IFT6135 – Homework #2

Regularization, weight decay, batch normalization convolutional neural network – programming part

Iban Harlouchet 1034545

Jean-Philippe Reid 701300

March 16, 2018

Regularization : weight decay, early stopping, dropout, domain prior knowledge

(a) *Early stopping and weight decay* – We show in Fig.1 the L2 norm of all the parameters plotted at each iteration during the training of both models, i.e. with and without regularization. It is important to note that the L2 norm corresponds to the sum of all the weight norms of the model. See the appendice A.1 for more details.

We also show in Fig.2 the total number of each epoch of training, as instructed. The coefficient λ was rescaled in order to adapt for the mini batch size,

```
weight_decay = 2.5 * batch_size / train_data.shape[0].
```

To emphasize the effect of regularization, we show in the table 1 the error rates for the train, valid and test datasets for both settings.

General notes:

- The main effect of regularization is to prevent overfitting by penalizing the square value of all the weights. This tends to decrease the weights to smaller values, as shown in Fig. 1.
- The regularization has a large effect on the training datasets as the error rate increases by a factor 30 (see table 1). This is in contrast with the validation and test datasets where the error rates with and without regularization are of the same order of magnitude.

  It is important to note that, with the regularization, the error rates for the training and valid datasets are asymptotically similar. This suggests that the regu-

larization helps to generalize well from the training data in order to make good predictions.

(b) *Dropout* –We show in Fig 3 the model accuracy as a function of the number of dropout masks over which the model outputs were averaged. The average was computed before and after the softmax, as instructed. We observe that both the methods yield to the same numerical results, and converges with N to the prediction with no dropout. This is to be expected as, when the number of samples N increases, the randomness introduced by the dropout masks averaged on the number of samples tends to vanish (to become quasi-deterministic). This is a consequence of the central limit theorem.

The effect of dropout is to mask some weights randomly, thus affecting the capacity of the model. By doing so, the prediction accuracy is poor if we keep the dropout activated.

Averaging multiple times the output of a model with the dropout activated is equivalent to the **bagging ensemble methods**. This method has for main objective to decrease the variance of all the predictions.

General notes:

- As indicated in the function `MLPLinear.forward()` (see appendix B), we included a dropout on the last hidden layer.
- The model was trained over 100 epochs with a dropout of 0.5. Before testing the capacity of the model, we multiplied the weights of the hidden layer by 0.5 to counterbalance the dropout upon validation. We refer the reader to the function `set_parameters()` presented in the appendix A.3 for more details.
- As instructed, we implemented the function `N_predictions()` (see appendix A.3) which computes the mean of the model outputs before and after the softmax activation function.

(c) *Convolutional neural network* – We show in Fig.4 the error rates with computed at the end of each epoch, as instructed. We summarize our results in table 2.

General notes:

- The convolutional and linear layers were initialized with the Kaiming and Glorot initializations respectively. See appendix A.4 for more details.

Table 1: Error rates on the classification of MNIST dataset with a two-layer perceptron (MLP) with and without regularization. See Annexe B for more details about the algorithm.

	Train dataset	Valid dataset	Test dataset
No regularization	0.13%	2.06%	1.97%
Regularization	2.91%	2.98%	3.27%

Table 2: Errors on the classification of MNIST dataset with a convolutional neural network. The model was trained over 10 epochs. See Annexe D for more details about the algorithm.

	Train accuracy	Valid accuracy	Test accuracy
No batch normalization	6.14%	5.71%	5.96%
Batch normalization	3.46%	3.05%	3.01%

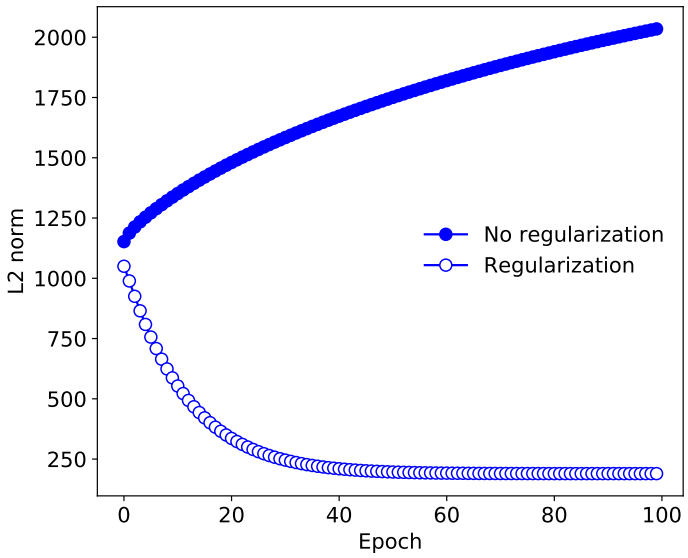

Figure 1: L2 norm of all the parameters of the models plotted as a function of epochs for the two-layer perceptron. Open and close symbols correspond the model with and without regularization.

