Yin Wang
dianawww

Assignment 1

1 Neural Network layer implementation

1.1 Fully-connected layer

Given $Y = W^T X + b$, i.e. $Y_j = \sum_i W^T_{ji} X_i + b_j$. where $X \in \mathbb{R}^{D_{in}}$, $Y \in \mathbb{R}^{D_{out}}$

$$\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial Y_k} \frac{\partial Y_k}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial Y_k} X_i (1_{k=j}) = \frac{\partial L}{\partial Y_j} X_i \Rightarrow \boxed{\frac{\partial L}{\partial W} = X \left(\frac{\partial L}{\partial Y}\right)^T}$$

$$\frac{\partial L}{\partial b_j} = \sum_k \frac{\partial L}{\partial Y_k} \frac{\partial Y_k}{\partial b_j} = \sum_k \frac{\partial L}{\partial Y_k} (1_{k=j}) = \frac{\partial L}{\partial Y_j} \Rightarrow \boxed{\frac{\partial L}{\partial b} = \frac{\partial L}{\partial Y}}$$

$$\frac{\partial L}{\partial X_i} = \sum_k \frac{\partial L}{\partial Y_k} \frac{\partial Y_k}{\partial X_i} = \sum_k \frac{\partial L}{\partial Y_k} W_{ik} = \left(W \frac{\partial L}{\partial Y}\right)_i \Rightarrow \boxed{\frac{\partial L}{\partial X} = W \frac{\partial L}{\partial Y}}$$

1.2 ReLU

Given $Y = \mathrm{ReLU}(X) = \max\{0, X\}$ i.e. $Y_{ij} = \begin{cases} X_{ij}, & X_{ij} > 0 \\ 0, & \text{o.w.} \end{cases}$

$$\Rightarrow \left(\frac{\partial Y}{\partial X}\right)_{ij} = \begin{cases} 1 & X_{ij} > 0 \\ 0 & \text{o.w.} \end{cases} \Rightarrow \boxed{\left(\frac{\partial L}{\partial X}\right)_{ij} = \begin{cases} \left(\frac{\partial L}{\partial Y}\right)_{ij} & X_{ij} > 0 \\ 0 & \text{o.w.} \end{cases}}$$

1.3 Dropout

Given $Y = X \odot M$, i.e. $Y_{ij} = X_{ij} M_{ij} = \begin{cases} X_{ij}, & M_{ij} = 1 \\ 0, & M_{ij} = 0 \end{cases}$

$$\frac{\partial L}{\partial X_{ij}} = \sum_{kl} \frac{\partial L}{\partial Y_{kl}} \frac{\partial Y_{kl}}{\partial X_{ij}} = \sum_{kl} \frac{\partial L}{\partial Y_{kl}} \cdot M_{ij} (1_{k=i})(1_{l=j}) = \frac{\partial L}{\partial Y_{ij}} M_{ij} = \begin{cases} \frac{\partial L}{\partial Y_{ij}}, & M_{ij} = 1 \\ 0 & M_{ij} = 0 \end{cases}$$

$$\Rightarrow \boxed{\left(\frac{\partial L}{\partial X}\right)_{ij} = \begin{cases} \left(\frac{\partial L}{\partial Y}\right)_{ij}, & M_{ij} = 1 \\ 0 & M_{ij} = 0 \end{cases}}$$

1.4 Batch Normalization

.

Batch Normalization.

$X_i \in \mathbb{R}^N \quad Y_i \in \mathbb{R}^N$, Given $Y_i = \gamma\left(\frac{X_i - \mu}{\sigma}\right) + \beta$, $\mu = \frac{1}{n}\sum_j X_j$, $\sigma = \sqrt{\frac{1}{n}\sum_j (X_j - \mu)^2 + \varepsilon}$

