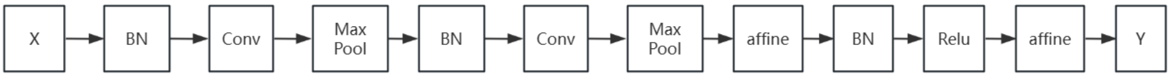
模型架构

我实现了一个4层的卷积神经网络，先对输入的数据进行batch normalization，将处理后的数据传到第一个卷积层，然后进行最大池化；将池化输出进行一次batch normalization，再传给第二个卷积层后，再进行最大池化；将上一层的池化结果传给第一个affine层后进行batch normalization，再将batch normalization处理后的结果传给$relu$层进行激活；最后将$relu$层的激活结果传给第二个affine层，该层输出即为该神经网络最后的输出。

图示如下：



在`cnn.py`中实现的模型的代码如下：

```
class MyConvNet(object):
    def __init__(self,
                input_dim=(3,32,32),
                num_filters_1=32,
                num_filters_2=64,
                hidden_dim = 100,
                filter_size=7,
                num_classes=10,
                weight_scale=1e-3,
                reg=0.0,
                dtype=np.float32,
                ):
        self.reg = reg
        self.dtype = dtype
        self.params = {}
        C, H, W = input_dim
        self.params["W1"] = np.random.normal(loc=0.0,scale=weight_scale,size=
(num_filters_1,C,filter_size,filter_size))
        self.params["b1"] = np.zeros(num_filters_1)
        self.params["gamma1"] = np.ones(C)
        self.params["beta1"] = np.zeros(C)
        self.params["W2"] = np.random.normal(loc=0.0,scale=weight_scale,size=
(num_filters_2, num_filters_1, filter_size, filter_size))
        self.params["b2"] = np.zeros(num_filters_2)