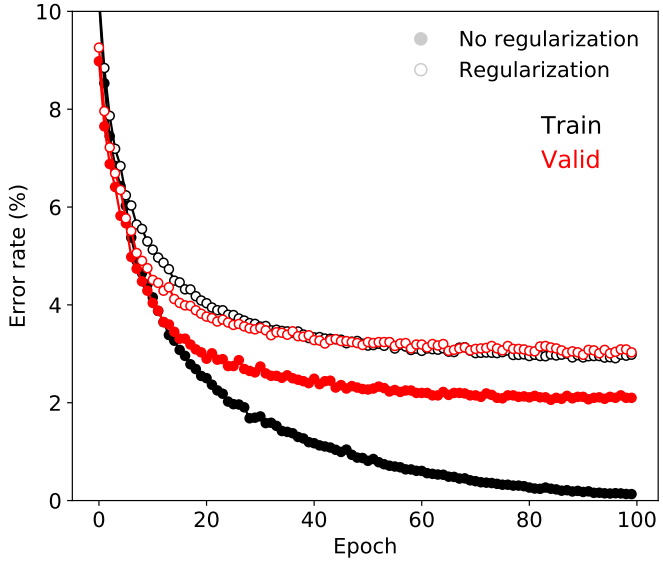

Figure 2: The error rate plotted as a function of epochs for the two-layer perceptron. Open and close symbols correspond the model with and without regularization.

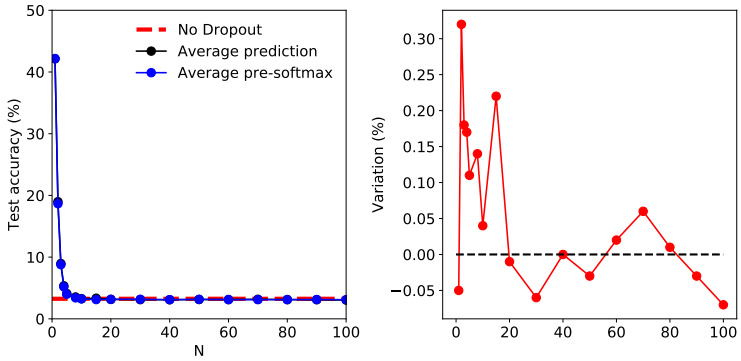

Figure 3: (left) Accuracy as a function of epochs for the two-layer perceptron. The blue and black circles represent the average prediction on the test dataset before and after the softmax function respectively. The red horizontal line corresponds to the prediction with no dropout. (right) The difference between the blue and the black curves presented in the left panel.

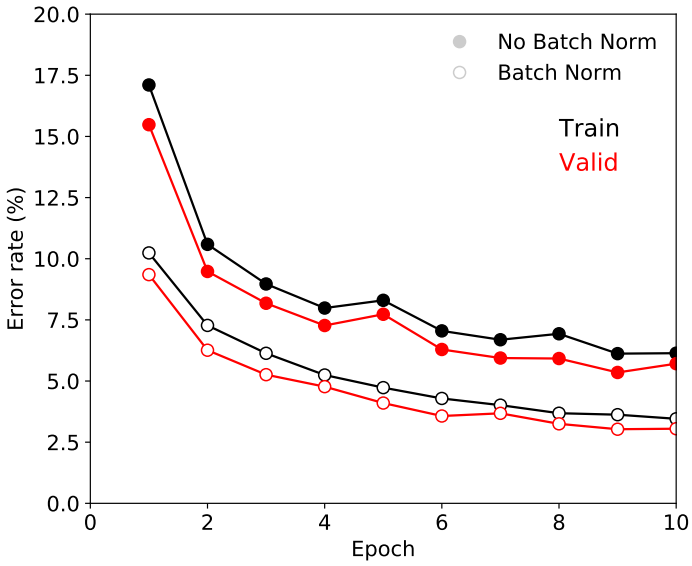

Figure 4: Errors plotted as a function of epochs for the convolutional neural networks. Open and close symbols correspond the model with and without batch normalization respectively.