$$\Rightarrow \boxed{\frac{\partial \mu}{\partial X_i} = \frac{1}{n}}$$

$$\frac{\partial \sigma}{\partial X_i} = \frac{\partial \sigma}{\partial X_i} + \frac{\partial \sigma}{\partial \mu}\frac{\partial \mu}{\partial X_i} = \frac{1}{2}\cdot\frac{1}{\sigma}\cdot\frac{1}{n}\cdot 2(X_i - \mu) + \frac{1}{2}\cdot\frac{1}{\sigma}\cdot\frac{1}{n}\underbrace{\sum_j\left[(X_j - \mu)\cdot 2\right]}_{0}\cdot(-1)\cdot\frac{1}{n}$$

$$\Rightarrow \boxed{\frac{\partial \sigma}{\partial X_i} = \frac{1}{\sigma n}(X_i - \mu)}$$

$$\frac{\partial L}{\partial X_{i,j}} = \underbrace{\sum_k \frac{\partial L}{\partial Y_{i,k}}\frac{\partial Y_{i,k}}{\partial X_{i,j}}}_{A} + \underbrace{\sum_k \frac{\partial L}{\partial Y_{i,k}}\frac{\partial Y_{i,k}}{\partial \mu}}_{B}\frac{\partial \mu}{\partial X_{i,j}} + \underbrace{\sum_k \frac{\partial L}{\partial Y_{i,k}}\frac{\partial Y_{i,k}}{\partial \sigma}\frac{\partial \sigma}{\partial X_{i,j}}}_{C}$$

, where $Y_{i,k}$ is the kth element in Y_i

where $A = \sum_k \frac{\partial L}{\partial Y_{i,k}}\frac{\gamma}{\sigma}(1_{k=j}) = \frac{\partial L}{\partial Y_{i,j}}\frac{\gamma}{\sigma}$.

$B = \sum_k \frac{\partial L}{\partial Y_{i,k}}\left(-\frac{\gamma}{\sigma}\right)\left(\frac{1}{n}\right) = -\frac{\gamma}{\sigma n}\sum_k\left(\frac{\partial L}{\partial Y_i}\right)_k$

$C = \sum_k \frac{\partial L}{\partial Y_{i,k}}\left(\frac{\gamma(X_{i,k} - \mu)}{-\sigma^2}\right)\cdot\left(\frac{X_{i,j} - \mu}{\sigma n}\right) = -\frac{\gamma(X_{i,j} - \mu)}{\sigma^3 n}\left[\sum_k\left(\frac{\partial L}{\partial Y_i}\right)_k((X_i)_k - \mu)\right]$

$$\Rightarrow \boxed{\frac{\partial L}{\partial X_i} = \frac{\gamma}{\sigma}\frac{\partial L}{\partial Y_i} - \frac{\gamma}{\sigma n}\sum_k\left(\frac{\partial L}{\partial Y_i}\right)_k - \frac{\gamma}{\sigma^3 n}(X_i - \mu)\left[\sum_k\left(\frac{\partial L}{\partial Y_i}\right)_k((X_i)_k - \mu)\right]}$$

where $X_i, \frac{\partial L}{\partial Y_i}$ are ith column in X, $\frac{\partial L}{\partial Y}$; $(X_i)_k, \left(\frac{\partial L}{\partial Y_i}\right)_k$ are kth element in X_i, $\frac{\partial L}{\partial Y_i}$

1.5 Convolution

. i) Verify the output size:

Given Image size of $H \times W$, and a filter size of $H' \times W'$,

For height, we have $\left(\frac{H'-1}{2}\right)$ pixels on the left border smaller than original image, and $\left(\frac{H'-1}{2}\right)$ pixels on the right side. Hence the output height is

$$H - \frac{H'-1}{2} - \frac{H'-1}{2} = H - H' + 1$$

Similarly, the output width is $W - \frac{W'-1}{2} - \frac{W'-1}{2} = W - W' + 1$.

Therefore, the output size is $\boxed{(H - H' + 1) \times (W - W' + 1)}$ QED.

ii). Note that there are totally 9 variables: $n, f, c, h, w, h', w', h'', w''$ (where $h=h'+h''-1$, $w=w'+w''-1$)

We have $Y_{n,f} = \sum_c X_{n,c} *_{valid} \overline{W}_{f,c} = \sum_c X_{n,c} *_{filt} W_{f,c}$

Given $X \in \mathbb{R}^{N\times C\times H\times W}$, $W \in \mathbb{R}^{F\times C\times H'\times W'}$, $Y \in \mathbb{R}^{N\times F\times H''\times W''}$.

