# ASSIGNMENT 2: CONVOLUTIONAL NETWORKS [IFT6135]

JOSEPH D. VIVIANO, JONATHAN GUYMONT, MARZIEH MEHDIZADEH

## 1. Regularization : weight decay, early stopping, dropout, domain prior knowledge

(a) *Early stopping and weight decay*

The Figure 1 shows that using both L2 regularization and ealy stopping is not optimal. L2 regularization and early stopping can be seen as equivalent under certain condition (Goodfellow et al. 2016). It gives a justification for not using a L2 regularization term when using early stopping (Collobert et al, 2004). This phenomenon can be observe in Figure 1 where the model without weigth decay perform better.

The Figure 2 show that using weigth decay push the weight toward 0. Let $\alpha$ be the learning rate, $\lambda$ be the regularization parameter and $m$ be the training size. Then we can show that if $|1 - \frac{\alpha\lambda}{m}| < 1$, then the norm of the weight parameters converge to zero as the number of epoch approach infinity. This is because each time you update the weight, your multiplying the previous weight by a number lower than 1 in absolute value, which cause the updated weigth to be lower then the previous one.

*Date*: Feburary 2018.

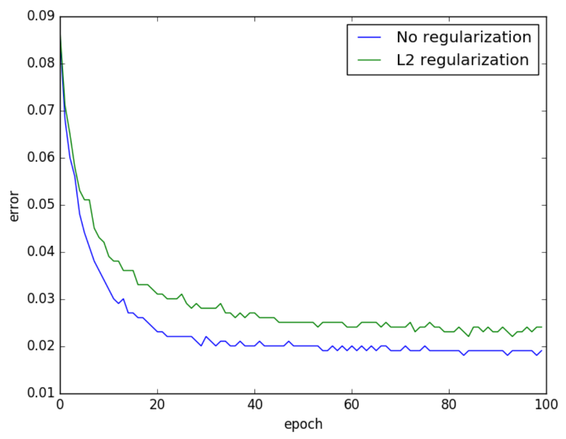

FIGURE 1. Test error at the end of each epoch of training for MLP models with and without regularization. The achitecture of the models is: 2 hidden layers with 800 units with ReLU activations. Optimization is done using SGD with learning rate of 0.02, minibatch size of 64, and the Glorot Normal initialization. Both models are train for 100 epochs.

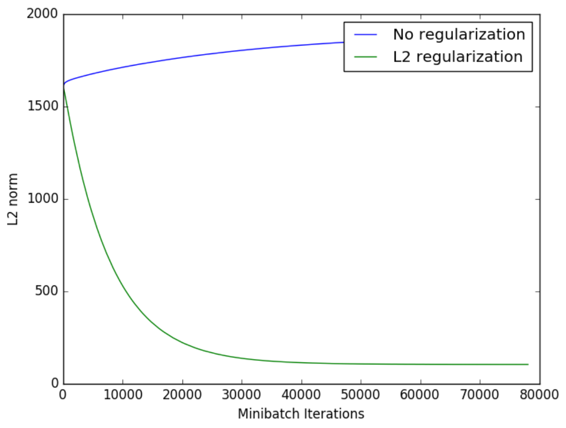

FIGURE 2. L2 norm of all the parameters at each iteration (minibatch update).

(b) *Dropout*

We showed in question 5 of the theory part of the assignment 2 that evaluating a model using dropout for all possible dropout configurations and with dropout probility of 0.5 and taking the geometric average is equivalent to evaluate the model without dropout and multiply the weight by 0.5. Since in schemes (*ii*) and (*iii*) we take the arithmetic average and not the geometric average, the approximation is not optimal but it should be close enough. We can see in Figure 3 that the three schemes converge to the same value as $N$ increases.

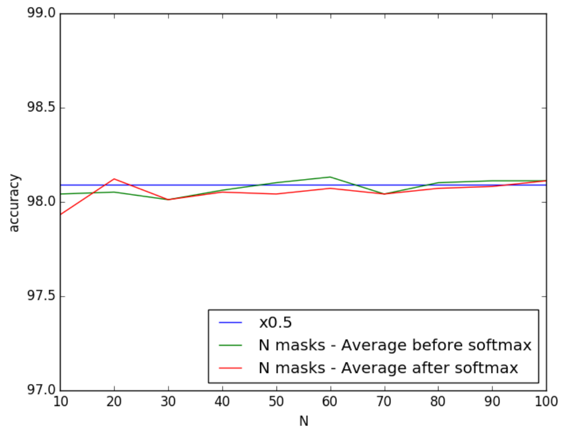

FIGURE 3. Accuracy of the MLP trained with dropout and evaluated using the proposed schemes in question 1 b).

