensions, i.e. the batch dimension and the feature dimension, e.g. after the FC layer, the subscripts j, k, l in the formula are merged into one j.

2.2 Backward Formulation

Given input $x \in R^{B\times C\times H\times W}$ (B:batch size, C: number of channel, H: input height, W: input width) and gradients to output of this layer $dy \in R^{B\times C\times H\times W}$, gradients to input dx will be caculated like this:

$$dx = indicator(x) \times dy$$

3 Covolution Layer

In the file `layers.py`, the class `Convolution` will be initialized with `conv_params`, `initializer` and `name`, shown as below:

```
def __init__(self, conv_params, initializer=Guassian(), name='conv'):
        super(Convolution, self).__init__(name=name)
        self.trainable = True
        self.kernel_h = conv_params['kernel_h'] # height of kernel
        self.kernel_w = conv_params['kernel_w'] # width of kernel
        self.pad = conv_params['pad']
```

```
        self.stride = conv_params['stride']
        self.in_channel = conv_params['in_channel']
        self.out_channel = conv_params['out_channel']

        self.weights = initializer.initialize((self.out_channel, self.in_channel, self.kernel_h,
        self.bias = np.zeros((self.out_channel))

        self.w_grad = np.zeros(self.weights.shape)
        self.b_grad = np.zeros(self.bias.shape)
```

`conv_params` is a dictionary, containing these parameters:

- 'kernel_h': The height of kernel.
- 'kernel_w': The width of kernel.
- 'stride': The number of pixels between adjacent receptive fields in the horizontal and vertical directions.
- 'pad': The number of pixels padded to the bottom, top, left and right of each feature map. **Here,** `pad=2` **means a 2-pixel border of padded with zeros. So the total number of zeros for horizontal (or vertical) direction is 2*pad=4**.
- 'in_channel': The number of input channels.
- 'out_channel': The number of output channels.

`initializer` is an instance of Initializer class (leave it out right now)

3.1 Forward

In the file `layers.py`, implement the forward pass for a convolutional layer in the function `forward` of class `Convolution`.

The input consists of N data points, each with C channels, height H and width W. We convolve each input with K different kernels, where each filter spans all C channels and has height HH and width WW.

Input:

- inputs: Input data of shape (N, C, H, W)

WARNING: Please implement the matrix product method of convolution as shown in Lecture notes. The naive version of implementing a sliding window will be too slow when you try to train the whole CNN in later sections.

You can test your implementation by restarting jupyter notebook kernel and running the following:

```
In [1]: import numpy as np
        from layers import Convolution
        from utils.tools import rel_error

        import keras
        from keras import layers
        from keras import models
        from keras import optimizers
```

```
from keras import backend as K

import warnings
warnings.filterwarnings('ignore')

inputs = np.random.uniform(size=(10, 3, 30, 30))
params = { 'kernel_h': 5,
           'kernel_w': 5,
           'pad': 0,
           'stride': 2,
           'in_channel': inputs.shape[1],
           'out_channel': 64,
}
layer = Convolution(params)
out = layer.forward(inputs)

keras_model = keras.Sequential()
keras_layer = layers.Conv2D(filters=params['out_channel'],
                            kernel_size=(params['kernel_h'], params['kernel_w']),
                            strides=(params['stride'], params['stride']),
                            padding='valid',
                            data_format='channels_first',
                            input_shape=inputs.shape[1:])
keras_model.add(keras_layer)
sgd = optimizers.SGD(lr=0.01)
keras_model.compile(loss='mean_squared_error', optimizer='sgd')
weights = np.transpose(layer.weights, (2, 3, 1, 0))
keras_layer.set_weights([weights, layer.bias])
keras_out = keras_model.predict(inputs, batch_size=inputs.shape[0])
print('Relative error (<1e-6 will be fine): ', rel_error(out, keras_out))
```

```
/home/zero/anaconda3/envs/tensorflow/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarn
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
/home/zero/anaconda3/envs/tensorflow/lib/python3.6/importlib/_bootstrap.py:219: RuntimeWarning:
  return f(*args, **kwds)


Relative error (<1e-6 will be fine):  3.990660772975816e-07
```

3.2 Backward

Implement the backward pass for the convolution operation in the function `backward` of `Convolution` class in the file `layers.py`.

When you are done, restart jupyter notebook and run the following to check your backward pass with a numeric gradient check.

In gradient checking, to get an approximate gradient for a parameter, we vary that parameter by a small amount (while keeping rest of parameters constant) and note the difference in the net-

work loss. Dividing the difference in network loss by the amount we varied the parameter gives us an approximation for the gradient. We repeat this process for all the other parameters to obtain our numerical gradient. Note that gradient checking is a slow process (2 forward propagations per parameter) and should only be used to check your backpropagation!

