Assignment 4

Assignment 4: Neural Networks

Goal: Get familiar with neural networks by implementing them and applying them to image classification.

In this assignment we are going to learn about neural networks (NNs). The goal is to implement two neural networks: a fully-connected neural network, a convolutional neural network, and analyze their behavior.

The considered task is image classification. We consider a dataset of small natural images (see the additional file) with multiple classes. We aim at formulating a model (a neural network) and learning it using the negative log-likelihood function (i.e., the cross-entropy loss) as the objective function, and the stochastic gradient descent as the optimizer.

In this assignment, **the code must be implemented in PyTorch**.

1 Understanding the problem

The considered problem is about classifying images to L classes. In the first part of the assignment, you are asked get familiar with PyTorch, a deep learning library, and the basics of neural networks, and implement neural-network-based classifiers. For this purpose, we will start with classifying small images (8px x 8px) of handwritten digits to one of 10 classes. The dataset is very small and all experiments could be achieved within a couple of minutes.

In the second part, you are asked to implement the whole pipeline for a given dataset by yourself.

Please run the code below and spend a while on analyzing the images.

If any code line is unclear to you, please read on that in numpy, scipy, matplotlib and PyTorch docs.

In [40]:
```
import os

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn import datasets
from sklearn.datasets import load_digits
from torch.utils.data import DataLoader, Dataset
```

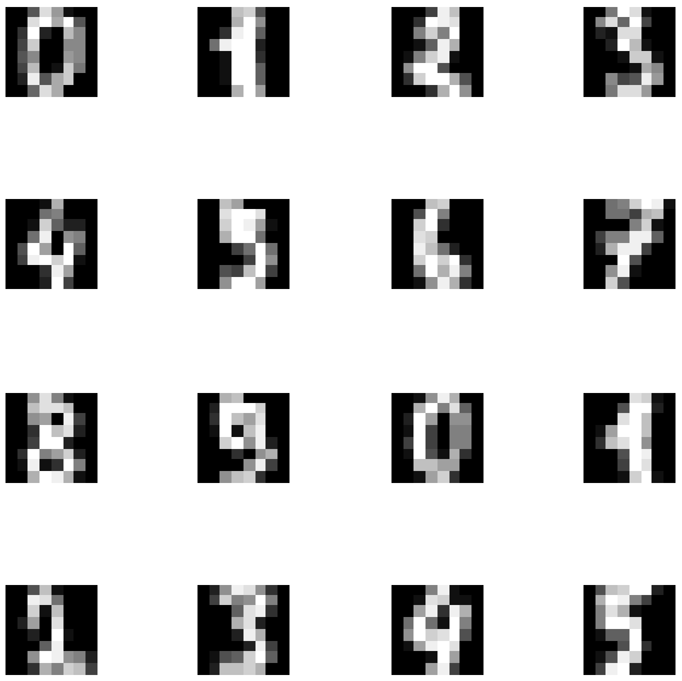
```
EPS = 1.0e-7
```

In [41]:
```
# IF YOU USE COLAB, THIS IS VERY USEFUL! OTHERWISE, PLEASE REMOVE IT.
# mount drive: WE NEED IT FOR SAVING IMAGES!
#from google.colab import drive

#drive.mount("/content/gdrive")
```

In [42]:
```
# IF YOU USE COLAB, THIS IS VERY USEFUL! OTHERWISE, PLEASE REMOVE IT.
# PLEASE CHANGE IT TO YOUR OWN GOOGLE DRIVE!
#results_dir = "/content/gdrive/My_Drive/Colab Notebooks/TEACHING/"
results_dir = r"\Users\hghol\OneDrive\Desktop\CI_Assignment_4"
```

In [43]:
```
# PLEASE DO NOT REMOVE!
# This is a class for the dataset of small (8px x 8px) digits.
# Please try to understand in details how it works!
class Digits(Dataset):
    """Scikit-Learn Digits dataset."""

    def __init__(self, mode="train", transforms=None):
        digits = load_digits()
        if mode == "train":
            self.data = digits.data[:1000].astype(np.float32)
            self.targets = digits.target[:1000]
        elif mode == "val":
            self.data = digits.data[1000:1350].astype(np.float32)
            self.targets = digits.target[1000:1350]
        else:
            self.data = digits.data[1350:].astype(np.float32)
            self.targets = digits.target[1350:]

        self.transforms = transforms

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample_x = self.data[idx]
        sample_y = self.targets[idx]
        if self.transforms:
            sample_x = self.transforms(sample_x)
        return (sample_x, sample_y)
```

In [44]:
```
# PLEASE DO NOT REMOVE
# Here, we plot some images (8px x 8px).
digits = load_digits()
x = digits.data[:16].astype(np.float32)

fig_data, axs = plt.subplots(4, 4, figsize=(4, 4))
fig_data.tight_layout()

for i in range(4):
    for j in range(4):
        img = np.reshape(x[4 * i + j], (8, 8))
        axs[i, j].imshow(img, cmap="gray")
        axs[i, j].axis("off")
```



2 Neural Networks for Digits (4pt)

In this assignment, you are asked to implement a neural network (NN) classifier. Please take a look at the class below and fill in the missing parts.

NOTE: Please pay attention to the inputs and outputs of each function.

2.1 Neural Network Classifier

Below, we have two helper modules (layers) that can be used to reshape and flatten a tensor. They are useful for creating sequentials with convolutional layers.

In [45]:

```
# PLEASE DO NOT REMOVE!
# Here are two auxiliary functions that can be used for a convolutional NN (CNN).


# This module reshapes an input (matrix -> tensor).
class Reshape(nn.Module):
    def __init__(self, size):
        super(Reshape, self).__init__()
        self.size = size  # a list

    def forward(self, x):
        assert x.shape[1] == np.prod(self.size)
        return x.view(x.shape[0], *self.size)


# This module flattens an input (tensor -> matrix) by blending dimensions
# beyond the batch size.
class Flatten(nn.Module):
    def __init__(self):
        super(Flatten, self).__init__()

    def forward(self, x):
        return x.view(x.shape[0], -1)
```

Below is the main class for a classifier parameterized by a neural network.