        self.params["gamma2"] = np.ones(num_filters_1)
        self.params["beta2"] = np.zeros(num_filters_1)
        self.params["W3"] = np.random.normal(loc=0.0,scale=weight_scale,size=
(num_filters_2*H*W//16, hidden_dim))
        self.params["b3"] = np.zeros(hidden_dim)
        self.params["gamma3"] = np.ones(hidden_dim)
        self.params["beta3"] = np.zeros(hidden_dim)
        self.params["W4"] = np.random.normal(loc=0.0, scale=weight_scale, size=
(hidden_dim, num_classes))
        self.params["b4"] = np.zeros(num_classes)
        self.bn_params = [{"mode": "train"}, {"mode": "train"}, {"mode":
"train"}, {"mode": "train"}]
        for k, v in self.params.items():
```

```
            self.params[k] = v.astype(dtype)
    def loss(self, X, y=None):
        X = X.astype(self.dtype)
        mode = "test" if y is None else "train"
        for bn_param in self.bn_params:
            bn_param["mode"] = mode

        W1, b1 = self.params["W1"], self.params["b1"]
        W2, b2 = self.params["W2"], self.params["b2"]
        W3, b3 = self.params["W3"], self.params["b3"]
        W4, b4 = self.params["W4"], self.params["b4"]
        gamma1, beta1 = self.params["gamma1"], self.params["beta1"]