More links on gradient checking:
http://ufldl.stanford.edu/tutorial/supervised/DebuggingGradientChecking/
https://www.coursera.org/learn/machine-learning/lecture/Y3s6r/gradient-checking

```
In [2]: from layers import Convolution
        import numpy as np
        from utils.check_grads import check_grads_layer

        batch = 10
        conv_params={
            'kernel_h': 3,
            'kernel_w': 3,
            'pad': 0,
            'stride': 2,
            'in_channel': 3,
            'out_channel': 10
        }
        in_height = 10
        in_width = 20
        out_height = 1+(in_height+2*conv_params['pad']-conv_params['kernel_h'])//conv_params['st
        out_width = 1+(in_width+2*conv_params['pad']-conv_params['kernel_w'])//conv_params['stri
        inputs = np.random.uniform(size=(batch, conv_params['in_channel'], in_height, in_width))
        in_grads = np.random.uniform(size=(batch, conv_params['out_channel'], out_height, out_wi
        conv = Convolution(conv_params)
        check_grads_layer(conv, inputs, in_grads)
```

```
<1e-8 will be fine
Gradients to inputs: 1.4932430945544107e-11
Gradients to weights:  5.64255171696083e-13
Gradients to bias:  8.642318016673506e-13
```

4 Dropout Layer

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

In the file `layers.py`, the class `FCLayer` will be initialized with `ratio`, `seed` and `name`, shown as below:

```
def __init__(self, ratio, name='dropout', seed=None):
        super(Dropout, self).__init__(name=name)
```

```
        self.ratio = ratio
        self.mask = None
        self.seed = seed
```

- `ratio`: The probability of setting a neuron to zero
- `seed`: Random seed to sample from inputs, so as to get mask. (default as None)

4.1 Forward

In the file `layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes. `p` refers to the probability of setting a neuron to zero. We will follow the Caffe convention where we multiply the outputs by `1/(1-p)` during training.

4.2 Backward

In the file `layers.py`, implement the backward pass for dropout. After doing so, restart jupyter notebook and run the following cell to numerically gradient-check your implementation.

```
In [7]: from layers import Dropout
        import numpy as np
        from utils.check_grads import check_grads_layer

        ratio = 0.1
        height = 10
        width = 20
        channel = 10
        np.random.seed(1234)
        inputs = np.random.uniform(size=(batch, channel, height, width))
        in_grads = np.random.uniform(size=(batch, channel, height, width))
        dropout = Dropout(ratio, seed=1234)
        dropout.set_mode(True)
        check_grads_layer(dropout, inputs, in_grads)
```

```
<1e-8 will be fine
Gradients to inputs: 3.977519800006909e-12
```

5 Pooling Layer

In the file `layers.py`, the class `Pooling` will be initialized with `pool_params`, and `name`, shown as below:

```
def __init__(self, pool_params, name='pooling'):
        super(Pooling, self).__init__(name=name)
        self.pool_type = pool_params['pool_type']
        self.pool_height = pool_params['pool_height']
        self.pool_width = pool_params['pool_width']
```

```
        self.stride = pool_params['stride']
        self.pad = pool_params['pad']
```

`pool_params` is a dictionary, containing these parameters:

- 'pool_type': The type of pooling, 'max' or 'avg'
- 'pool_h': The height of pooling kernel.
- 'pool_w': The width of pooling kernel.
- 'stride': The number of pixels between adjacent receptive fields in the horizontal and vertical directions.
- 'pad': The number of pixels that will be used to zero-pad the input in each x-y direction. **Here,** `pad=2` **means a 2-pixel border of padding with zeros**.

5.1 Forward

Implement the forward pass for the pooling operation in the function `forward` of class `Pooling` in the file `layers.py`.

You can test your implementation by restarting jupyter notebook kernel and running the following:

```
In [3]: import numpy as np
        from layers import Pooling
        from utils.tools import rel_error

        import keras
        from keras import layers
        from keras import models
        from keras import optimizers
        from keras import backend as K

        import warnings
        warnings.filterwarnings('ignore')

        inputs = np.random.uniform(size=(10, 3, 30, 30))
        params = { 'pool_type': 'max',
                   'pool_height': 5,
                   'pool_width': 5,
                   'pad': 0,
                   'stride': 2,
        }
        layer = Pooling(params)
        out = layer.forward(inputs)

        keras_model = keras.Sequential()
        keras_layer = layers.MaxPooling2D(pool_size=(params['pool_height'], params['pool_width']
                                          strides=params['stride'],
                                          padding='valid',
                                          data_format='channels_first',
                                          input_shape=inputs.shape[1:])
```

```
        keras_model.add(keras_layer)
        sgd = optimizers.SGD(lr=0.01)
        keras_model.compile(loss='mean_squared_error', optimizer='sgd')
        keras_out = keras_model.predict(inputs, batch_size=inputs.shape[0])
        print('Relative error (<1e-6 will be fine): ', rel_error(out, keras_out))
```

```
Relative error (<1e-6 will be fine):  7.61550641696389e-09
```

5.2 Backward

Implement the backward pass for the max-pooling operation in the function `backward` of class `Pooling` in the file `layers.py`.