(c) *Convolutional Networks*

The Figure 4 shows test errors of two CNN models with the following common architecture except that one model use dropout where the other use batch normalisation. See Appencide A for the structure of this models. We see in Figure 4 that batch normalization cause the model to learn faster. This is expected because batch normalization stabilize the distribution of the parameters and their gradiants leading to an improvment in the optimization. Furthermore, when using a big dataset such as MNIST, regularization is not as important as optimization since it is likely that the data is big enough to infer the true distribution of the inputs and ouputs, hence the chance of overfitting is low.

## 2. Dogs vs. Cats Classification

(a) For the dogs vs. cats classification challenge, we attempted a number of architectures inspired by the VGG architecture, but scaled down to fit on the hardware used for training (Nvidia GTX 760, 2 GB of RAM). All architectures made use of a set of Conv $\rightarrow$ Maxpool blocks followed by two fully connected layers (each with 1024 neurons) and finally a two node output softmax layer. All convolutional kernels were $3 \times 3$ and all maxpool layers were $2 \times 2$ with a stride of 2.
1) shallow: conv1 = 16 filters, conv2 = 32 filters, conv3 = 64 filters
2) shallow and fat: conv1 = 32 filters, conv2 = 64 filters, conv3 = 128 filters
3) deep: conv1 = 32 filters, conv2 = 64 filters, conv3 = 128 filters, conv4 = 64 filters, conv5 = 32 filters.

The number of parameters for the shallow model is $5,274,882$, the shallow and fat model is $9,539,074$, and the deep model is $1,373,986$. The parameter savings of the deep model is due to reducing the number of filters as we go deeper, reducing the number of parameters between the final convolutional layer and the first fully-connected layer.

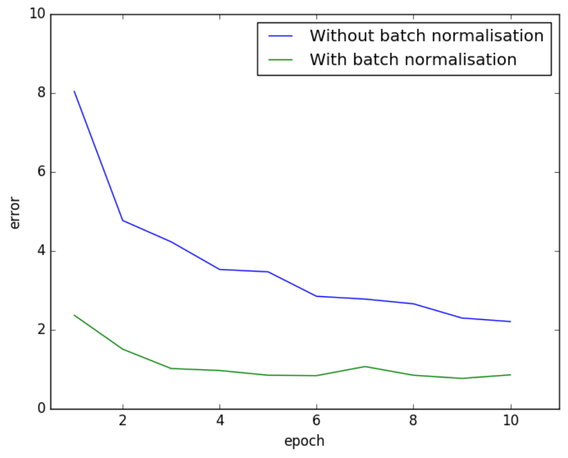

FIGURE 4. Test error of two convolutional neural networks; the blue line show the error of the CNN trained with dropout (and without batch normalisation); the blue line show the error of the CNN trained using batch normalisation (and no dropout).

(b) These three architectures were tested during a grid search. We tested batch sizes 4, 8, 16, 32, 64, and the three architectures above. We used the Adam optimizer with the default parameters and batchnorm, so we did not search over learning rate and learning rate decay. Furthermore we trained for 25 epochs and used the best validation score across all epochs to evaluate training performance (i.e., early stopping). The following training and validation error were obtained using the best performing model: batchsize = 16 and the big architecture.

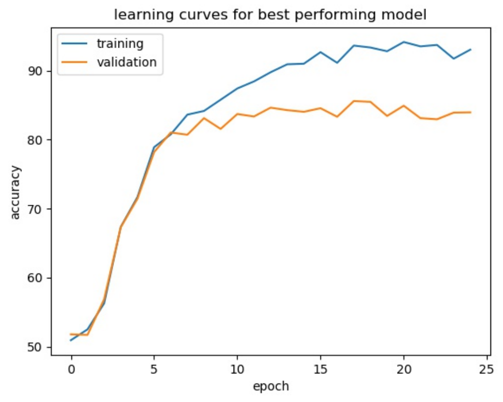

FIGURE 5. Training and validation accuracy for the best performing model.