Convolutional neural network: Dogs vs. Cats Classification

- The data was separated in three sets: training, validation and test datasets. We took 2×10000 pictures of cats and dogs for training and validating. From this set, we sampled 2×2000 pictures for the validation dataset. We took the last 2×2500 pictures for the test dataset.
- Each data set was standardized, i.e.

$$\mathrm{x} \to \frac{\mathrm{x} - \mu}{\sigma} \tag{2.1}$$

where μ and σ are the mean and standard deviation of each dataset. The reader should appreciate that the training, validation and test datasets were standardized with their respective μ and σ.

$$\mu_{\text{train}} = [0.4902, 0.4550, 0.4164] \qquad \sigma_{\text{train}} = [0.2519, 0.2449, 0.2471] \tag{2.2}$$
$$\mu_{\text{valid}} = [0.4880, 0.4531, 0.4144] \qquad \sigma_{\text{valid}} = [0.2527, 0.2459, 0.2479] \tag{2.3}$$
$$\mu_{\text{test}} = [0.4922, 0.4577, 0.4183] \qquad \sigma_{\text{test}} = [0.2536, 0.2471, 0.2487] \tag{2.4}$$

(a) *Describe the architecture (number of layers, filter sizes, pooling, etc.), and report the number of parameters. You can take inspiration from some modern deep neural network architectures such as the VGG networks to improve the performance*

- We built four different convolutional neural networks (CNN), labelled A, B, D and E in Fig.5. These models have the very same structure as that presented by K. Simonyan *et al.* arXiv :1409.1556, 2014.

  The number of parameters for each model are:

  - Model A: 10.31M
  - Model B: 10.49M
  - Model D: 15.81M
  - Model E: 20.53M

  Each CNN layer is followed by a batch normalization, a ReLU activation function and, optionally, a max-pooling (see K. Simonyan *et al.*). Fully connected (FC) linear layers are added on the output of the last CNN to adjust the dimension for the problem in hand.

  The models A, B, D and E are detailed in the `Q2_model_x.py` in the supplementary materials.

  Additionally to these models, we also implemented multiple other models with different minor characteristics. Still, their performances are dramatically different.

We show here a list of differences between each of them. We will use different colours for each these models, namely 'black' and 'blue'. The result of these models are shown in the table 3.

- The fully connected linear layers are of different dimensions. In the models A, B, D and E,

  $$\text{Linear-}2048 \times 64,\ \text{ReLU},\ \text{Dropout}$$
  $$\text{Linear-}64 \times 64,\ \text{ReLU},\ \text{Dropout}$$
  $$\text{Linear-}64 \times 2.$$

  The dimensions for the models A, B, D and E are shown in the figure 5.

- In the models A, B, D and E, we included a batch normalization layer after the first linear layer. This is in contrast with A, B, D and E where we used a batch normalization after the first two linear layers.

- The numbers of parameters are:

  * Model A: 9.36M
  * Model B: 9.54M
  * Model D: 14.85M
  * Model E: 20.17M

We also implemented several "naive" models (i.e., not inspired by a classical model) called `conv1`, `conv2`, `conv3`. The results are detailed on online jupyter notebook **Q2.html** attached to this document. To say a few words about these models, they have 1.60M, 1.60M and 1.58M parameters respectively, and their test errors are 15.22%, 15.70% and 13.14% respectively.

Finally, we implemented a "last chance model" called `convXf` (see jupyter notebook **Q2-Last chance model.html** in the supplementary materials) with 123.64M parameters. To avoid memory shortage, we had in this case to restrict ourselves to 1 GPU alone with 125GB memory, and a batchsize equal to 20). This model consists of the pretrained vgg19_bn model with the last linear layer modified so as to output 2 real numbers instead of 1000. We had to rescale the images to dimension 224×224 so as to fit the model. We got a test error rate of 9.32%.

- We initialized the CNNs with two methods :

  (a) Kaiming initialization designed to train extremely deep rectified models directly [Kaiming He *et al.*, arXiv:1502.01852]. See the appendix A.4 for more details.

  (b) the pre-trained VGG-XX, CNNs trained on the image-net dataset. See the appendix D for more details.

The FC linear layers are initialized with the Xavier method. See the appendix A.4 for more details.

- The optimizing method is constructed with two internal optimizers with different learning rates. We used the `MultipleOptimizer` class.

  We further expand our exploration by considering two different optimizers: `SGD` and `Adam`. Both performed well, but Adam showed a better robustness. See the appendix A.5 for more details.

  Of course, the performance of the models depend on the learning rate and momentum. The optimal parameters are

  - Kaiming initialization: `lr_fc` $= 1E-2$, `momentum`$= 0.9$,
  - Pre-trained initialization: `lr_cnn` $= 1E-4$, `momentum`$= 0.9$.