Please make sure you have implemented both 'max' and 'avg' pooing in your codes. And then test the gradients by yourself.

6 FC Layer

FC layer (short for fully connected layer) is also called linear layer or dense layer.

In the file `layers.py`, the class `FCLayer` will be initialized with `in_features`, `out_features`, and `name`, shown as below:

```
def __init__(self, in_features, out_features, name='fclayer', initializer=Guassian()):
        super(FCLayer, self).__init__(name=name)
        self.trainable = True
        self.weights = initializer.initialize((in_features, out_features))
        self.bias = initializer.initialize(out_features)

        self.w_grad = np.zeros(self.weights.shape)
        self.b_grad = np.zeros(self.bias.shape)
```

- `in_features`: The number of inputs features
- `out_features`: The numbet of required outputs features

6.1 Forward

Implement the forward pass for the pooling operation in the function `forward` of class `FCLayer` in the file `layers.py`.

You can test your implementation by restarting jupyter notebook kernel and running the following:

```
In [5]: import numpy as np
        from layers import FCLayer
        from utils.tools import rel_error

        import keras
        from keras import layers
        from keras import models
```

```
        from keras import optimizers
        from keras import backend as K

        import warnings
        warnings.filterwarnings('ignore')

        inputs = np.random.uniform(size=(10, 20))

        layer = FCLayer(in_features=inputs.shape[1], out_features=100)
        out = layer.forward(inputs)

        keras_model = keras.Sequential()
        keras_layer = layers.Dense(100, input_shape=inputs.shape[1:], use_bias=True, kernel_init
        # print (len(keras_layer.get_weights()))
        keras_model.add(keras_layer)
        sgd = optimizers.SGD(lr=0.01)
        keras_model.compile(loss='mean_squared_error', optimizer='sgd')
        keras_layer.set_weights([layer.weights, layer.bias])
        keras_out = keras_model.predict(inputs, batch_size=inputs.shape[0])
        print('Relative error (<1e-6 will be fine): ', rel_error(out, keras_out))
```

```
Relative error (<1e-6 will be fine):  1.9344922343073387e-07
```

6.2 Backward

Implement the backward pass for the max-pooling operation in the function `backward` of class `FCLayer` in the file `layers.py`. Please test the gradients by yourself.

7 SoftmaxCrossEntropy Loss

We write Softmax and CrossEntropy together because it can avoid some numeric overflow problem.In the file `loss.py`, the class `SoftmaxCrossEntropy` will be initialized with `num_class`, shown as below:

```
def __init__(self, num_class):
        super(SoftmaxCrossEntropy, self).__init__()
        self.num_class = num_class
```

`num_class`: The number of category

7.1 Forward

Implement the forward pass for the pooling operation in the function `forward` of class `FCLayer` in the file `layers.py`.

You can test your implementation by restarting jupyter notebook kernel and running the following:

```
In [8]: import numpy as np
        from loss import SoftmaxCrossEntropy
        from utils.tools import rel_error

        import keras
        from keras import layers
        from keras import models
        from keras import optimizers
        from keras import backend as K

        import warnings
        warnings.filterwarnings('ignore')

        batch = 10
        num_class = 10
        inputs = np.random.uniform(size=(batch, num_class))
        targets = np.random.randint(num_class, size=batch)

        loss = SoftmaxCrossEntropy(num_class)
        out, _ = loss.forward(inputs, targets)

        keras_inputs = K.softmax(inputs)
        keras_targets = np.zeros(inputs.shape, dtype='int')
        for i in range(batch):
                keras_targets[i, targets[i]] = 1
        keras_out = K.mean(K.categorical_crossentropy(keras_targets, keras_inputs, from_logits=F
        print('Relative error (<1e-6 will be fine): ', rel_error(out, K.eval(keras_out)))

Relative error (<1e-6 will be fine):  9.377699188976416e-17
```

7.2 Backward

In the file `loss.py`, implement the backward pass for `SodtmaxCrossEntropy`. Please test the gradients by yourself.