For a full convolution, $Y_{n,f} = \sum_c X_{n,c} *_{full} W_{f,c} \Leftrightarrow Y_{n,f,h'',w''} = X_{n,c,h,w} W_{f,c,h',w'}$

$$\Rightarrow \frac{\partial L}{\partial X_{n,c,h,w}} = \sum_{f,h'',w''} \frac{\partial L}{\partial Y_{n,f,h'',w''}} \cdot \frac{\partial Y_{n,f,h'',w''}}{\partial X_{n,c,h,w}} = \sum_{f,h',w'} \frac{\partial L}{\partial Y_{n,f,h-h'+1,w-w'+1}} W_{f,c,h',w'}$$

$$= \sum_f \left(\sum_{h',w'} \frac{\partial L}{\partial Y_{n,f,h-h'+1,w-w'+1}} W_{f,c,h',w'} \right) = \sum_f \left(W_{f,c} *_{full} \left(\frac{\partial L}{\partial Y_{n,f}}\right) \right)_{h,w}$$

since (h,h',h''), (w,w',w'') are related

$$\Rightarrow \boxed{\frac{\partial L}{\partial X_{n,c}} = \sum_f W_{f,c} *_{full} \left(\frac{\partial L}{\partial Y_{n,f}}\right)}$$

Similarly for filter convolution,

$$\frac{\partial L}{\partial W_{f,c,h',w'}} = \sum_{n,h'',w''} \frac{\partial L}{\partial Y_{n,f,h'',w''}} \frac{\partial Y_{n,f,h'',w''}}{\partial W_{f,c,h',w'}} = \sum_{n,h'',w''} \frac{\partial L}{\partial Y_{n,f,h'',w''}} X_{n,c,h'+h''-1,w'+w''-1}$$

$$= \sum_n \left(\sum_{h'',w''} \frac{\partial L}{\partial Y_{n,f,h'',w''}} X_{n,c,h,w} \right) = \sum_n \left(X_{n,c} *_{filt} \left(\frac{\partial L}{\partial Y_{n,f}}\right) \right)_{h',w'}$$

$$\Rightarrow \boxed{\frac{\partial L}{\partial W_{f,c}} = \sum_n X_{n,c} *_{filt} \left(\frac{\partial L}{\partial Y_{n,f}}\right)}$$

2 NN with logistic regression for binary calssification

2.1 Simple NN with no hidden layer

The test accuracy is 0.928, and the parameters of my best model are shown below.

List 1: Simple NN with logistic regression

```
np.random.seed(598)
model_logit_simple = LogisticClassifier(
        input_dim=Din,
        hidden_dim=None,
        weight_scale=0.5,
        reg=0.0,
        fwd_fun=relu_forward,
        bwd_fun=relu_backward
)
net_logit_simple = Solver(
        model=model_logit_simple, data=data,
        update_rule='sgd_momentum',
        optim_config={
                'learning_rate': 0.1
        },
        lr_decay=1.0,
        batch_size=20,
```

```
            num_epochs=400,
            verbose=False,
            print_every=10
    )
```

2.2 NN with one hidden layer

The test accuracy is 0.932 with 40 hidden units, and the parameters are shown below.

List 2: 2-layer NN with logistic loss

```
    np.random.seed(598)
    print('2 layer NN with logistic regression:')
    model_logit = LogisticClassifier(
            input_dim=Din,
            hidden_dim=40,
            reg=0.0,
            fwd_fun=relu_forward,
            bwd_fun=relu_backward
    )
    net_logit = Solver(
            model=model_logit, data=data,
            update_rule='sgd_momentum',
            optim_config={
                            'learning_rate': 0.01,
                            },
            lr_decay=1.0,
            batch_size=30,
            num_epochs=200,
            verbose=False,
            print_every=10
    )
```

3 NN with hinge-loss output layer for binary classification

3.1 Simple NN with no hidden layer

The test accuracy is 0.924, and the parameters are shown below.