In [46]:
```
# =========
# GRADING:
# 0
# 0.5 pt if code works but it is explained badly
# 1.0 pt if code works and it is explained well
# =========
# Implement a neural network (NN) classifier.
class ClassifierNeuralNet(nn.Module):
    def __init__(self, classnet):
        super(ClassifierNeuralNet, self).__init__()
        # We provide a sequential module with layers and activations
        self.classnet = classnet
        # The loss function (the negative log-likelihood)
        self.nll = nn.NLLLoss(reduction="none")  # it requires log-softmax as input

    # This function classifies an image x to a class.
    # The output must be a class label (long).
    def classify(self, x):
        # ------
        # PLEASE FILL IN
        # y_pred = ...
        with torch.no_grad():
            logits = self.classnet(x) #defines logits from classnet
            probabilities = F.log_softmax(logits, dim=1) #set probabilities using l
            # Choose the class with maximum probability
            _, y_pred = torch.max(probabilities, 1) #set y predecessor equal to max
        return y_pred

    # This function is crucial for a module in PyTorch.
    # In our framework, this class outputs a value of the loss function.
    def forward(self, x, y, reduction="avg"):
        # ------
        # PLEASE FILL IN
        # loss = ...
        # ------
        logits = self.classnet(x) # defines logits variable from classnet(x)
        probabilities = F.log_softmax(logits, dim=1) # calculates probabilities usi
        loss = self.nll(probabilities, y.long()) # uses negative log-likelihood los

        if reduction == "sum":
            return loss.sum() # return sum is needed
        else:
            return loss.mean() # if reduciton does not equal sum, return the mean o
```

Question 1 (0-0.5pt): What is the objective function for a classification task? In other words, what is nn.NLLLoss in the code above? Pelase write it in mathematical terms.

Answer: For a classification task, an objective function is used to minimize negative log-likelihood (NLL) loss. This can be represented as

NLLloss(input, target)= - (1/N) ∑ (i=1:N) log(input_i,target)

where input = input tensor w/ log probabilities target = target tensor w/ true labels N = batch size input_i,target = log probability of true class for i-th sample

Question 2 (0-0.5pt): In the code above, it is said to use the logarithm of the softmax as the final activation function. Is it correct to use the log-softmax instead of the softmax for making predictions (i.e., picking the most probable label).

Answer: It is correct to use the log-softmax sunction instead of the softmax function for making predictions.

This is because the log-softmax function represents the predictions over a logarithmic scale. Since these probabilities all multiply, the independent events add together and are therefore better for computations. Logmax can also handle more extreme values.

Softmax is a simpler way to calculate a gradient, but often not as effective.

2.2 Evaluation

In [47]:
```python
# PLEASE DO NOT REMOVE
def evaluation(test_loader, name=None, model_best=None, epoch=None):
    # If available, load the best performing model
    if model_best is None:
        model_best = torch.load(name + ".model")

    model_best.eval()  # set the model to the evaluation mode
    loss_test = 0.0
    loss_error = 0.0
    N = 0.0
    # start evaluation
    for indx_batch, (test_batch, test_targets) in enumerate(test_loader):
        # loss (nll)
        loss_test_batch = model_best.forward(test_batch, test_targets, reduction="s
        loss_test = loss_test + loss_test_batch.item()
        # classification error
        y_pred = model_best.classify(test_batch)
        e = 1.0 * (y_pred == test_targets)
        loss_error = loss_error + (1.0 - e).sum().item()
        # the number of examples
        N = N + test_batch.shape[0]
    # divide by the number of examples
    loss_test = loss_test / N
    loss_error = loss_error / N
    # Print the performance
    if epoch is None:
        print(f"-> FINAL PERFORMANCE: nll={loss_test}, ce={loss_error}")
    else:
        if epoch % 10 == 0:
            print(f"Epoch: {epoch}, val nll={loss_test}, val ce={loss_error}")

    return loss_test, loss_error


# An auxiliary function for plotting the performance curves
def plot_curve(
    name,
    signal,
    file_name="curve.pdf",
    xlabel="epochs",
    ylabel="nll",
    color="b-",
    test_eval=None,
):
    # plot the curve
    plt.plot(
        np.arange(len(signal)), signal, color, linewidth="3", label=ylabel + " val"
    )
    # if available, add the final (test) performance
    if test_eval is not None:
        plt.hlines(
            test_eval,
            xmin=0,
            xmax=len(signal),
            linestyles="dashed",
            label=ylabel + " test",
        )
        plt.text(
            len(signal),
            test_eval,
            "{:.3f}".format(test_eval),
        )
    # set x- and ylabels, add legend, save the figure
    plt.xlabel(xlabel), plt.ylabel(ylabel)
    plt.legend()
    plt.savefig(name + file_name, bbox_inches="tight")
    plt.show()
```

2.3 Training procedure

In [48]:
```
# PLEASE DO NOT REMOVE!
# The training procedure
def training(
    name, max_patience, num_epochs, model, optimizer, training_loader, val_loader
):
    nll_val = []
    error_val = []
    best_nll = 1000.0
    patience = 0

    # Main training loop
    for e in range(num_epochs):
        model.train()  # set the model to the training mode
        # load batches
        for indx_batch, (batch, targets) in enumerate(training_loader):
            # calculate the forward pass (loss function for given images and labels
            loss = model.forward(batch, targets)
            # remember we need to zero gradients! Just in case!
            optimizer.zero_grad()
            # calculate backward pass
            loss.backward(retain_graph=True)
            # run the optimizer
            optimizer.step()

        # Validation: Evaluate the model on the validation data
        loss_e, error_e = evaluation(val_loader, model_best=model, epoch=e)
        nll_val.append(loss_e)  # save for plotting
        error_val.append(error_e)  # save for plotting

        # Early-stopping: update the best performing model and break training if no
        # progress is observed.
        if e == 0:
            torch.save(model, name + ".model")
            best_nll = loss_e
        else:
            if loss_e < best_nll:
                torch.save(model, name + ".model")
                best_nll = loss_e
                patience = 0
            else:
                patience = patience + 1

        if patience > max_patience:
            break

    # Return nll and classification error.
    nll_val = np.asarray(nll_val)
    error_val = np.asarray(error_val)

    return nll_val, error_val
```

2.4 Experiments

Initialize dataloaders

In [49]:
```
# PLEASE DO NOT REMOVE
# Initialize training, validation and test sets.
train_data = Digits(mode="train")
val_data = Digits(mode="val")
test_data = Digits(mode="test")

# Initialize data loaders.
```

```
training_loader = DataLoader(train_data, batch_size=64, shuffle=True)
val_loader = DataLoader(val_data, batch_size=64, shuffle=False)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

In [56]:
```
print("How do we get our data from Digits class? \n")
print(f"Feature example: {train_data[1][0]}")
print(f"Feature example shape: {train_data[1][0].shape}")
print(f"Label example: {train_data[1][1]}")
```

```
How do we get our data from Digits class?