8 Optimizer

In the file `optimizers.py`, there are 4 types of optimizer (`SGD`, `Adam`, `RMSprop` and `Adagrad`). You only need to implement the `update` function of `SGD`(mini-batch SGD with momentum) and `Adam`. These two types of optimizers are initialized like this:

```
class SGD(Optimizer):
    def __init__(self, lr=0.01, momentum=0, decay=0, sheduler_func = None):
        super(SGD, self).__init__(lr)
        self.momentum = momentum
        self.moments = None
        self.decay = decay
```

```
        self.sheduler_func = sheduler_func

class Adam(Optimizer):
    def __init__(self, lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0, sheduler_func=
        super(Adam, self).__init__(lr)
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.epsilon = epsilon
        self.decay = decay
        if not self.epsilon:
            self.epsilon = 1e-8
        self.moments = None
        self.accumulators = None
        self.sheduler_func = sheduler_func
```

For Both optimizers: - `lr`: The initial learning rate. - `decay`: The learning rate decay ratio - `sheduler_func`: Function to change learning rate with respect to iterations

For `SGD`: - `momentum`: The ratio of moments

For `Adam`: More details can be seen in reference.

For you reference: http://cs231n.github.io/neural-networks-3/#update

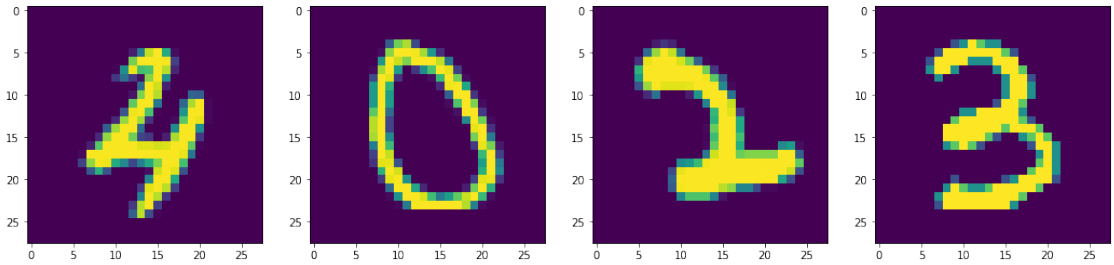
9 Train the net on full MNIST data

By training the `MNISTNet` for one epoch, you should achieve about 90% on the validation and test set. You may have to wait about 5 minutes for training to be completed.

```
In [10]: %matplotlib inline
         import matplotlib.pyplot as plt
         from applications import MNISTNet
         from loss import SoftmaxCrossEntropy, L2
         from optimizers import Adam
         from utils.datsets import MNIST
         import numpy as np

         mnist = MNIST()
         mnist.load()
         idx = np.random.randint(mnist.num_train, size=4)
         print('\nFour examples of training images:')
         img = mnist.x_train[idx][:,0,:,:]

         plt.figure(1, figsize=(18, 18))
         plt.subplot(1, 4, 1)
         plt.imshow(img[0])
         plt.subplot(1, 4, 2)
         plt.imshow(img[1])
         plt.subplot(1, 4, 3)
         plt.imshow(img[2])
```

```
        plt.subplot(1, 4, 4)
        plt.imshow(img[3])

start download mnist dataset...
Number of training images:  48000
Number of validation images:  12000
Number of testing images:  10000

Four examples of training images:
```