List 3: Simple NN with hinge loss

```
    np.random.seed(598)
    model_svm_simple = SVM(
            input_dim=Din,
            hidden_dim=None,
```

```
            weight_scale=0.1,
            reg=0.0,
            fwd_fun=relu_forward,
            bwd_fun=relu_backward
)
net_svm_simple = Solver(
            model=model_svm_simple, data=data,
            update_rule='sgd_momentum',
            optim_config={
                        'learning_rate': 0.01
            },
            lr_decay=1.0,
            batch_size=10,
            num_epochs=400,
            verbose=False,
            print_every=10
)
```

3.2 NN with one hidden layer

The test accuracy is 0.928 with 20 hidden units, and the parameters are shown below.

List 4: 2-layer NN with hinge loss

```
np.random.seed(598)
print('2 layer NN with hinge loss:')
model_svm = SVM(
            input_dim=Din,
            hidden_dim=20,
            weight_scale=0.1,
            reg=0.0,
            fwd_fun=relu_forward,
            bwd_fun=relu_backward
)
net_svm = Solver(
            model=model_svm, data=data,
            update_rule='sgd_momentum',
            optim_config={
                                    'learning_rate': 0.01,
                                    },
            lr_decay=1.0,
            batch_size=10,
            num_epochs=400,
            verbose=False,
            print_every=10
)
```

4 NN with softmax output layer for multi-class classification

4.1 Simple NN with no hidden layer

The test accuracy is 0.922, and parameters are shown below.

List 5: Simple NN with softmax loss

```
print('Simple NN with softmax loss:')
np.random.seed(598)
model_softmax_simple = SoftmaxClassifier(
        input_dim=data['X_train'].shape[1:],
        hidden_dim=None,
        num_classes=10,
        reg=0.0,
        fwd_fun=relu_forward,
        bwd_fun=relu_backward
)
net_softmax_simple = Solver(
        model=model_softmax_simple, data=data,
        update_rule='adam',
        optim_config={
                'learning_rate': 0.001
        },
        lr_decay=1.0,
        batch_size=100,
        num_epochs=50,
        verbose=True,
        print_every=100
)
```

4.2 NN with one hidden layer

The test accuracy is 0.987 with 600 hidden units, and the parameters are shown below.

List 6: 2-layer NN with softmax loss

```
np.random.seed(598)
model_softmax_simple = SoftmaxClassifier(
        input_dim=data['X_train'].shape[1:],
        hidden_dim=600,
        num_classes=10,
        reg=0.0,
        fwd_fun=relu_forward,
        bwd_fun=relu_backward
)
net_softmax_simple = Solver(
```

```
                model=model_softmax_simple, data=data,
                update_rule='adam',
                optim_config={
                        'learning_rate': 0.001
                },
                lr_decay=1.0,
                batch_size=100,
                num_epochs=20,
                verbose=True,
                print_every=100
        )
```

5 Convolutional Neural Network

5.1 CNN without dropout and normalization

The test accuracy is 0.991 with a filter size of (5, 5) and 512 hidden units in fully-connected layer. The parameters are shown below.

List 7: 2-layer CNN with softmax loss

```
    np.random.seed(598)
        model_cnn = ConvNet(
                num_filters=32,
                filter_size=5,
                hidden_dim=512,
                num_classes=10,
                reg=0.0,
                dropout = 0,
                normalization = False
        )
        net_cnn = Solver(
                model=model_cnn, data=data,
                update_rule='adam',
                optim_config={
                        'learning_rate': 0.01
                },
                lr_decay=1.0,
                batch_size=50,
                num_epochs=15,
                verbose=True,
                print_every=10
        )
```

5.2 CNN with dropout and normalization

The test accuracy is 0.992 with a filter size of (5, 5) and 512 hidden units in fully-connected layer as before. The dropout rate is 0.3 and the architecture is like

conv - norm - ReLU - dropout - FC - norm - ReLU - dropout - FC - softmax

Some parameters are shown below.