Feature example: [ 0.  0.  0. 12. 13.  5.  0.  0.  0.  0.  0. 11. 16.  9.  0.  0.
0.  0.
  3. 15. 16.  6.  0.  0.  0.  7. 15. 16. 16.  2.  0.  0.  0.  0.  1. 16.
 16.  3.  0.  0.  0.  0.  1. 16. 16.  6.  0.  0.  0.  0.  1. 16. 16.  6.
  0.  0.  0.  0.  0. 11. 16. 10.  0.  0.]
Feature example shape: (64,)
Label example: 1
```

In [57]:
```
print("How do we get our data from Pytorch DataLoader class? \n")
train_features, train_labels = next(iter(training_loader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")

print("\n\nWhat happens if we reshape a feature batch? \n")
reshape = Reshape(size=(1, 8, 8))
train_features_reshaped = reshape(train_features)
print(f"Feature batch shape after reshape: {train_features_reshaped.size()}")

print("\n\nWhat happens if we flatten a reshaped feature batch? \n")
flatten = Flatten()
train_features_flattened = flatten(train_features_reshaped)
print(f"Feature batch shape after flatten: {train_features_flattened.size()}")
```

```
How do we get our data from Pytorch DataLoader class?

Feature batch shape: torch.Size([64, 64])
Labels batch shape: torch.Size([64])


What happens if we reshape a feature batch?

Feature batch shape after reshape: torch.Size([64, 1, 8, 8])


What happens if we flatten a reshaped feature batch?

Feature batch shape after flatten: torch.Size([64, 64])
```

Initialize hyperparameters

In [ ]:
```
# PLEASE DO NOT REMOVE
# Hyperparameters
# -> data hyperparams
D = 64  # input dimension
```

```
# -> model hyperparams
M = 256  # the number of neurons in scale (s) and translation (t) nets
K = 10  # the number of labels
num_kernels = 32  # the number of kernels for CNN