Out[10]: <matplotlib.image.AxesImage at 0x7f080c035438>

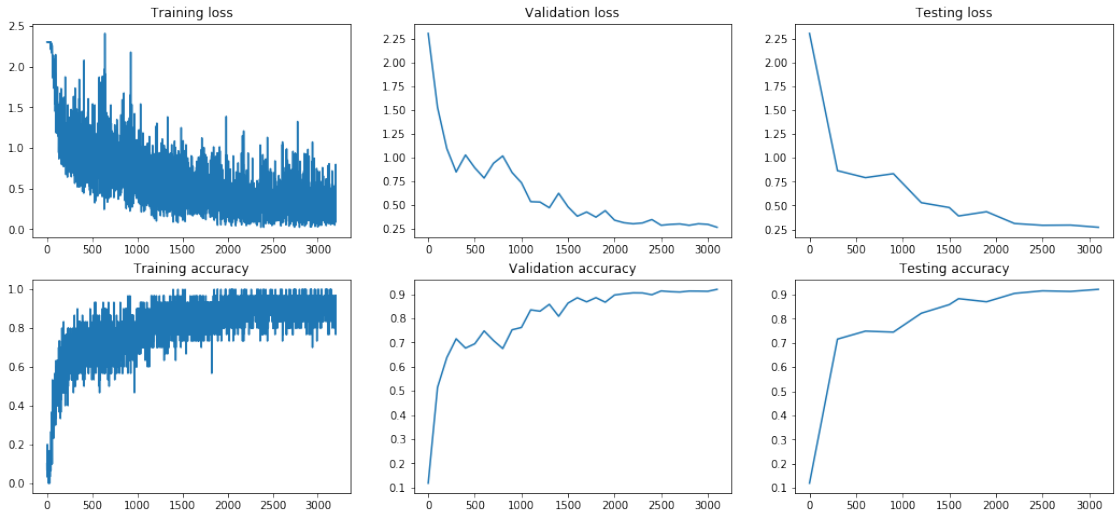


```
In [11]: model = MNISTNet()
         loss = SoftmaxCrossEntropy(num_class=10)

         # define your learning rate sheduler
         def func(lr, iteration):
             if iteration % 1000 ==0:
                 return lr*0.5
             else:
                 return lr

         adam = Adam(lr=0.001, decay=0,  sheduler_func = func)
         l2 = L2(w=0.001) # L2 regularization with lambda=0.001
         model.compile(optimizer=adam, loss=loss, regularization=l2)
         train_results, val_results, test_results = model.train(
             mnist,
             train_batch=30, val_batch=1000, test_batch=1000,
             epochs=2,
             val_intervals=100, test_intervals=300, print_intervals=100)

Epoch 0:
Test accuracy=0.11870, loss=2.30259
Validation accuracy: 0.11708, loss: 2.30259
Iteration 0:        accuracy=0.10000, loss=2.30259, regularization loss= 0.005269938172136857
Validation accuracy: 0.51458, loss: 1.52543
```

```
Iteration 100:        accuracy=0.40000, loss=1.59506, regularization loss= 0.03260591173363043
Validation accuracy: 0.63808, loss: 1.09270
Iteration 200:        accuracy=0.73333, loss=0.87318, regularization loss= 0.047275163161484376
Test accuracy=0.71590, loss=0.86478
Validation accuracy: 0.71575, loss: 0.84772
Iteration 300:        accuracy=0.70000, loss=0.86414, regularization loss= 0.04760929372624637
Validation accuracy: 0.67725, loss: 1.02614
Iteration 400:        accuracy=0.56667, loss=1.09188, regularization loss= 0.047217153027193266
Validation accuracy: 0.69517, loss: 0.89132
Iteration 500:        accuracy=0.80000, loss=0.76418, regularization loss= 0.06032709938373698
Test accuracy=0.74910, loss=0.79160
Validation accuracy: 0.74842, loss: 0.78370
Iteration 600:        accuracy=0.80000, loss=0.65367, regularization loss= 0.05752137141049369
Validation accuracy: 0.70867, loss: 0.93868
Iteration 700:        accuracy=0.80000, loss=0.66310, regularization loss= 0.05035930428183589
Validation accuracy: 0.67525, loss: 1.01610
Iteration 800:        accuracy=0.73333, loss=0.75446, regularization loss= 0.04692606939121045
Test accuracy=0.74500, loss=0.83317
Validation accuracy: 0.75317, loss: 0.84184
Iteration 900:        accuracy=0.76667, loss=0.78753, regularization loss= 0.04913583978137451
Validation accuracy: 0.76258, loss: 0.73549
Iteration 1000:        accuracy=0.90000, loss=0.42376, regularization loss= 0.05007358127576435
Validation accuracy: 0.83567, loss: 0.53591
Iteration 1100:        accuracy=0.80000, loss=0.67582, regularization loss= 0.03358499642914821
Test accuracy=0.82320, loss=0.52936
Validation accuracy: 0.83008, loss: 0.53189
Iteration 1200:        accuracy=0.86667, loss=0.34392, regularization loss= 0.03338925115793336
Validation accuracy: 0.85867, loss: 0.47215
Iteration 1300:        accuracy=0.86667, loss=0.53032, regularization loss= 0.03380768441804182
Validation accuracy: 0.80908, loss: 0.62410
Iteration 1400:        accuracy=0.73333, loss=0.88796, regularization loss= 0.03447974632892434
Test accuracy=0.85890, loss=0.47847
Validation accuracy: 0.86367, loss: 0.48224
Iteration 1500:        accuracy=0.76667, loss=0.36684, regularization loss= 0.03502458572614516
Epoch 1:
Test accuracy=0.88360, loss=0.38989
Validation accuracy: 0.88600, loss: 0.38295
Iteration 0:        accuracy=0.93333, loss=0.27179, regularization loss= 0.03556883529476457
Validation accuracy: 0.86892, loss: 0.42753
Iteration 100:        accuracy=0.86667, loss=0.46303, regularization loss= 0.036594178444966645
Validation accuracy: 0.88633, loss: 0.37325
Iteration 200:        accuracy=0.80000, loss=0.49726, regularization loss= 0.037809383125557074
Test accuracy=0.87070, loss=0.43465
Validation accuracy: 0.86742, loss: 0.44171
Iteration 300:        accuracy=0.90000, loss=0.25347, regularization loss= 0.03926519190481703
Validation accuracy: 0.89733, loss: 0.34161
Iteration 400:        accuracy=0.83333, loss=0.29950, regularization loss= 0.04029422202574584
Validation accuracy: 0.90242, loss: 0.31498
```

```
Iteration 500:        accuracy=0.93333, loss=0.46144, regularization loss= 0.04043127373961942
Test accuracy=0.90550, loss=0.31126
Validation accuracy: 0.90625, loss: 0.30370
Iteration 600:        accuracy=0.90000, loss=0.26672, regularization loss= 0.04095590661628099
Validation accuracy: 0.90583, loss: 0.31266
Iteration 700:        accuracy=0.90000, loss=0.28558, regularization loss= 0.0410743016717501
Validation accuracy: 0.89833, loss: 0.34832
Iteration 800:        accuracy=0.86667, loss=0.24720, regularization loss= 0.041437681955690496
Test accuracy=0.91610, loss=0.29238
Validation accuracy: 0.91425, loss: 0.28816
Iteration 900:        accuracy=0.83333, loss=0.40230, regularization loss= 0.04170566862922389
Validation accuracy: 0.91117, loss: 0.29751
Iteration 1000:        accuracy=0.93333, loss=0.59547, regularization loss= 0.042046001803607146
Validation accuracy: 0.90967, loss: 0.30237
Iteration 1100:        accuracy=0.93333, loss=0.17752, regularization loss= 0.04253029836741135
Test accuracy=0.91360, loss=0.29449
Validation accuracy: 0.91367, loss: 0.28771
Iteration 1200:        accuracy=0.96667, loss=0.19884, regularization loss= 0.042785308580394
Validation accuracy: 0.91342, loss: 0.30401
Iteration 1300:        accuracy=0.96667, loss=0.15347, regularization loss= 0.04306964078714564
Validation accuracy: 0.91267, loss: 0.29789
Iteration 1400:        accuracy=1.00000, loss=0.02740, regularization loss= 0.04315949498784981
Test accuracy=0.92230, loss=0.27149
Validation accuracy: 0.92083, loss: 0.26691
Iteration 1500:        accuracy=0.93333, loss=0.19517, regularization loss= 0.04342309713102317
```

```
In [12]: plt.figure(2, figsize=(18, 8))
         plt.subplot(2, 3, 1)
         plt.title('Training loss')
         plt.plot(train_results[:,0], train_results[:,1])
         plt.subplot(2, 3, 4)
         plt.title('Training accuracy')
         plt.plot(train_results[:,0], train_results[:,2])
         plt.subplot(2, 3, 2)
         plt.title('Validation loss')
         plt.plot(val_results[:,0], val_results[:,1])
         plt.subplot(2, 3, 5)
         plt.title('Validation accuracy')
         plt.plot(val_results[:,0], val_results[:,2])
         plt.subplot(2, 3, 3)
         plt.title('Testing loss')
         plt.plot(test_results[:,0], test_results[:, 1])
         plt.subplot(2, 3, 6)
         plt.title('Testing accuracy')
         plt.plot(test_results[:, 0], test_results[:,2])

Out[12]: [<matplotlib.lines.Line2D at 0x7f08077e2e48>]
```