        gamma2, beta2 = self.params["gamma2"], self.params["beta2"]
        gamma3, beta3 = self.params["gamma3"], self.params["beta3"]

        filter_size = W1.shape[2]
        conv_param = {"stride": 1, "pad": (filter_size - 1) // 2}

        pool_param = {"pool_height": 2, "pool_width": 2, "stride": 2}
        scores = None
        bn_x_1,cache_bn_1 =
spatial_batchnorm_forward(X,gamma1,beta1,self.bn_params[0])
        out_1,cache_1 =
conv_relu_pool_forward(bn_x_1,W1,b1,conv_param,pool_param)
        bn_x_2, cache_bn_2 =
spatial_batchnorm_forward(out_1,gamma2,beta2,self.bn_params[1])
        out_2,cache_2 =
conv_relu_pool_forward(bn_x_2,W2,b2,conv_param,pool_param)
        out_3, cache_3 = affine_bn_relu_forward(out_2, W3, b3, gamma3, beta3,
self.bn_params[3])
        scores,cache_4 = affine_forward(out_3,W4,b4)

        if y is None:
            return scores

        loss, grads = 0, {}

        loss, da = softmax_loss(scores,y)
        loss += 0.5*self.reg*
(np.sum(W1*W1)+np.sum(W2*W2)+np.sum(W3*W3)+np.sum(W4*W4))
        d_affine_out, d_affine_w, d_affine_b = affine_backward(da,cache_4)
        d_relu_out, d_relu_w, d_relu_b, dgamma3, dbeta3 =
affine_bn_relu_backward(d_affine_out,cache_3)
        reshaped_d_relu_out = d_relu_out.reshape(out_2.shape)