# -> training hyperparams
lr = 1e-3  # learning rate
wd = 1e-5  # weight decay
num_epochs = 1000  # max. number of epochs
max_patience = 20  # an early stopping is used, if training doesn't improve for lon
```

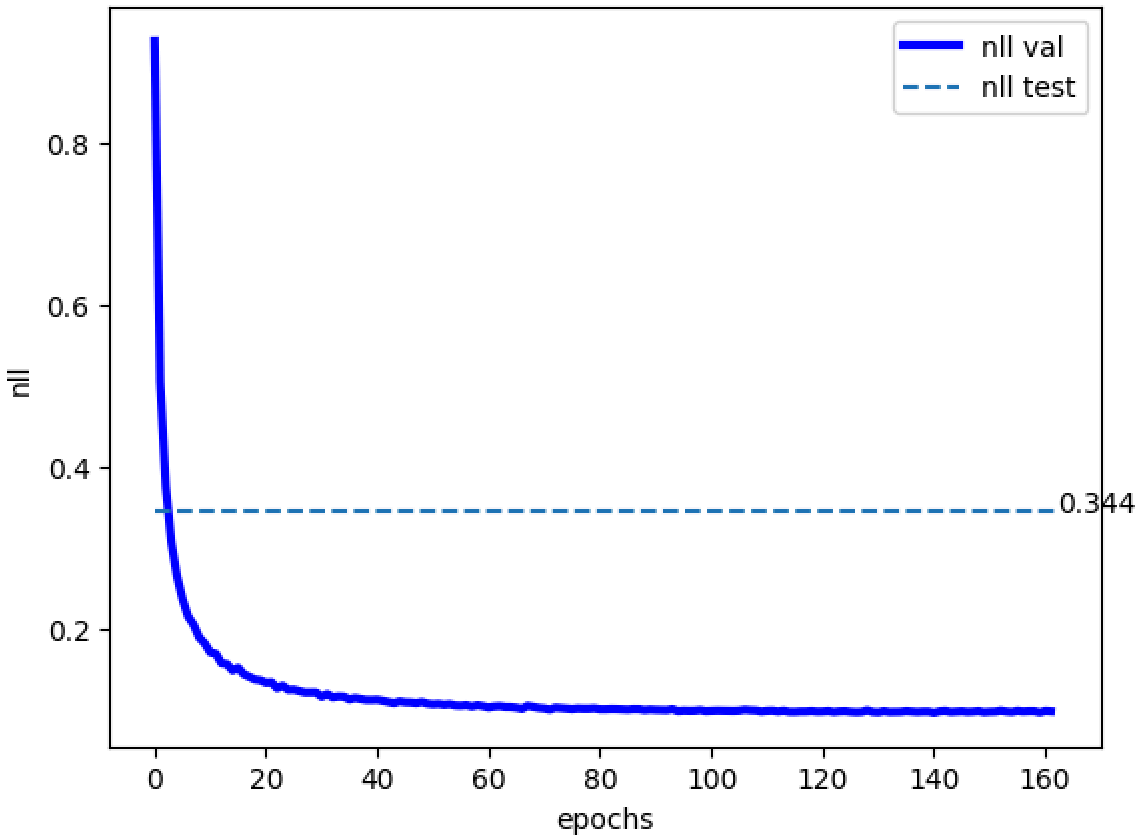
Running experiments

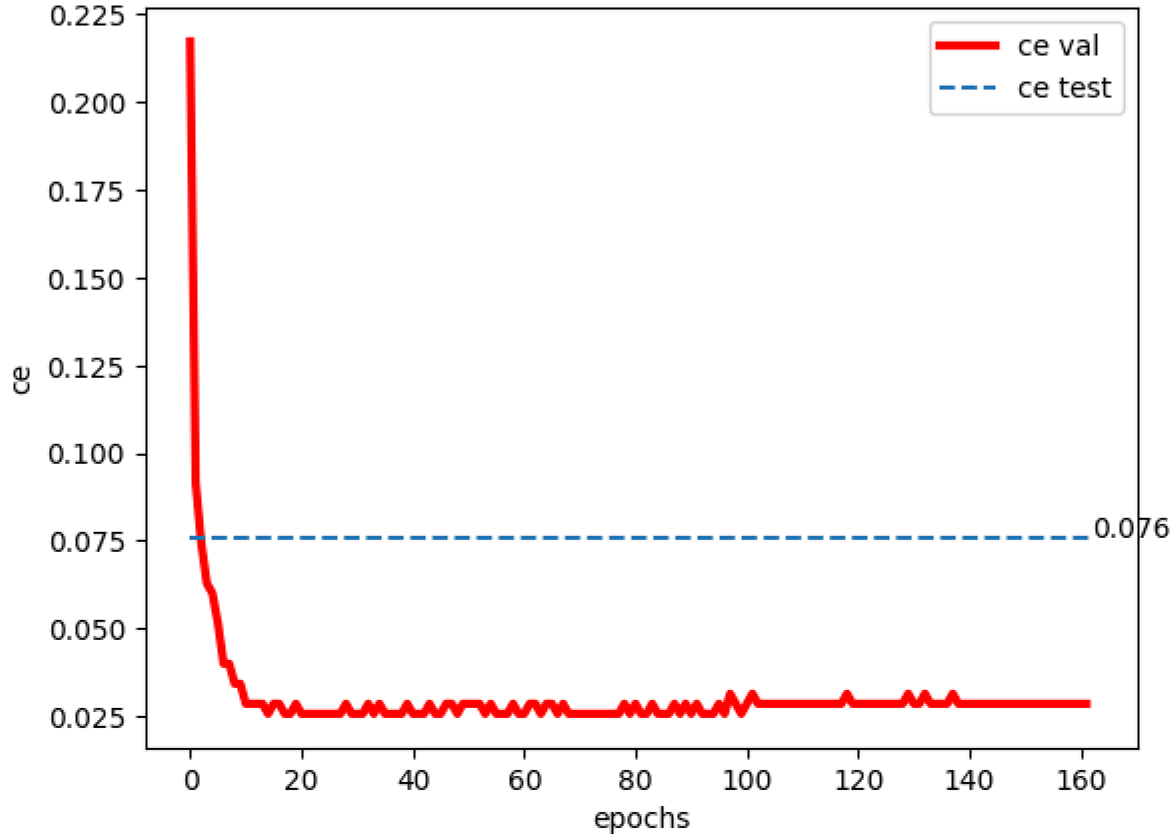
In the code below, you are supposed to implement architectures for MLP and CNN. For properly implementing these architectures, you can get 0.5pt for each of them.

In [60]:

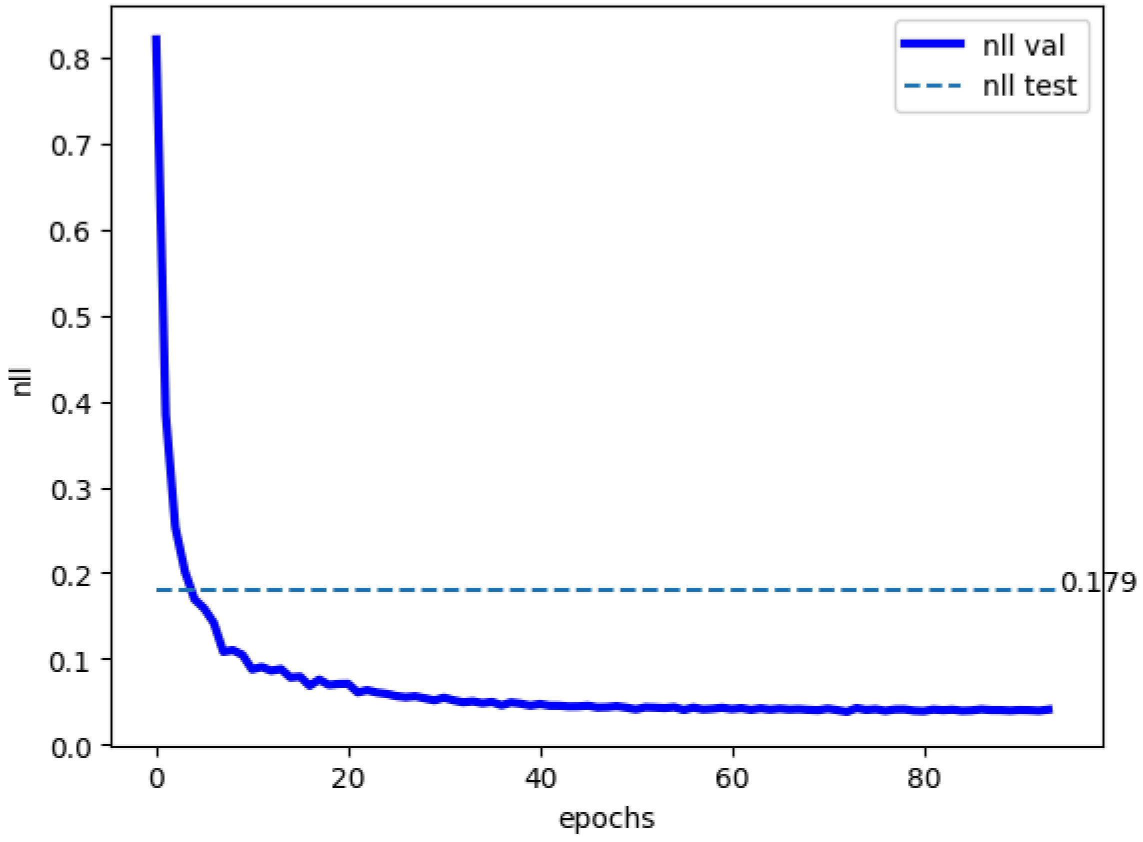
```
# PLEASE DO NOT REMOVE and FILL IN WHEN NECESSARY!
# We will run two models: MLP and CNN
names = ["classifier_mlp", "classifier_cnn"]