9.1 Change of learning rate

If we change the initial learning rate from 0.001 to 0.1, the training process becomes unstable and the loss is out of control. Thus, you need to be careful when setting the initial learning rate.

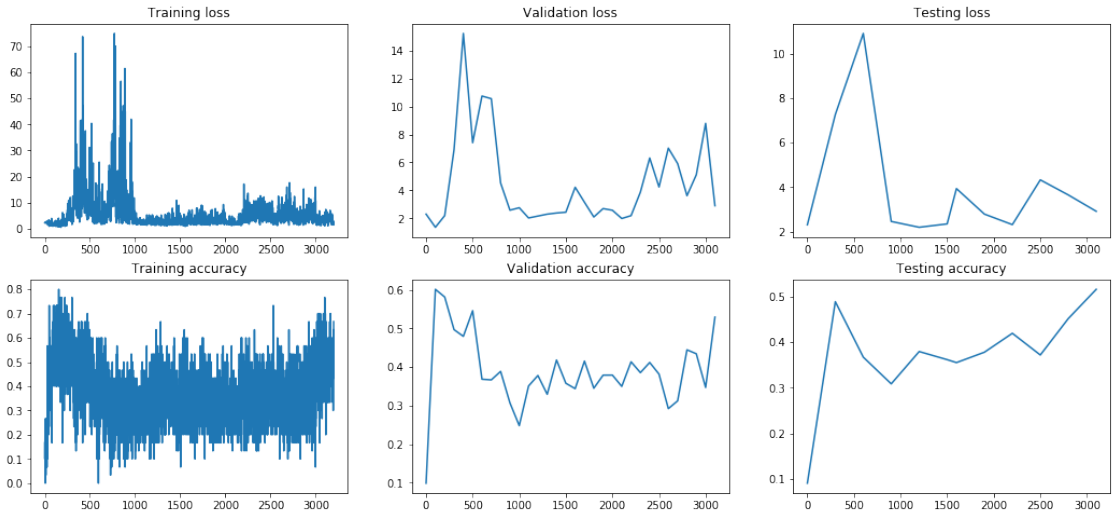
```
In [13]: model = MNISTNet()
         loss = SoftmaxCrossEntropy(num_class=10)

         # define your learning rate sheduler
         def func(lr, iteration):
             if iteration % 1000 ==0:
                 return lr*0.5
             else:
                 return lr

         adam = Adam(lr=0.01, decay=0,  sheduler_func = func)
         l2 = L2(w=0.001) # L2 regularization with lambda=0.001
         model.compile(optimizer=adam, loss=loss, regularization=l2)
         train_results, val_results, test_results = model.train(
             mnist,
             train_batch=30, val_batch=1000, test_batch=1000,
             epochs=2,
             val_intervals=100, test_intervals=300, print_intervals=100)
```

```
Epoch 0:
Test accuracy=0.09040, loss=2.30259
Validation accuracy: 0.09850, loss: 2.30259
Iteration 0:        accuracy=0.16667, loss=2.30259, regularization loss= 0.005212815460199879
Validation accuracy: 0.60192, loss: 1.36148
Iteration 100:        accuracy=0.60000, loss=1.32383, regularization loss= 0.3158125830973321
```

```
Validation accuracy: 0.58192, loss: 2.19570
Iteration 200:        accuracy=0.70000, loss=1.12469, regularization loss= 0.48396360224237345
Test accuracy=0.48930, loss=7.26545
Validation accuracy: 0.49725, loss: 6.90743
Iteration 300:        accuracy=0.46667, loss=7.91728, regularization loss= 0.9180977023590724
Validation accuracy: 0.47983, loss: 15.22511
Iteration 400:        accuracy=0.36667, loss=16.29151, regularization loss= 2.129209698521958
Validation accuracy: 0.54642, loss: 7.41058
Iteration 500:        accuracy=0.40000, loss=10.95386, regularization loss= 3.1761807625406595
Test accuracy=0.36740, loss=10.91698
Validation accuracy: 0.36850, loss: 10.74736
Iteration 600:        accuracy=0.30000, loss=9.56630, regularization loss= 4.109611084464017
Validation accuracy: 0.36675, loss: 10.55818
Iteration 700:        accuracy=0.50000, loss=7.08085, regularization loss= 4.186017132482088
Validation accuracy: 0.38883, loss: 4.52645
Iteration 800:        accuracy=0.56667, loss=3.89148, regularization loss= 5.774158891895546
Test accuracy=0.30870, loss=2.45384
Validation accuracy: 0.30700, loss: 2.58460
Iteration 900:        accuracy=0.23333, loss=2.41846, regularization loss= 7.2542336095567705
Validation accuracy: 0.24800, loss: 2.76055
Iteration 1000:        accuracy=0.26667, loss=2.31911, regularization loss= 7.500158186975829
Validation accuracy: 0.35092, loss: 2.02720
Iteration 1100:        accuracy=0.36667, loss=1.74690, regularization loss= 5.418911497329971
Test accuracy=0.37970, loss=2.19015
Validation accuracy: 0.37817, loss: 2.16324
Iteration 1200:        accuracy=0.40000, loss=1.88189, regularization loss= 4.751865255555399
Validation accuracy: 0.32933, loss: 2.29982
Iteration 1300:        accuracy=0.16667, loss=2.94359, regularization loss= 4.6936209938196045
Validation accuracy: 0.41833, loss: 2.38247
Iteration 1400:        accuracy=0.46667, loss=2.11959, regularization loss= 4.641843185246907
Test accuracy=0.36190, loss=2.34294
Validation accuracy: 0.35800, loss: 2.43842