(b) *Plot the training error and validation error. You can use the optimization techniques you learned in the class, such as different optimizers or feature normalization, to accelerate training. In particular, try Batch Normalization.*

- We show in Fig.6 the training and validation error rates for the model E. We include the corresponding figures in the appendix E.
- See also **Q2.html** in the supplementary materials for additional plots.

(c) *Compare different hyperparameter settings and report the final results of performance on test set. Aside from quantitative results, also include some visual analysis such as visualizing the feature maps or kernels, or showing examples where the images are (a) clearly misclassified and (b) where the classifier predicts around 50% on both classes. Explain your observation and/or suggest any improvements you think may help.*

- We show in table 3 the performance of the 2×4 models (black & blue) on the train, validation and test datasets. As mentioned above, we tested two different initialization methods, i.e. kaiming and pre-trained initializations. We used the early stopping method for all the models and the initialization methods we considered.
- We also present in Fig. 7 some feature maps of the first CNN layer along with the corresponding three filters. These results were taken from the model E parameters. We refer the reader to the function `model.selec_layer()` in the `Q2_model_e.py` in the supplementary materials.
- Finally, we present in Fig. 8 some examples where the model misclassified cats and dogs. We also show in Fig.9 examples where the model has a hard time to make a prediction. We used the model E in both cases.

Discussion:

- The pre-trained model converges faster than the one initialized with kaiming (see Fig. 6). This suggests that transfer learning technique works, even if the transferred model were trained on a different dataset (ImageNet).
- As mentioned above, the kernel filters enhance certain features of the original image. For instance, the horizontal lines are accentuated in Fig.7(b). This is due to kernel filter (j) which displays a horizontal line.
- The misclassification of cats and dogs may be explained by several factors. Some animals are just non-identifiable (see upper left image in Fig. 8), but others may look like the opposite class, e.g. dogs with cats' ears (see upper left panel in Fig.8).
- Training convolutional neural models of $\sim$ 20M parameters require a large quantity of data. The training dataset considered here contains $16K$ images. In order to improve the training of the model, we could have implemented data augmentation.

Table 3: Error rates (%) on the classification of Cats and Dogs dataset for all models considered, i.e. A, A, B, B, D, D, E, E (2×4)

	Train dataset	Valid dataset	Test dataset
Model A - kaiming	0.07(0.13)	7.75(12.78)	8.74(13.12)
Model A - pre-trained	0.11(0.10)	5.92 (6.12)	6.86 (6.56)
Model B - kaiming	0.28(0.31)	6.43(11.17)	6.72(11.82)
Model B - pre-trained	0.12(2.13)	5.77(6.15)	6.40 (6.88)
Model D - kaiming	0.09(0.36)	6.55(10.87)	7.48 (10.24)
Model D - pre-trained	0.12(0.11)	5.75(5.82)	6.30 (5.94)
Model E - kaiming	0.34(0.60)	6.60(10.80)	7.38(11.50)
Model E - pre-trained	0.25(0.03)	6.35(5.03)	6.76(5.36)