# loop over models
for name in names:
    print("\n-> START {}".format(name))
    # Create a folder (REMEMBER: You must mount your drive if you use Colab!)
    if name == "classifier_mlp":
        name = name + "_M_" + str(M)
    elif name == "classifier_cnn":
        name = name + "_M_" + str(M) + "_kernels_" + str(num_kernels)

    # Create a folder if necessary
    result_dir = os.path.join(results_dir, "results", name + "/")

    # =========
    # MAKE SURE THAT "result_dir" IS A PATH TO A LOCAL FOLDER OR A GOOGLE COLAB FOL
    result_dir = "./"  # (current folder)
    # =========
    if not (os.path.exists(result_dir)):
        os.mkdir(result_dir)

    # MLP
    if name[0:14] == "classifier_mlp":
        # =========
        # GRADING:
        # 0
        # 0.5pt if properly implemented
        # =========
        # ------
        # PLEASE FILL IN:
        # classnet = nn.Sequential(...)
        #
        # You are asked here to propose your own architecture
        # NOTE: Please remember that the output must be LogSoftmax!
        # ------
        classnet=nn.Sequential(
            #define architecture
            nn.Linear(D, M), #input layer
            nn.ReLU(), #activation function
            nn.Linear(M, K), #output layer
            nn.LogSoftmax(dim=1) #log softmax must be returned for output
        )
        pass

    # CNN
    elif name[0:14] == "classifier_cnn":
        # =========
        # GRADING:
        # 0
        # 0.5pt if properly implemented
        # =========
        # ------
        # PLEASE FILL IN:
        # classnet = nn.Sequential(...)
        #
        # You are asked here to propose your own architecture
        # NOTE: Plese note that the images are represented as vectors, thus, you mu
        # use Reshape(size) as the first layer, and Flatten() after all convolution
        # layers and before linear layers.
        # NOTE: Please remember that the output must be LogSoftmax!
        # ------
        classnet=nn.Sequential(
            #nn.Reshape(num_kernels),
            #input_data = input_data.unsqueeze(0),
            Reshape((1,8,8)), #numbers picked based on 1d dimensions
            nn.Conv2d(1, num_kernels, kernel_size=3, stride=1, padding=1), #apply 2
            nn.ReLU(), # apply rectified linear unit, activation function
            nn.MaxPool2d(kernel_size=2, stride=2), # apply 2d max pooling, detects
            #pool size is determined by channels, kernel size, stride, and padding

            Flatten(), #reshapes image into one dimension
            nn.Linear(4 * 4 * num_kernels, M), #applies linear transformation to da
            #linear transformation is based on a 4x4 matrix and multiplied by the n


            nn.ReLU(), #apply activation function (again)
            nn.Linear(M, K), #apply linear transformation again
            nn.LogSoftmax(dim=1) #apply logarithmic function to 1 dimensioned tenso
        )
        pass

    # Init ClassifierNN
    model = ClassifierNeuralNet(classnet)

    # Init OPTIMIZER (here we use ADAMAX)
    optimizer = torch.optim.Adamax(
        [p for p in model.parameters() if p.requires_grad == True],
        lr=lr,
        weight_decay=wd,
    )

    # Training procedure
    nll_val, error_val = training(
        name=result_dir + name,
        max_patience=max_patience,
        num_epochs=num_epochs,
        model=model,
        optimizer=optimizer,
        training_loader=training_loader,
        val_loader=val_loader,
    )