Iteration 1500:        accuracy=0.23333, loss=2.54566, regularization loss= 4.514996825108424
Epoch 1:
Test accuracy=0.35540, loss=3.93225
Validation accuracy: 0.34400, loss: 4.22218
Iteration 0:        accuracy=0.26667, loss=4.55969, regularization loss= 4.6941092122399315
Validation accuracy: 0.41558, loss: 3.15016
Iteration 100:        accuracy=0.30000, loss=3.25757, regularization loss= 4.96978475146627
Validation accuracy: 0.34508, loss: 2.09704
Iteration 200:        accuracy=0.50000, loss=1.74439, regularization loss= 5.342696263901063
Test accuracy=0.37820, loss=2.78332
Validation accuracy: 0.37908, loss: 2.69979
Iteration 300:        accuracy=0.30000, loss=2.04807, regularization loss= 5.516101926983693
Validation accuracy: 0.37917, loss: 2.57896
Iteration 400:        accuracy=0.33333, loss=1.85732, regularization loss= 6.087318804967881
Validation accuracy: 0.34983, loss: 1.98947
Iteration 500:        accuracy=0.40000, loss=1.50551, regularization loss= 3.052576220928409
```

```
Test accuracy=0.41980, loss=2.31006
Validation accuracy: 0.41375, loss: 2.19342
Iteration 600:        accuracy=0.36667, loss=2.24940, regularization loss= 1.8357222121206758
Validation accuracy: 0.38558, loss: 3.87686
Iteration 700:        accuracy=0.33333, loss=4.36138, regularization loss= 1.4356603957792482
Validation accuracy: 0.41200, loss: 6.31662
Iteration 800:        accuracy=0.20000, loss=8.74972, regularization loss= 1.3643956817511314
Test accuracy=0.37210, loss=4.32443
Validation accuracy: 0.38167, loss: 4.24413
Iteration 900:        accuracy=0.43333, loss=3.24708, regularization loss= 1.3705888130684114
Validation accuracy: 0.29208, loss: 7.02236
Iteration 1000:        accuracy=0.36667, loss=6.02999, regularization loss= 1.386764535933595
Validation accuracy: 0.31242, loss: 5.91398
Iteration 1100:        accuracy=0.36667, loss=4.07254, regularization loss= 1.4312016528710743
Test accuracy=0.45210, loss=3.64708
Validation accuracy: 0.44483, loss: 3.62134
Iteration 1200:        accuracy=0.56667, loss=2.94765, regularization loss= 1.5000725696762864
Validation accuracy: 0.43458, loss: 5.11615
Iteration 1300:        accuracy=0.36667, loss=6.40507, regularization loss= 1.540135570782486
Validation accuracy: 0.34708, loss: 8.79349
Iteration 1400:        accuracy=0.26667, loss=10.81153, regularization loss= 1.6425053447776141
Test accuracy=0.51640, loss=2.91002
Validation accuracy: 0.52975, loss: 2.90536
Iteration 1500:        accuracy=0.76667, loss=1.13222, regularization loss= 1.5586740572554882
```

```
In [14]: plt.figure(2, figsize=(18, 8))
         plt.subplot(2, 3, 1)
         plt.title('Training loss')
         plt.plot(train_results[:,0], train_results[:,1])
         plt.subplot(2, 3, 4)
         plt.title('Training accuracy')
         plt.plot(train_results[:,0], train_results[:,2])
         plt.subplot(2, 3, 2)
         plt.title('Validation loss')
         plt.plot(val_results[:,0], val_results[:,1])
         plt.subplot(2, 3, 5)
         plt.title('Validation accuracy')
         plt.plot(val_results[:,0], val_results[:,2])
         plt.subplot(2, 3, 3)
         plt.title('Testing loss')
         plt.plot(test_results[:,0], test_results[:, 1])
         plt.subplot(2, 3, 6)
         plt.title('Testing accuracy')
         plt.plot(test_results[:, 0], test_results[:,2])

Out[14]: [<matplotlib.lines.Line2D at 0x7f0805dc3940>]
```