The model clearly overfits around epoch = 10. The largest validation accuracy was found at epoch = 17 (acc=85.6%). The corresponding test accuracy was 84.5% and our generalization gap was 9.15%.

See Figure 6 for an example filter in the deep layers of our convolutional network. See Figure 7 for and analysis of the errors our model makes.

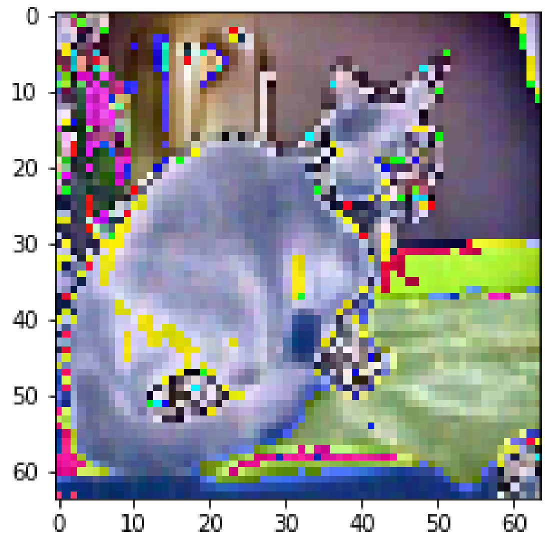

FIGURE 6. Example filter from the deep layers of our model

In some cases (2/36) the images mostly feature a person, which are an obvious challenge for our model. In many cases (10/36), the animal was cropped. It could be that our model needs to see the full animal in order to do well. In some cases (2/36) it isn't even clear to us what is in the image, the files are too low resolution. From this information, the obvious next step to improve the performance of our classifier would be to produce a large number of cropped images from our dataset, so the model would learn how to classify images where the entire animal is not present. This data augmentation step would take care of a large percentage of our errors.
The other images that were misclassified look like they should be easy to learn. What would likely help in this case would be a larger model and more time to train said model (i.e., more epochs). With more epochs and data augmentation we would likely be able to reach at least 90% accuracy.

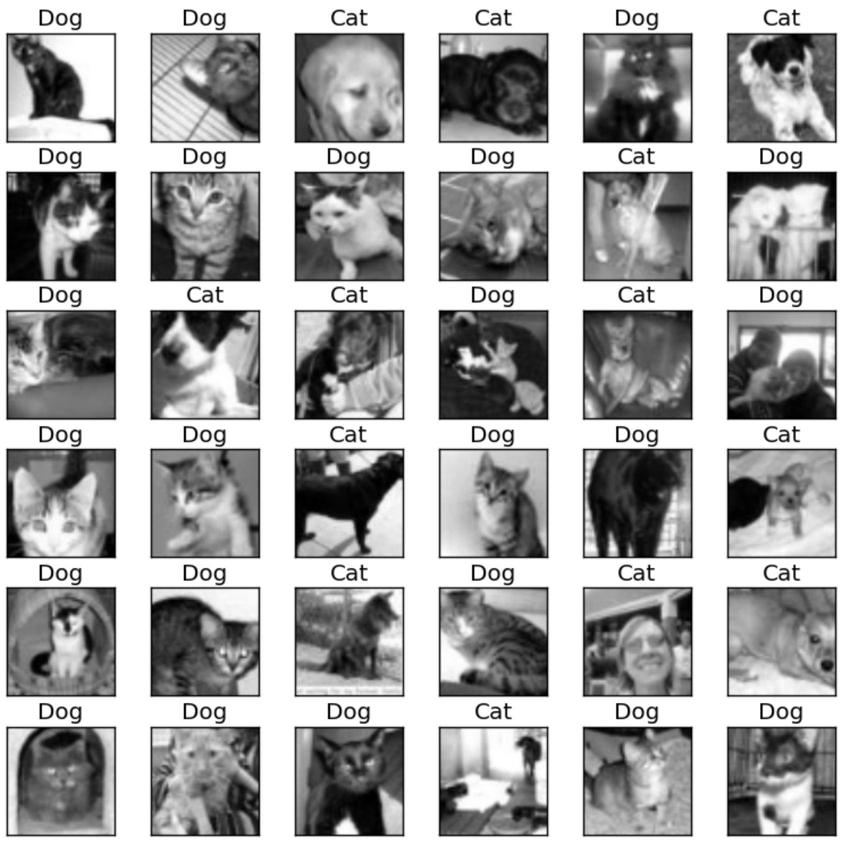

Figure 7. Examples of misclassified images.