    # The final evaluation (on the test set)
    test_loss, test_error = evaluation(name=result_dir + name, test_loader=test_loa
    # write the results to a file
    f = open(result_dir + name + "_test_loss.txt", "w")
    f.write("NLL: " + str(test_loss) + "\nCE: " + str(test_error))
    f.close()
    # create curves
    plot_curve(
        result_dir + name,
        nll_val,
        file_name="_nll_val_curve.pdf",
        ylabel="nll",
        test_eval=test_loss,
    )
    plot_curve(
        result_dir + name,
        error_val,
        file_name="_ca_val_curve.pdf",
        ylabel="ce",
        color="r-",
        test_eval=test_error,
    )
```

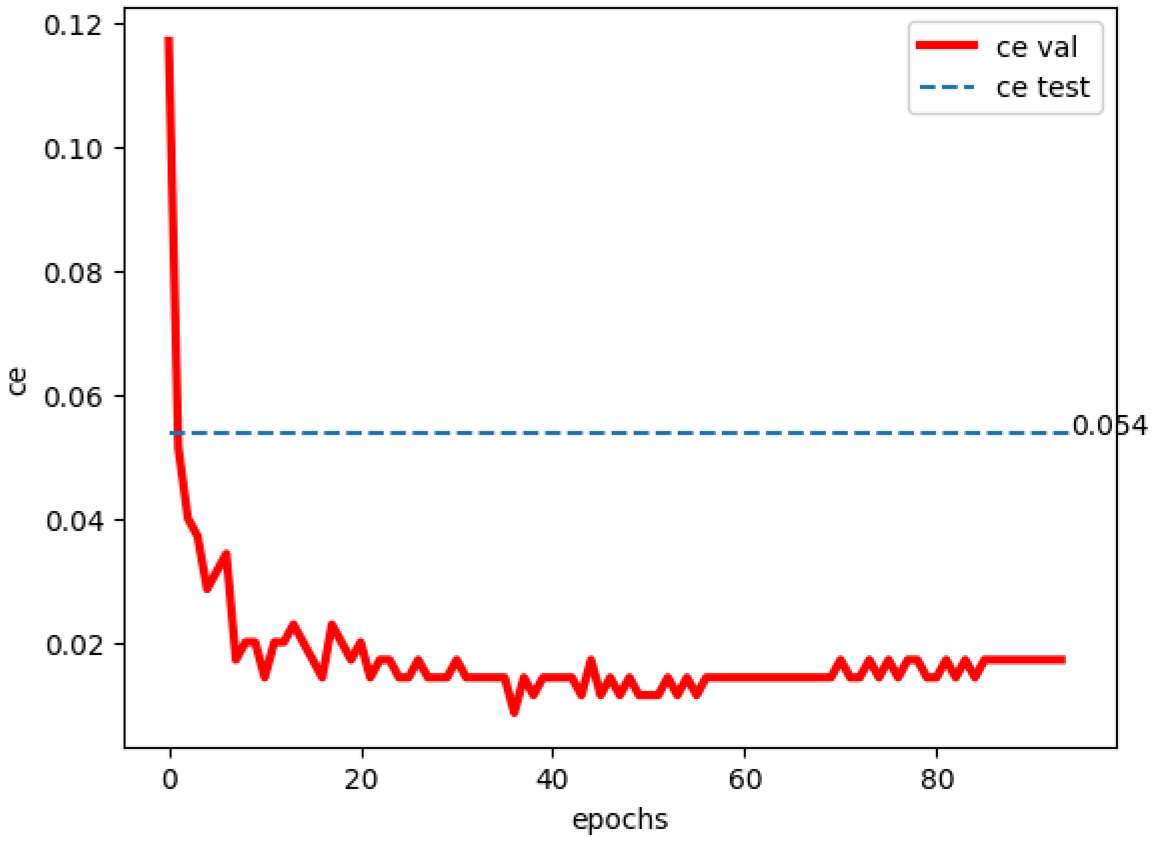
```
-> START classifier_mlp
Epoch: 0, val nll=0.926091570172991, val ce=0.21714285714285714
Epoch: 10, val nll=0.1700299324308123, val ce=0.02857142857142857
Epoch: 20, val nll=0.13246192829949516, val ce=0.025714285714285714
Epoch: 30, val nll=0.1151305399622236, val ce=0.025714285714285714
Epoch: 40, val nll=0.1114744736467089, val ce=0.025714285714285714
Epoch: 50, val nll=0.10580573269299098, val ce=0.02857142857142857
Epoch: 60, val nll=0.1019064325945718, val ce=0.025714285714285714
Epoch: 70, val nll=0.10071593386786325, val ce=0.025714285714285714
Epoch: 80, val nll=0.09966671517917088, val ce=0.02857142857142857
Epoch: 90, val nll=0.09894844131810325, val ce=0.025714285714285714
Epoch: 100, val nll=0.09762134603091649, val ce=0.02857142857142857
Epoch: 110, val nll=0.09771673151424953, val ce=0.02857142857142857
Epoch: 120, val nll=0.09692713043519428, val ce=0.02857142857142857
Epoch: 130, val nll=0.0958273520214217, val ce=0.02857142857142857
Epoch: 140, val nll=0.09523875440870012, val ce=0.02857142857142857
Epoch: 150, val nll=0.09656545843396867, val ce=0.02857142857142857
Epoch: 160, val nll=0.09770195756639753, val ce=0.02857142857142857
-> FINAL PERFORMANCE: nll=0.34360359072418556, ce=0.07606263982102908
```


nll val
nll test
0.344
nll
epochs
0.8
0.6
0.4
0.2
0
20
40
60
80
100
120
140
160


ce val
ce test
0.076
ce
epochs
0.225
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
20
40
60
80
100
120
140
160

```
-> START classifier_cnn
Epoch: 0, val nll=0.8204865591866629, val ce=0.11714285714285715
Epoch: 10, val nll=0.08764842680522374, val ce=0.014285714285714285
Epoch: 20, val nll=0.07105536358697073, val ce=0.02
Epoch: 30, val nll=0.054891128369740076, val ce=0.017142857142857144
Epoch: 40, val nll=0.04747198692389897, val ce=0.014285714285714285
Epoch: 50, val nll=0.041181928558009014, val ce=0.011428571428571429
Epoch: 60, val nll=0.04144142316920417, val ce=0.014285714285714285
Epoch: 70, val nll=0.04208529421261379, val ce=0.017142857142857144
Epoch: 80, val nll=0.03914202353784016, val ce=0.014285714285714285
Epoch: 90, val nll=0.040440373952899664, val ce=0.017142857142857144
-> FINAL PERFORMANCE: nll=0.1794632345237988, ce=0.053691275167785234
```





2.5 Analysis

Question 3 (0-0.5pt): Please compare the convergence of MLP and CNN in terms of the loss function and the classification error.

Answer: PLEASE FILL IN and PASTE THE IMAGES HERE

The MLP and CNN models both converge eventually. CNN convergence occurs faster than MLP, as it reaches lower negative log-likelihood and classification error values at epoch 100 as compared to CNN. Refer to the above images.

MLP has a final performance of nll=0.3436 and ce=0.0760. CNN has a final performance of nll=0.1795 and ce=0.0537. This proves that CNN is significantly better in this case. However, this might not always be true as there are outside factors to consider such as specific resources and requirements.

Question 4 (0-0.5pt): In general, for a properly picked architectures, a CNN should work better than an MLP. Did you notice that? Why (in general) CNNs are better suited to images than MLPs?

Answer: PLEASE FILL IN and PASTE THE IMAGES HERE

In the instances above it is clear that CNN performs better than an MLP. This can be attributed to the fact that MLPs use vectors while CNN's use tensors. This makes it significantly easier for the CNN to understand the spatial relationships of the pixels within an image.

3 Application to Street House View Numbers (SVHN) (6pt)

Please repeat (some) of the code in the previous section and apply a bigger convolutional neural network (CNN) to the following dataset:

http://ufldl.stanford.edu/housenumbers/

Please follow the following steps:

1. (1pt) Create appropriate Dataset class. Please remember to use the original training data and test data, and also to create a validation set from the traning data (at least 10% of the training examples). **Do not use extra examples!**
2. (1pt) Implement an architecture that will give at most 0.1 classification error. For instance, see this paper as a reference: https://arxiv.org/pdf/1204.3968.pdf#:~:text=The%20SVHN%20classification%20dataset%20
3. (1pt) Think of an extra component that could improve the performance (e.g., a regularization, specific activation functions).
4. (1pt) Provide a good explanation of the applied architecture and a description of all components.
5. (2pt) Analyze the results.

Please be very precise, comment your code and provide a comprehensive and clear analysis.

In [90]:
```
#This cell downloads/loads the training and testing data from the SHVN dataset!!