10 Train your best MNISTNet!

Tweak the hyperparameters of the above MNISTNet and use what you've learnt to train the best net.

Credits will be given based on your test accuracy, your explanations/insights of the training. The network is small, hence the training should finish quickly using your CPU (less than 1 hour).

Please report the following: - Training validation and testing loss as well as accuracy over iterations - Architecture and training method (eg. optimization scheme, data augmentation): explain your design choices, what has failed and what has worked and why you think they worked/failed - Try different hyper-parameters, e.g. rate decaying, weight decay, number of layers and total number of epochs.

Do NOT use external libraries like Tensorflow, keras and Pytorch in your implementation, i.e. optimizer.py, layer.py and loss.py. Do NOT copy the code from the internet, e.g. github. You should also give credits to any material that you refer to for your implementation.

11 Final submission instructions

Please submit the following:

1) Your code files in a folder `codes`

2) A short report (1-2 pages) in pdf titled `report.pdf`, explaining the logic (expressed using mathematical formulation) of your implementation (including the forward and backward function like ReLU) and the findings from training your best net

ASSIGNMENT DEADLINE: 4 MAR 2018 (SUN) 17:00PM

Do not include the `data` folder as it takes up substantial memory. Please zip up the following folders under a folder named with your NUSNET ID: eg. 'e0123456g.zip' and submit the zipped folder to IVLE/workbin/assignment 1 submission.