        d_conv_out_2, d_conv_w_2, d_conv_b_2 =
conv_relu_pool_backward(reshaped_d_relu_out,cache_2)
        d_spatial_out_2, dgamma2, dbeta2 =
spatial_batchnorm_backward(d_conv_out_2,cache_bn_2)
        reshaped_d_spatial_out_2 = d_spatial_out_2.reshape(out_1.shape)
        d_conv_out_1, d_conv_w_1, d_conv_b_1 =
conv_relu_pool_backward(reshaped_d_spatial_out_2,cache_1)
        d_spatial_out_1, dgamma1, dbeta1 =
spatial_batchnorm_backward(d_conv_out_1,cache_bn_1)

        grads["W1"] = d_conv_w_1 + self.reg*W1
```

```
        grads["W2"] = d_conv_w_2 + self.reg*W2
        grads["W3"] = d_relu_w + self.reg*W3
        grads["W4"] = d_affine_w + self.reg*W4
        grads["b1"], grads["b2"], grads["b3"], grads["b4"] = d_conv_b_1,
d_conv_b_2, d_relu_b, d_affine_b
        grads["gamma1"], grads["gamma2"], grads["gamma3"] = dgamma1, dgamma2,
dgamma3
        grads["beta1"], grads["beta2"], grads["beta3"] = dbeta1, dbeta2, dbeta3
        return loss, grads
```

在`ConvolutionalNetworks.ipynb`中编写的调用和测试在`cnn.py`中实现的架构的代码如下：

```
## design and train your model
model = MyConvNet()

N = 100
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size = N)

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss)