List 8: 2-layer CNN with softmax loss

```
np.random.seed(598)
model_cnn = ConvNet(
        num_filters=32,
        filter_size=5,
        hidden_dim=512,
        num_classes=10,
        reg=0.0,
        dropout = 0.3,
        normalization = True
)
net_cnn = Solver(
        model=model_cnn, data=data,
        update_rule='adam',
        optim_config={
                'learning_rate': 0.01
        },
        lr_decay=1.0,
        batch_size=50,
        num_epochs=10,
        verbose=True,
        print_every=10
)
```

6 CNN image classification using VGG11

6.1 Default Architecture

The test accuracy is 0.71 with batchnorm and ReLu activation. The architecture and parameters are shown below.

List 9: Default Architecture of CNN using VGG11

```
def __init__(self):
super(VGG, self).__init__()
self.conv = nn.Sequential(
    # Stage 1
    # TODO: convolutional layer, input channels 3,
```

```
#       output channels 8, filter size 3
# TODO: max-pooling layer, size 2
nn.Conv2d(3, 8, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(8),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Stage 2
# TODO: convolutional layer, input channels 8,
#       output channels 16, filter size 3
# TODO: max-pooling layer, size 2
nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(16),
nn.ReLU(),
nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(16),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Stage 3
# TODO: convolutional layer, input channels 16,
#       output channels 32, filter size 3
# TODO: convolutional layer, input channels 32,
#       output channels 32, filter size 3
# TODO: max-pooling layer, size 2
nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(32),
nn.ReLU(),
nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(32),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Stage 4
# TODO: convolutional layer, input channels 32,
#       output channels 64, filter size 3
# TODO: convolutional layer, input channels 64,
#       output channels 64, filter size 3
# TODO: max-pooling layer, size 2
nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(64),
nn.ReLU(),
nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(64),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Stage 5
# TODO: convolutional layer, input channels 64,
#       output channels 64, filter size 3
```

```
        # TODO: convolutional layer, input channels 64,
        #       output channels 64, filter size 3
        # TODO: max-pooling layer, size 2
        nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2)

    )
    self.fc = nn.Sequential(
        # TODO: fully-connected layer (64->64)
        # TODO: fully-connected layer (64->10)
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, 10)
    )
```

6.2 My Architecture of CNN using VGG11

The test accuracy is 0.81. The new architecture and parameters are shown below.

List 10: My architecture of CNN using VGG11

```
    np.random.seed(598)
    def __init__(self):
    super(VGG, self).__init__()
    self.conv = nn.Sequential(
        nn.Conv2d(3, 48, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(48),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Dropout(0.25),

        nn.Conv2d(48, 96, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(96),
        nn.ReLU(),
        nn.Conv2d(96, 96, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(96),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Dropout(0.25),

        nn.Conv2d(96, 192, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(192),
        nn.ReLU(),
        nn.Conv2d(192, 192, kernel_size=3, stride=1, padding=1),
```

```
            nn.BatchNorm2d(192),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.25),

        )
        self.fc = nn.Sequential(
            nn.Linear(3072, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )
```

7 Short answer questions

1. The main difficulty when training deep NN with sigmoid non-linearity is that it vanishes gradient during backpropgation. Since sigmoid function maps a number in $(-\infty, \infty)$ to (0, 1), thus updating input value will only lead to a small update in output. Then if we have a large number of layers, the last layer might be easy to converge with large gradient, but the first layer is hardly different from initial random values.

2. After this dropout layer, the output y has 100(1-p)% zero terms, while other non-zero terms are the same as those in x. In other words, the output behaves just like the input but with a reduced dimension ((1-p)*x.shape). This layer could be used after an activation function, before a new convolutional or fully-connected layer. Applying the dropout method, we can reduce the train/test mismatch which might lead to overfitting, and also increase the speed of trainning.

3. I would firstly change the learning rate (check its initial value, or its decay rule) since this problem could result from a large learning rate. If it is not working, I would check the update rule, maybe there is some bugs in the solver.