import torchvision
import torchvision.transforms as transforms

transform=torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(), #converts to tensors

    torchvision.transforms.Resize((32,32)), #size of images from dataset

])

train_download = torchvision.datasets.SVHN(root='~/Desktop/CI_Assignment_4', split
#train_upload = torch.utils.data.DataLoader(train_data) #loads training data

test_download = torchvision.datasets.SVHN(root='~/Desktop/CI_Assignment_4', split =
test_upload = torch.utils.data.DataLoader(test_data, batch_size=32) # loads testing
```

```
#train_upload = DataLoader(train_data, batch_size=32, shuffle=True)
#val_loader = DataLoader(val_data, batch_size=32, shuffle=False)
#test_upload = DataLoader(test_data, batch_size=32, shuffle=False)

train_download,val_data=torch.utils.data.random_split(train_download, [65197, 8060]
train_upload = torch.utils.data.DataLoader(train_download, batch_size=32) #uploads
val_load = torch.utils.data.DataLoader(val_data, batch_size=32) #loads values using
```

```
Using downloaded and verified file: C:\Users\hghol/Desktop/CI_Assignment_4\train_32x
32.mat
Using downloaded and verified file: C:\Users\hghol/Desktop/CI_Assignment_4\test_32x3
2.mat
```

In [99]:
```
# PLEASE DO NOT REMOVE
# Hyperparameters
# -> data hyperparams
D = 32  # input dimension 64

# -> model hyperparams
M = 256  # the number of neurons in scale (s) and translation (t) nets 256
K = 10  # the number of labels
num_kernels = 32  # the number of kernels for CNN 32

# -> training hyperparams
lr = 1e-3  # learning rate
wd = 1e-5  # weight decay
num_epochs = 1000  # max. number of epochs
max_patience = 20  # an early stopping is used, if training doesn't improve for lon
```

In [100…
```
# PLEASE DO NOT REMOVE and FILL IN WHEN NECESSARY!
# We will run two models: MLP and CNN
names = ["classifier_cnn"]

# loop over models
for name in names:
    print("\n-> START {}".format(name))
    # Create a folder (REMEMBER: You must mount your drive if you use Colab!)
    if name == "classifier_mlp":
        name = name + "_M_" + str(M)
    elif name == "classifier_cnn":
        name = name + "_M_" + str(M) + "_kernels_" + str(num_kernels)

    # Create a folder if necessary
    result_dir = os.path.join(results_dir, "results", name + "/")

    # =========
    # MAKE SURE THAT "result_dir" IS A PATH TO A LOCAL FOLDER OR A GOOGLE COLAB FOL
    result_dir = "./"  # (current folder)
    # =========
    if not (os.path.exists(result_dir)):
        os.mkdir(result_dir)



    # CNN
    elif name[0:14] == "classifier_cnn":
        # =========
        # GRADING:
        # 0
        # 0.5pt if properly implemented
        # =========
        # ------
        # PLEASE FILL IN:
        # classnet = nn.Sequential(...)
        #
        # You are asked here to propose your own architecture
        # NOTE: Plese note that the images are represented as vectors, thus, you mu
        # use Reshape(size) as the first layer, and Flatten() after all convolution
        # layers and before linear layers.
        # NOTE: Please remember that the output must be LogSoftmax!
        # ------
        classnet=nn.Sequential(
            #nn.Reshape(num_kernels),
            #input_data = input_data.unsqueeze(0),

            nn.Conv2d(3, num_kernels, kernel_size=3, padding=1), #apply 2d convolut
            nn.ReLU(), # apply rectified linear unit, activation function
            nn.MaxPool2d(kernel_size=2, stride=2), # apply 2d max pooling, detects

            nn.Conv2d(num_kernels, num_kernels, kernel_size=3, padding=1), #apply 2
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # apply 2d max pooling, detects

            #nn.Conv2d(num_kernels, num_kernels, kernel_size=3, padding=1), #apply
            #nn.ReLU(),
            #nn.MaxPool2d(kernel_size=2, stride=2), # apply 2d max pooling, detects

            Flatten(), #reshapes image into one dimension
            nn.Linear(8 * 8 * num_kernels, M), #applies linear transformation to ,
            nn.ReLU(), #apply activation function (again)
            nn.Linear(M, K), #apply linear transformation again
            nn.LogSoftmax(dim=1) #apply logarithmic function to 1 dimensioned tenso
        )
        pass

    # Init ClassifierNN
    model = ClassifierNeuralNet(classnet)

    # Init OPTIMIZER (here we use ADAMAX)
    optimizer = torch.optim.Adamax(
        [p for p in model.parameters() if p.requires_grad == True],
        lr=lr,
        weight_decay=wd,
    )

    # Training procedure
    nll_val, error_val = training(
        name=result_dir + name,
        max_patience=max_patience,
        num_epochs=num_epochs,
        model=model,
        optimizer=optimizer,
        training_loader=train_upload,
        val_loader=val_load,
    )

    # The final evaluation (on the test set)
    test_loss, test_error = evaluation(name=result_dir + name, test_loader=test_upl
    # write the results to a file
    f = open(result_dir + name + "_test_loss.txt", "w")
    f.write("NLL: " + str(test_loss) + "\nCE: " + str(test_error))
    f.close()
    # create curves
    plot_curve(
        result_dir + name,
        nll_val,
        file_name="_nll_val_curve.pdf",
        ylabel="nll",
        test_eval=test_loss,
    )
    plot_curve(
        result_dir + name,
        error_val,
        file_name="_ca_val_curve.pdf",
        ylabel="ce",
        color="r-",
        test_eval=test_error,
    )
```