model = MyConvNet(weight_scale=0.001, hidden_dim=100, reg=0.001)

solver = Solver(model, data,
                num_epochs=5, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()

print(
    "Full data training accuracy:",
    solver.check_accuracy(small_data['X_train'], small_data['y_train'])
)
print(
    "Full data validation accuracy:",
    solver.check_accuracy(data['X_val'], data['y_val'])
)
from annp.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```

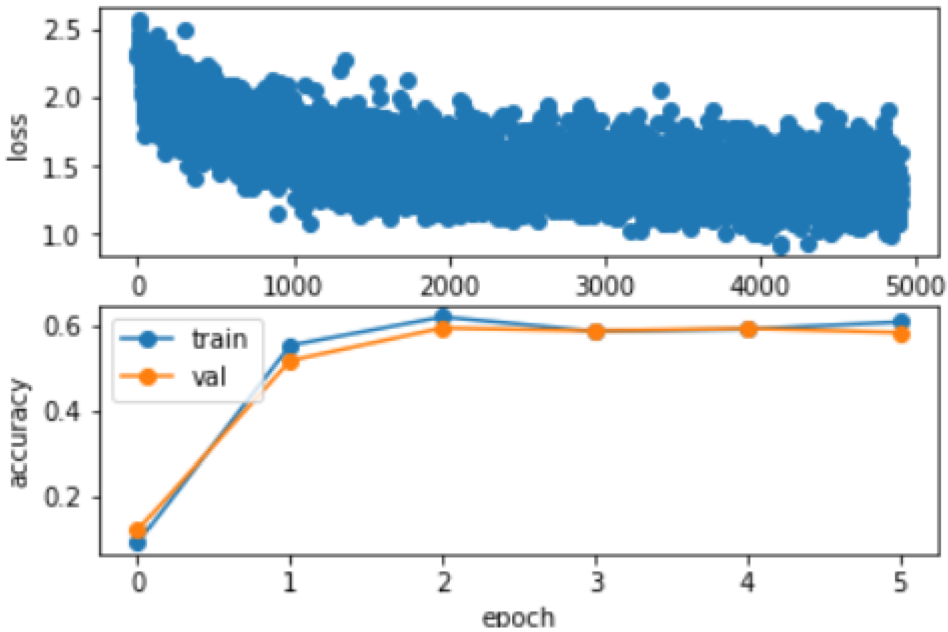
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

调参过程

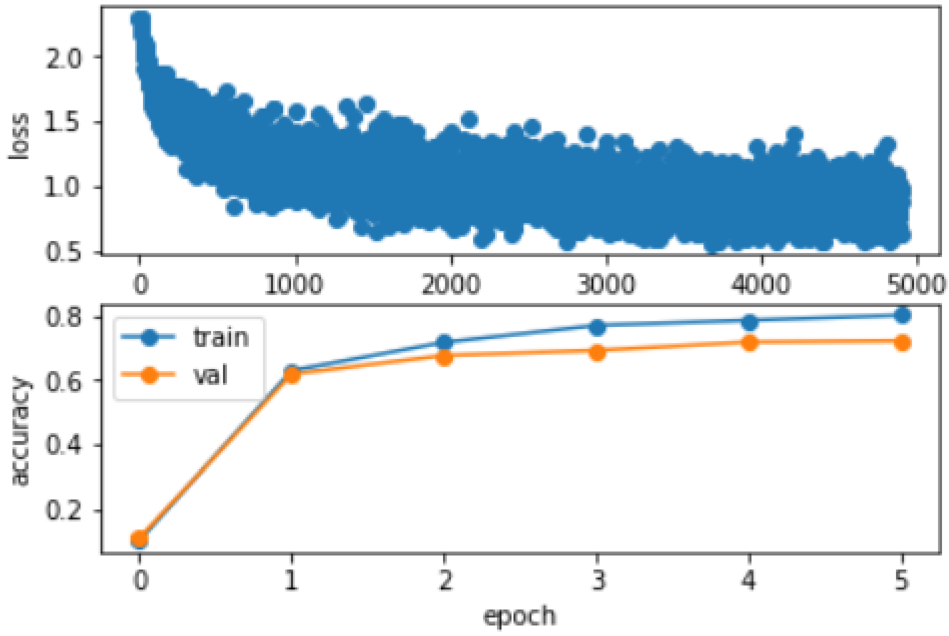
以下调参过程epoch均设置为5。

learning rate

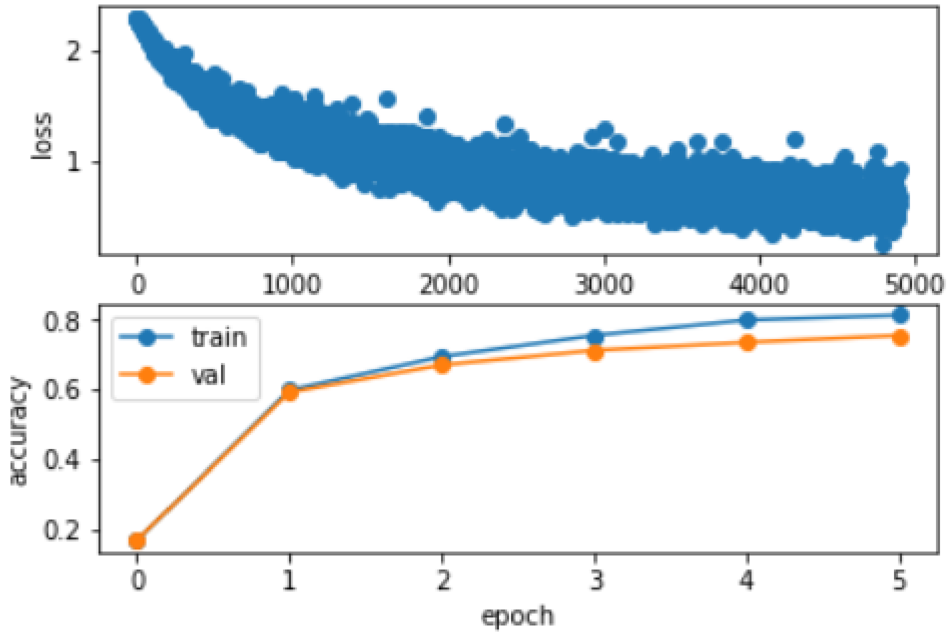
- $\mathbf{batch\ size = 50; learning\ rate = 1e-2; reg = 0.5}$



- $\mathbf{batch\ size = 50; learning\ rate = 1e-3; reg = 0.5}$

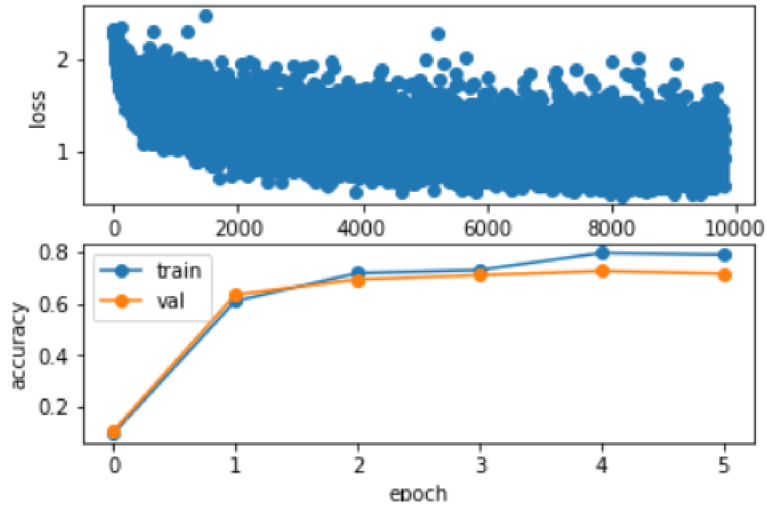


- $\mathbf{batch\ size = 50; learning\ rate = 1e-4; reg = 0.5}$

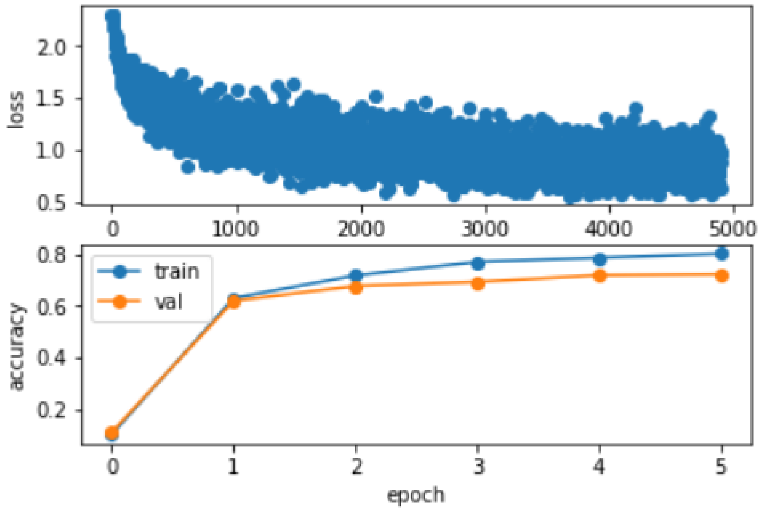


batch size

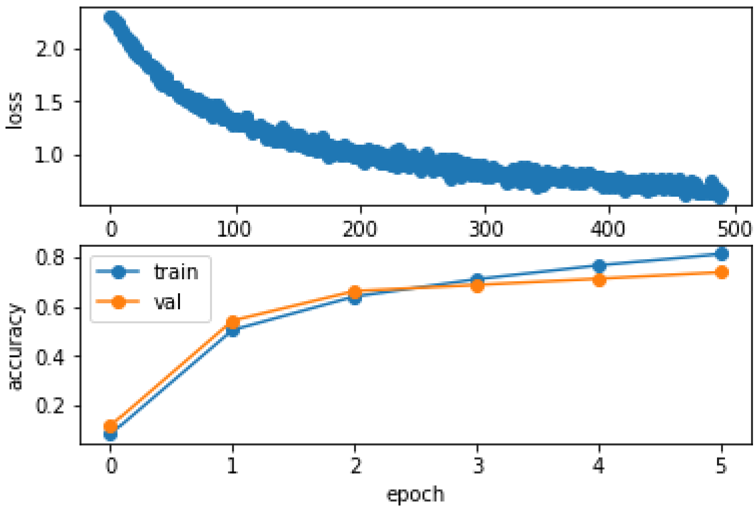
- $\mathbf{batch\ size = 25; learning\ rate = 1e-3; reg = 0.001}$



- **batch size = 50; learning rate = 1e − 3; reg = 0.001**

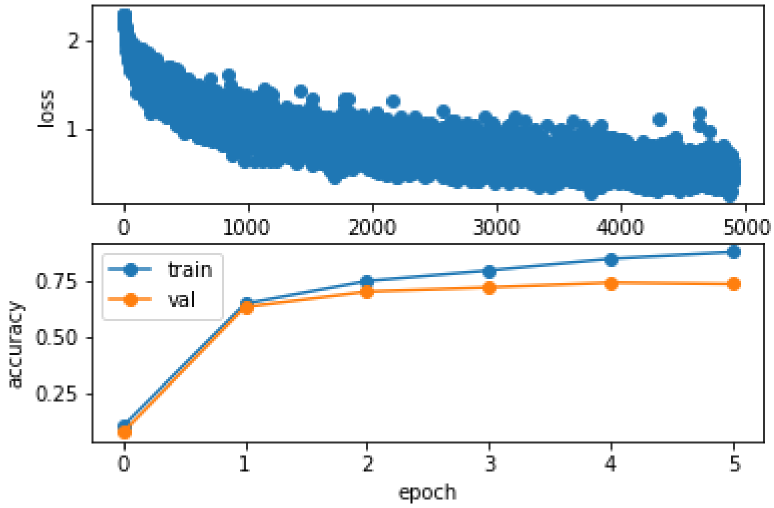


- **batch size = 400; learning rate = 1e − 3; reg = 0.001**