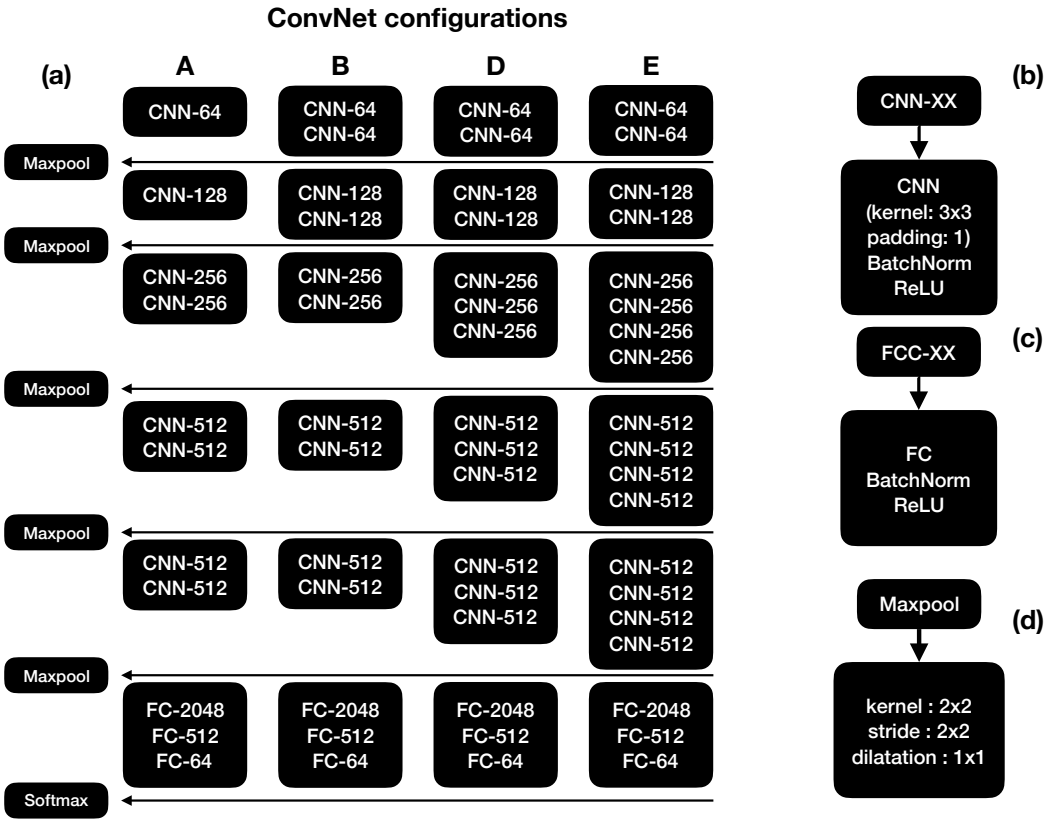

Figure 5: (a) Convolutional neural network(CNN) configurations, composed of stacks of CNNs with different kernel dimensions (64, 128, 256, and 512). (b) Each CNN block is composed of convolutional layer with kernel size of 3×3, padding of 1 and stride of 1 followed by a batch normalization and ReLU activation function. (c) Idem as (b), but for a fully connected linear layer (FC). (d) The maxpool, with filter size of 2×2, stride of 2×2 and dilatation of 1×1. We also trained similar models, but with different structures. These differences are listed in the text and the results are indicated in blue in the table 3.

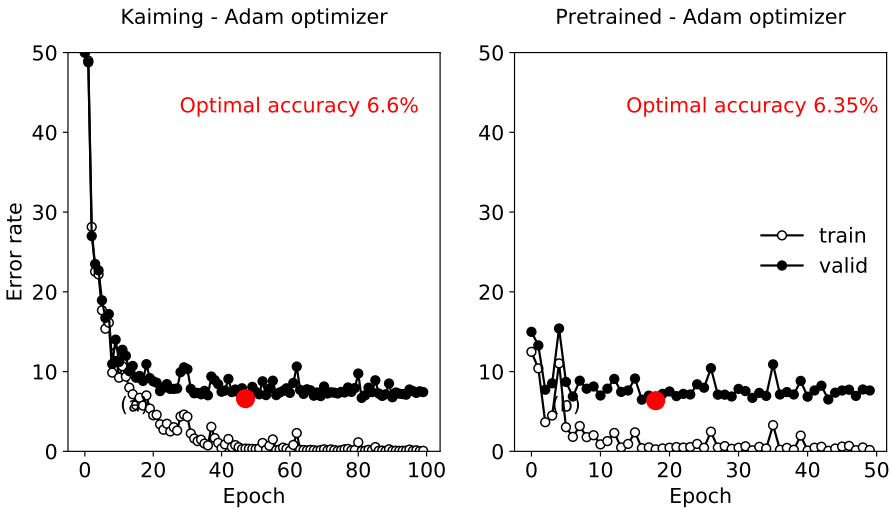

Figure 6: Training and validation errors for the model E, i.e. VGG-19. (a) Kaiming and (b) pre-trained initialization. The optimal accuracy are inidicated in both panels (red circles). We used batch normalization for both experiments. Similar figures for the model A, B and D are included in the appendix E.