```
-> START classifier_cnn
Epoch: 0, val nll=0.865976527547718, val ce=0.2532258064516129
Epoch: 10, val nll=0.43195636381878155, val ce=0.11823821339950372
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[100], line 78
     71 optimizer = torch.optim.Adamax(
     72     [p for p in model.parameters() if p.requires_grad == True],
     73     lr=lr,
     74     weight_decay=wd,
     75 )
     77 # Training procedure
---> 78 nll_val, error_val = training(
     79     name=result_dir + name,
     80     max_patience=max_patience,
     81     num_epochs=num_epochs,
     82     model=model,
     83     optimizer=optimizer,
     84     training_loader=train_upload,
     85     val_loader=val_load,
     86 )
     88 # The final evaluation (on the test set)
     89 test_loss, test_error = evaluation(name=result_dir + name, test_loader=test_
upload)

Cell In[48], line 17, in training(name, max_patience, num_epochs, model, optimizer,
training_loader, val_loader)
     14 # load batches
     15 for indx_batch, (batch, targets) in enumerate(training_loader):
     16     # calculate the forward pass (loss function for given images and labels)
---> 17     loss = model.forward(batch, targets)
     18     # remember we need to zero gradients! Just in case!
     19     optimizer.zero_grad()

Cell In[46], line 36, in ClassifierNeuralNet.forward(self, x, y, reduction)
     31 def forward(self, x, y, reduction="avg"):
     32     # ------
     33     # PLEASE FILL IN
     34     # loss = ...
     35     # ------
---> 36     logits = self.classnet(x) # defines logits variable from classnet(x)
     37     probabilities = F.log_softmax(logits, dim=1) # calculates probabilities
using log-softmax function
     38     loss = self.nll(probabilities, y.long()) # uses negative log-likelihood
loss function from probabilities

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\nn\modules\mo
dule.py:1532, in Module._wrapped_call_impl(self, *args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\nn\modules\mo
dule.py:1541, in Module._call_impl(self, *args, **kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the logic in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or self._forward_ho
oks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or _global_backward_hooks
   1540             or _global_forward_hooks or _global_forward_pre_hooks):
-> 1541         return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\nn\modules\co
ntainer.py:217, in Sequential.forward(self, input)
    215 def forward(self, input):
    216     for module in self:
--> 217         input = module(input)
    218     return input

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\nn\modules\mo
dule.py:1532, in Module._wrapped_call_impl(self, *args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\nn\modules\mo
dule.py:1541, in Module._call_impl(self, *args, **kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the logic in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or self._forward_ho
oks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or _global_backward_hooks
   1540         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\nn\modules\co
nv.py:460, in Conv2d.forward(self, input)
    459 def forward(self, input: Tensor) -> Tensor:
--> 460     return self._conv_forward(input, self.weight, self.bias)

File ~\AppData\Local\Programs\Python\Python311\Lib\site-packages\torch\nn\modules\co
nv.py:456, in Conv2d._conv_forward(self, input, weight, bias)
    452 if self.padding_mode != 'zeros':
    453     return F.conv2d(F.pad(input, self._reversed_padding_repeated_twice, mode
=self.padding_mode),
    454                     weight, bias, self.stride,
    455                     _pair(0), self.dilation, self.groups)
--> 456 return F.conv2d(input, weight, bias, self.stride,
    457                 self.padding, self.dilation, self.groups)

KeyboardInterrupt:
```

Explanation of architecture, description of components

This code uses a Convolutional Neural Network (CNN). The architecture utilized here is a CNN, as that is particularly effective for image classification tasks since it can automatically learn hierarchical representations of features directly from pixel values.

Convolutional Layers: The CNN starts with two convolutional layers, each followed by a batch normalization layer (nn.MaxPool2D) and a rectified linear unit (ReLU) activation function. These convolutional layers (nn.Conv2d) perform feature extraction by applying learnable filters to the input image. The ReLU activation function introduces non-linearity, allowing the network to learn complex patterns in the data. Batch normalization helps stabilize and accelerate training by normalizing the activations of each convolutional layer.

Max Pooling Layers: After each convolutional layer, there is a max-pooling layer (nn.MaxPool2d), which reduces the spatial dimensions of the feature maps while retaining the most important information. Max-pooling helps to make the representations smaller and more manageable, reducing the number of parameters and computation in the network.

Flatten Layer: After the convolutional layers, the feature maps are flattened into a vector using the Flatten operation. This prepares the data for input into the fully connected layers. Fully Connected Layers (Linear): The flattened feature vector is then passed through two fully connected (linear) layers. The first fully connected layer reduces the dimensionality of the data and learns high-level features. The second fully connected layer produces the final output logits, which are passed to the softmax function for probability estimation.

Output Layer (LogSoftmax): The final layer applies the LogSoftmax activation function (nn.LogSoftmax) to the output logits. LogSoftmax normalizes the logits into probabilities, making it suitable for multi-class classification tasks. It computes the logarithm of the softmax function, which helps to stabilize the computation and improve numerical stability during training. Summary: In summary, the architecture consists of convolutional layers for feature extraction, max-pooling layers for spatial reduction, fully connected layers for high-level feature learning, and an output layer for probability estimation. Batch normalization is applied throughout the network to stabilize training, and ReLU activation functions introduce non-linearity. Overall, this architecture is designed to effectively capture and learn hierarchical representations of features from input images, enabling accurate classification.

Analysis

The current version of CNN algorithm is running extremely slowly. Based off of the current results, it is definitely trending in the right direction. NLL and CE were both cut in more than half between epochs 0 and 10. However, I believe that if given enough time to run, it would reach a classification error below 0.1. Based on my other trials, I have been able to get the error under 0.1, then it raises slightly and ends around 0.12. Super frustrating. In this iteration, I increased the number of epochs and added a second convolution layer. Obviously, this accounts for the dramatic decrease in runtime. NLLLoss does actually converge in every version I have tested. This can help us know that the model is effectively learning the house numbers.

With more time, I would want to further adjust the hyperparameters as well as add another convolution layer to further analyze the results. I think this would decrease my classification error value greatly. The most challenging thing about this assignment was definitely understanding the dimensions at every step in the process. However, I do feel like I really understand how the channels, stride, padding, and kernels all fit together to create an image, and I think it is super interesting how that information is used in neural networks!

Extra Component for Improvement

I would implement L2 regularization (weight decay) to further improve this code. This penalizes large weights by adding a term to the loss function proportional to the square of the weights. This would help to prevent overfitting and generally improve the entire model's performance.