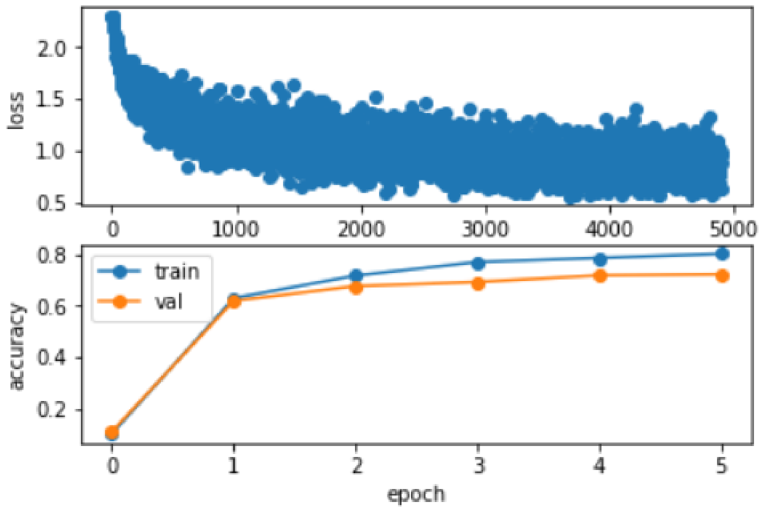


reg

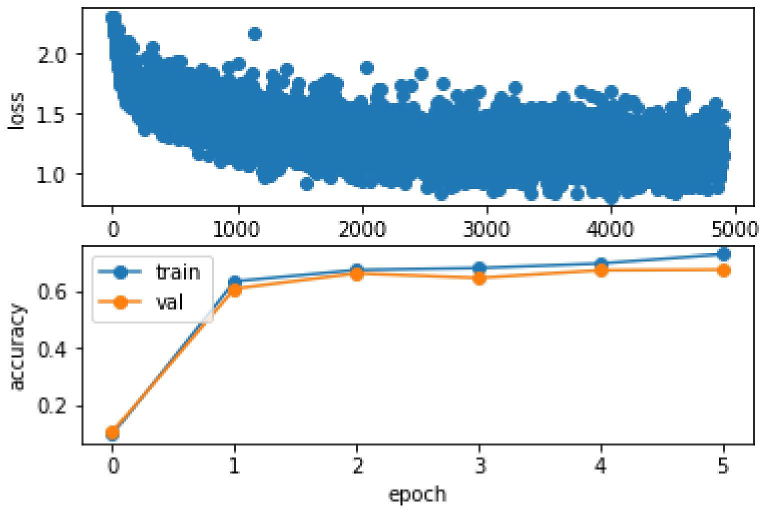
- **batch size = 50; learning rate = 1e − 3; reg = 0.0001**



- $\mathbf{batch\ size = 50; learning\ rate = 1e-3; reg = 0.001}$



- $\mathbf{batch\ size = 50; learning\ rate = 1e-3; reg = 0.01}$



实验结果分析

learning rate

从上述结果来看，当学习率较小时(1e-3 VS 1e-4)，损失降低的速度较慢(从损失分布图来看，学习率为1e-3时下降得比学习率为1e-4要快)，训练和验证的准确率提高的速度较慢(从训练和验证准确率图表可以看到学习率为1e-3时准确率提升得比学习率为1e-4时要快)；而当学习率较大时(1e-3 VS 1e-2)，损失降低的速度也会变慢(从损失分布图来看，学习率为1e-3时下降得比学习率为1e-2要快)，训练和验证的准确率提高的速度较慢(从训练和验证准确率图表可以看到学习率为1e-3时准确率提升得比学习率为1e-2时要快)，且学习率较大时，总体准确率不高、甚至还出现了振荡。

当学习率过低时，收敛速度过慢，导致模型需要更多的训练轮次和时间才能达到理想的分类效果；而当学习率过高时，不够稳定，会在最优情况附近“徘徊”，导致很难收敛到最优模型、模型的分类效果极不稳定等。因此，在神经网络中，要慎重设置学习率，选择适中的学习率来确保收敛速度和收敛效果。

batch size

可以看到在一个相对合理的范围内增大batch size时(本实验为25-400)，收敛的速度会提高且收敛更稳定(从损失分布图可以看出)，训练和验证的准确率上升也更加稳定，但是batch size增大时会导致训练时间增加；而降低batch size时，收敛速度降低(从损失分布图可以看出)，训练和验证的准确率上升相对不稳定，但是batch size较小时，训练时间相对地会更短。

reg

可以看到当正则化项较大时(0.001 VS 0.01)，训练和验证的准确率不高(观察准确率图表可知)，且模型的收敛速度和收敛情况也远不如reg为0.001和0.0001的情况(观察3种情况的损失分布图的损失下降趋势)，这是因为较大的reg值会增加正则化项的权重，导致模型过于简单，无法捕捉数据的复杂性，导致准确率过低。

而当正则化项过小时(0.001 VS 0.0001)，虽然reg为0.0001时模型的收敛速度和收敛情况和reg为0.001的相近，训练准确率也很接近，但是验证的准确率不及reg为0.001的，这是因为正则化项的权重过低，导致模型更容易出现过拟合，进而导致验证准确率降低。