## Appendix A. MLP Model

The MLP was implemented as a pytorch `Module` class. It accepts a list of hidden layer sizes, allowing the user to specify an MLP of any length using 3 variables: `h0` = input size, `hn` = hidden layer sizes (as a list), and `ho` = output size.

This class also contains an `initializer` method for initializing weights and `count_params` method for counting all model parameters).

```
Class MnistMLP(torch.nn.Module):
    """MLP classifier for MNIST"""
    def __init__(self, h0, hn, ho):
        """
        h0 -- input size (flat)
        hn -- a list of all hidden layer sizes
        ho -- output size (number of classes; flat)
        """
        super(MnistMLP, self).__init__()
        # input --> hid1
        architecture = [torch.nn.Linear(h0, hn[0]), torch.nn.ReLU()]

        # hidden layers
        for i in range(1, len(hn)):
            architecture.append(torch.nn.Linear(hn[i-1], hn[i]))
            architecture.append(torch.nn.ReLU())

        # output
        architecture.append(torch.nn.Linear(hn[-1], ho))

        # use nn to define model
        self.mlp = torch.nn.Sequential(*architecture)
        self.clf = torch.nn.LogSoftmax(dim=0)

    def forward(self, X):
        return(self.clf(self.mlp(X).squeeze()))

    def initalizer(self, init_type='glorot'):
        """
        model     -- a pytorch sequential model
        init_type -- one of 'zero', 'normal', 'glorot'

        Takes in a model, initializes it to all-zero, normal
            distribution
        sampled, or glorot initialization. Golorot == xavier.
        """
        if init_type not in ['zero', 'normal', 'glorot']:
            raise Exception('init_type invalid]')

        for k, v in self.mlp.named_parameters():
            if k.endswith('weight'):
                if init_type == 'zero':
                    torch.nn.init.constant(v, 0)
                elif init_type == 'normal':
                    torch.nn.init.normal(v)
                elif init_type == 'glorot':
                    torch.nn.init.xavier_uniform(v, gain=
                        calculate_gain('relu'))
                else:
                    raise Exception('invalid init_type')

    def count_params(self):
        """
        Returns a count of all parameters
        """
        param_count = 0
        for k, v in self.mlp.named_parameters():
            param_count += np.prod(np.array(v.size()))

        return(param_count)
```

## Appendix B. DeepBoi

The winning model can be seen below. It accepts an architecture in the variable conv_blocks, which is a list of tuples denoting (number of filters, number of repeats). In the end we only used a single repeat per block. All layers make use of batchnorm.

```
class DeepBoi(torch.nn.Module):
    """
    Convnet Classifier taken from "Very deep convolutional networks
        for
    large-scale image recognition"
    """
    def __init__(self, conv_blocks, in_channels=3, n_out=2):
        super(DeepBoi, self).__init__()

        # builds the convolutional layers
        conv_arch = []

        for block in conv_blocks:
            for convlayers in range(block[1]):

                # conv2d --> batchnorm --> relu, done n=convlayer
                    times / block
                conv_arch.extend([
                    torch.nn.Conv2d(in_channels=in_channels,
                        out_channels=block[0], kernel_size=(3, 3),
                            padding=1),
                    torch.nn.BatchNorm2d(block[0]),
                    torch.nn.ReLU()])

                in_channels = block[0]

            # add a maxpool layer between blocks
            conv_arch.append(torch.nn.MaxPool2d(kernel_size=(2, 2),
                stride=2))

        # builds the fully connected layers
        self.conv = torch.nn.Sequential(*conv_arch)
        test_data = Variable(torch.rand(1, 3, 64, 64))
        test_data = self.conv(test_data)
        init_layer = test_data.data.view(1, -1).size(1)

        fc_layers = [init_layer, 1024, 1024]
        fc_arch = []
        for i in range(1, len(fc_layers)):
            fc_arch.extend([
                torch.nn.Linear(fc_layers[i-1], fc_layers[i]),
                torch.nn.BatchNorm1d(fc_layers[i]),
                torch.nn.ReLU()])
        fc_arch.append(torch.nn.Linear(fc_layers[-1], n_out))

        # assemble the model
        self.fc = torch.nn.Sequential(*fc_arch)
        self.clf = torch.nn.LogSoftmax(dim=0)


    def forward(self, x):
        # input.view(self.size(0), -1)
        x = self.conv(x)
        dims = x.shape
        x = x.view(dims[1]*dims[2]*dims[3], -1).transpose(0, 1)
        return(self.fc(x))
```

```
def predict(self, X):
    return(self.clf(X))

# init_type here for compatibility with previous models but is
    ugly af
def initalizer(self, init_type=None):
    for k, v in self.conv.named_parameters():
        if k.endswith('weight') and len(v.shape) > 1:
            torch.nn.init.xavier_uniform(v, gain=calculate_gain('
                relu'))
```