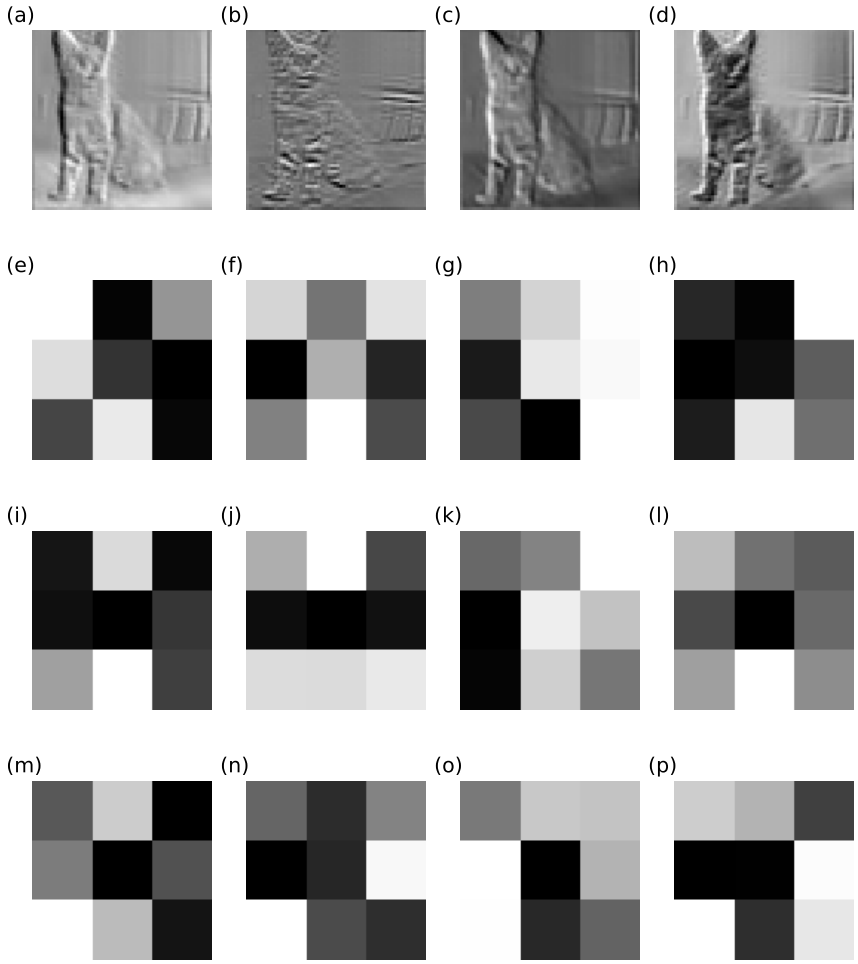

Figure 7: (a-d) Feature maps of cats. (e-p) Kernel associated. (e-h) 'red' channels, (e-h) 'green' channels and (e-h) 'blue' channels.

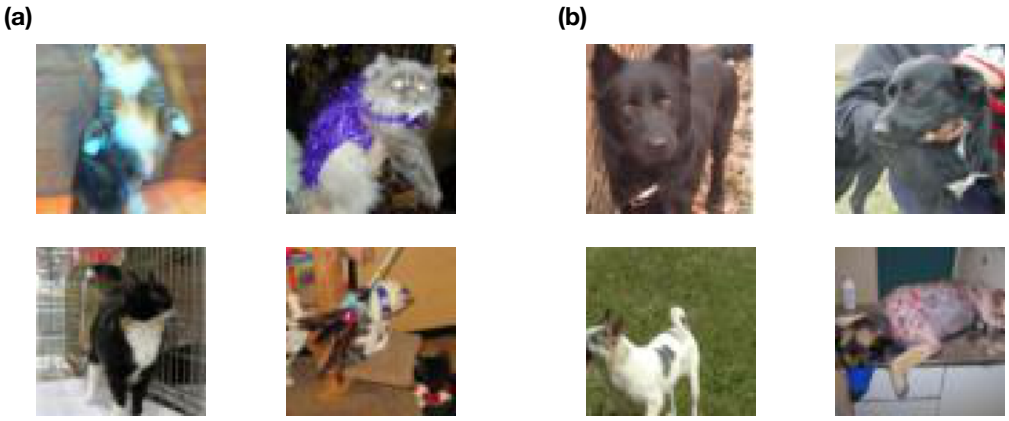

Figure 8: The misclassification of cats and dogs.

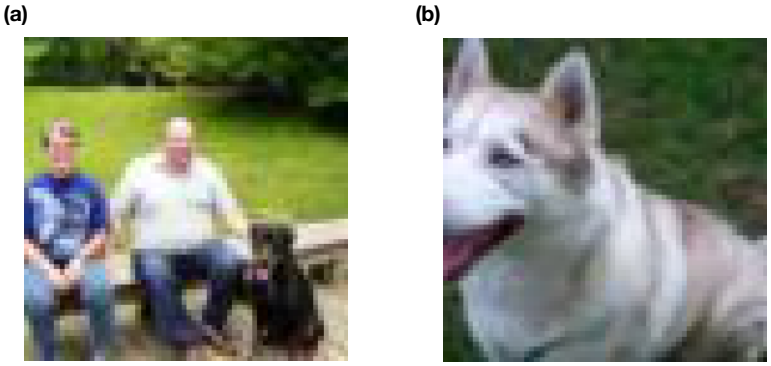

Figure 9: (a) The model E predicts that this lovely animal beside the enormous man is a dog with a probability of 54.3%. (b) Model E predicts that the animal is a dog with a probability of 51.9%.