## Appendix C. Training Loop

The implementation of the training loop can be seen below, and is used throughout the report:

```
CUDA = torch.cuda.is_available()

def run_experiment(clf, lr, epochs, loaders, momentum=False, init_type
    ='glorot'):

    if len(loaders) == 3:
        train, valid, test = loaders
    elif len(loaders) == 2:
        train, valid = loaders
        test = None
    else:
        raise Exception('loaders malformed')

    clf.initalizer(init_type=init_type)

    if CUDA:
        clf = clf.cuda()

    if momentum:
        optimizer = torch.optim.SGD(clf.parameters(), lr=lr, momentum
            =0.9)
    else:
        optimizer = torch.optim.SGD(clf.parameters(), lr=lr)

    #lossfn = torch.nn.CrossEntropyLoss() # don't use! B/C I specify
        LogSoftmax
    lossfn = torch.nn.NLLLoss()

    epoch_loss, valid_acc, train_acc = [], [], []
    best_valid_acc, gen_gap = 0, 0

    all_losses = []
    for ep in range(epochs):

        epoch_losses = []

        # training data
        for batch_idx, (X_train, y_train) in enumerate(train):

            if CUDA:
                X_train, y_train = X_train.cuda(), y_train.cuda()

            # initalize batch
            optimizer.zero_grad()
            X_train, y_train = Variable(X_train), Variable(y_train)

            # make predictions -- flatten each image (batchsize x
                pixels)
            train_pred = clf.forward(X_train.view(X_train.shape[0],
                -1))
            # calculate loss (cross entropy)
            loss = lossfn(train_pred, y_train)
            epoch_losses.append(loss.data[0])
            all_losses.append(loss.data[0])

            # calculate dloss/dx for all parameters that have
                requires_grad
            loss.backward()

            # update paramater values
            optimizer.step()

        # average loss for epoch
        epoch_loss.append(np.mean(epoch_losses))

        # validation accuracy for this epoch
        this_valid_acc = evaluate(clf, valid)
        valid_acc.append(this_valid_acc)

        # training accuracy for this epoch
        this_train_acc = evaluate(clf, train)
        train_acc.append(this_train_acc)

        # keep track of the best validation accuracy, generalization
            gap
        if this_valid_acc > best_valid_acc:
            best_valid_acc = this_valid_acc

            if test:
                this_test_acc = evaluate(clf, test)
                gen_gap = this_train_acc - this_test_acc

        # update every n epochs
        if (ep+1) % 5 == 0:
            curr_lr =  optimizer.state_dict()['param_groups'][0]['lr']
            print('+ [{:03d}] loss={:0.6f} acc={:0.2f}/{:0.2f} lr
                ={:0.5f}'.format(
                ep+1, epoch_loss[-1], train_acc[-1], valid_acc[-1],
                    curr_lr))

    results = {'clf': clf,
               'epoch_loss': epoch_loss,
               'all_loss': all_losses,
               'train_acc': train_acc,
               'valid_acc': valid_acc,
               'best_valid_acc': best_valid_acc,
               'gen_gap': gen_gap}

    return(results)
```