Appendice A : Utility functions

A.1 L2-Norm

```
def L2_weights_norm(model):
    weights = []
    for name, parameter in model.named_parameters():
        if 'weight' in name:
            weights.append(parameter)
    return (torch.norm(weights[0]).data[0] + torch.norm(weights
    [1]).data[0])**2
```

A.2 Weights scaling

```
def set_parameters(model,param,factor):
    for name, p in model.named_parameters():
        if param in name:
            p.data =  factor * p.data
```

A.3 Predictive models for dropout

```
def N_prediction(data_batch, model, N, method):
    softmax = nn.Softmax(dim = 1)
    correct = 0
    total = 0
    for x,y in data_batch:
        out = 0
        x,y = model.input_shape(x,y)

    # Different methods for evaluating predictions.
    if method == 'before_activation':
        for i in range(N):
            out += model(x)
        out = softmax(out/N)

    elif method == 'after_activation':
        for i in range(N):
            out += softmax(model(x))
        out = out/N

        _, pred = torch.max(out.data, 1)
        pred = Variable(pred)
        correct += float((pred == y).sum())
        total += float(y.size(0))
    return 100*correct/total, total - correct
```

A.4 Initialization

```
def weights_init_1d(m):
    classname = m.__class__.__name__

    if isinstance(m, nn.BatchNorm1d):
        m.weight.data.normal_(0, 1)
        m.bias.data.fill_(0)

    elif isinstance(m, nn.Linear):
        nn.init.xavier_uniform(m.weight.data,
                gain=calculate_gain('relu'))
        nn.init.constant(m.bias.data, 0)

def weights_init_2d(m):
    classname = m.__class__.__name__

    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_uniform(m.weight.data, mode='fan_in')
        nn.init.constant(m.bias.data, 0)
    elif isinstance(m, nn.BatchNorm2d):
        m.weight.data.normal_(0, 1)
        m.bias.data.fill_(0)
```

A.5 Multi-optimization

```
class MultipleOptimizer(object):
  def __init__(self, *op):
    self.optimizers = op

  def zero_grad(self):
    for op in self.optimizers:
      op.zero_grad()

  def step(self):
    for op in self.optimizers:
      op.step()
```

```
def opt_func(opt):
  if opt == 'adam':
    optimizer = MultipleOptimizer(
      optim.Adam(model.features.parameters(), lr=0.001),
      optim.Adam(model.classifier.parameters(), lr=0.01))

  elif opt == 'sgd':
    optimizer = MultipleOptimizer(
      optim.SGD(model.features.parameters(),
        lr=0.01, momentum=0.9),
      optim.SGD(model.classifier.parameters(),
        lr=0.01, momentum=0.9))
  return optimizer
```

Appendice B : Two layer perceptron

```
class MLPLinear(nn.Module):
  def __init__(self, dimensions, dp, cuda):
    super(MLPLinear, self).__init__()
    self.h0 = int(dimensions[0])
    self.h1 = int(dimensions[1])
    self.h2 = int(dimensions[2])

    self.fc1 = torch.nn.Linear(self.h0, self.h1)
    self.fc2 = torch.nn.Linear(self.h1, self.h2)
    self.fc2_drop = nn.Dropout(p=dp)
    self.relu = nn.ReLU()
    self.criterion = nn.CrossEntropyLoss()
    self.cuda = cuda

    if cuda:
      self.fc1.cuda()
      self.fc2.cuda()
      self.fc2_drop.cuda()
      self.relu.cuda()
      self.criterion.cuda()

  def initialization(self,method):
    self.fc1 = linear_ini(self.fc1,method)
    self.fc2 = linear_ini(self.fc2,method)

  def input_shape(self,x,y):
    x = (x.view(-1,784))
    if self.cuda :
      x = Variable(x.cuda())
      y = Variable(y.cuda())
    else:
      x = Variable(x)
      y = Variable(y)
    return x,y

  def forward(self, x):
    out = self.fc1(x)
    out = self.relu(out)
    out = self.fc2(out)
    out = self.fc2_drop(out)
    return  out
```

Appendice C : Convolutional neural network

```
class Convolution(nn.Module):
  def __init__(self):
    super(Convolution, self).__init__()

    self.criterion = nn.CrossEntropyLoss()
    self.conv = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=16,
          kernel_size=(3, 3), padding=1),
    nn.BatchNorm2d(16, eps=1e-05, momentum=0.1,
          affine=True),
    nn.Dropout(p=0.5),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2, 2), stride=2),

    nn.Conv2d(in_channels=16, out_channels=32,
          kernel_size=(3, 3), padding=1),
    nn.BatchNorm2d(32, eps=1e-05, momentum=0.1,
          affine=True),
    nn.Dropout(p=0.5),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2, 2), stride=2),

    nn.Conv2d(in_channels=32, out_channels=64,
          kernel_size=(3, 3), padding=1),
    nn.BatchNorm2d(64, eps=1e-05, momentum=0.1,
          affine=True),
    nn.Dropout(p=0.5),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2, 2), stride=2),

    nn.Conv2d(in_channels=64, out_channels=128,
          kernel_size=(3, 3), padding=1),
    nn.BatchNorm2d(128, eps=1e-05, momentum=0.1,
          affine=True),
    nn.Dropout(p=0.5),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=(2, 2), stride=2)
    )

    self.fc1 = nn.Linear(128, 10)

  def input_shape(self,x,y):
    return x, y

  def initialization(self,method):
    self.fc1 = linear_ini(self.fc1,method)

  def forward(self, x):
    return self.fc1(self.conv(x).squeeze())
```

Appendice C : VGG-Pretrained – Model E

```
1
2
3  class Classifier_pretrained(nn.Module):
4    def __init__(self):
5      super(Classifier_pretrained, self).__init__()
6
7      vgg19_bn = models.vgg19_bn(pretrained=True)
8      self.features = nn.Sequential(*list(vgg19_bn.children())[0])
9
10     self.classifier = nn.Sequential(
11       nn.Linear(2048, 512),
12       nn.BatchNorm1d(512),
13       nn.Dropout(p=0.5),
14       nn.ReLU(),
15       nn.Linear(512, 64),
16       nn.BatchNorm1d(64),
17       nn.Dropout(p=0.5),
18       nn.ReLU(),
19       nn.Linear(64, 2))
20
21   def forward(self, x):
22     out = self.features(x)
23     out = self.classifier(out.view(out.size(0),-1))
24     return out
```

Appendice D : Performace of model A, B, and D

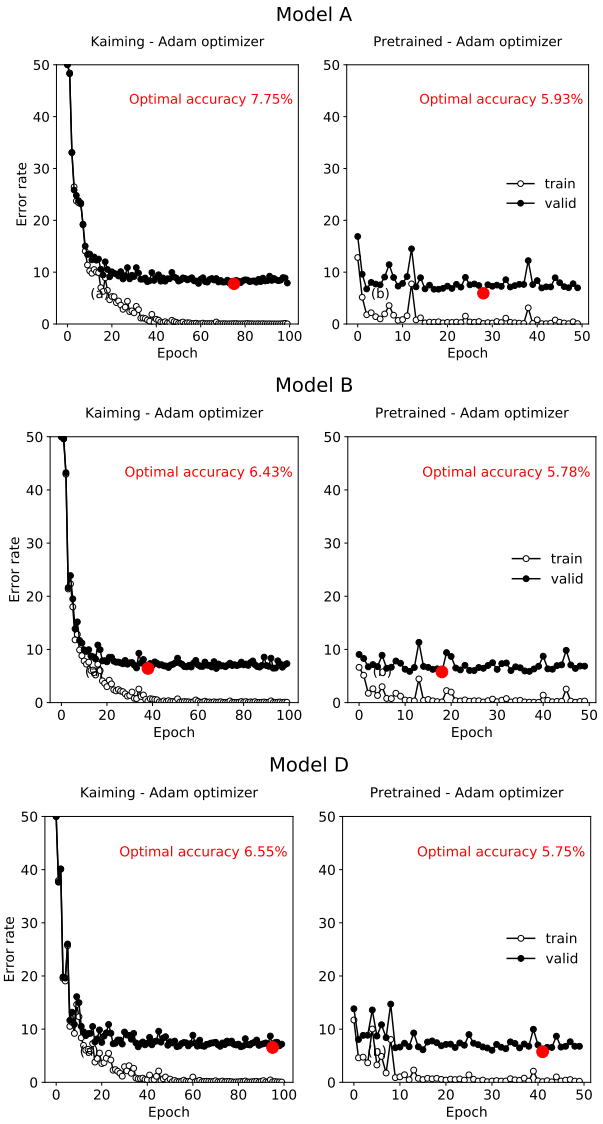

Figure 10: Training and validation errors for the model A, B and C, i.e. VGG-11-13-16. (left) Kaiming and (right) pre-trained initializations. The optimal accuracies on the validation dataset are indicated in both panels (red circles).