assignment_4

May 16, 2024

1 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**.

1.1 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.

[322]:
```
import os
#!pip install torch
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
```

[323]:
```
# 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")
```

[324]:
```
# 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/"
```

[325]:
```
# 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!
#Loads the data and prepares it, like splitting into training, val and test so␣
 ↪to be used later on
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)
```

[326]:
```
# PLEASE DO NOT REMOVE
# Here, we plot some images (8px x 8px).
#The code snippet you mentioned is primarily for visualizing the first 16␣
 ↪images from the Digits dataset. This is an essential part of data␣
 ↪exploration, allowing you to understand what the data looks like before␣
 ↪moving into more
# complex procedures like training a model.
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")
```

1.2 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.

1.2.1 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.

[327]:
```
# 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.

[328]:
```
# =========
# 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):
        # ------
        # Compute the logits by passing input x through classnet
        logits = self.classnet(x)
        # Convert logits to class probabilities using log_softmax
        log_probs = F.log_softmax(logits, dim=1)
        # Find the class with the maximum probability
        # this is useful and efficient for later training phrase
        y_pred = log_probs.argmax(dim=1)

        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"):
        # ------
        # Pass input x through the classnet to get the logits
        logits = self.classnet(x)
        # Calculate log-softmax of logits for NLLLoss
        log_probs = F.log_softmax(logits, dim=1)

        # Calculate the loss using NLLLoss
        loss = self.nll(log_probs, y)
        # ------
        if reduction == "sum":
            return loss.sum()
        else:
            return loss.mean()
```

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 the given classification problem in the code, Negative Log-Likelihood Loss (NLLLoss) used as an objective function, which can be seen from this line of the forward function:

```
loss = self.nll(log_probs, y)
```

But the form of the obj function chnages due to this part of the code:

```
if reduction == "sum":
    return loss.sum()
else:
    return loss.mean()
```

If reduction part is set to 'mean' then the mathematical form of the Negative Log-Likelihood Loss as an objective function is:

$$L = -\frac{1}{N}\sum_{i=1}^{N} \log p(y_i \mid x_i, \theta)$$

And if its set to 'sum';

$$L = -\sum_{i=1}^{N} \log p(y_i \mid x_i, \theta)$$

Hereby:

- N is the number of samples in the batch.
- y_i is the true class labels
- x_i is the input features
- represents the model parameters

The term logp(yi xi ,) is the log probability of the true label y_i given the input x_i. The input x_i used to make logits that transformed into logarithm of probabilities bylog_softmax for each class. So, the transformed log probabilities are used to calculate loss by NLL.
The function nn.NLLLoss in PyTorch is specifically designed to work with these log probabilities. It calculates the negative log of the log probabilities according to the actual labels and the calculated values over all samples in the batch. The main idea of the function is to minimize the average negative log probability of the trues class across all samples, if reduction is set to 'mean'.

Additionally, with the help of the reduction="none" from self.nll = nn.NLLLoss(reduction="none") parameter of nn.NLLLoss the loss values are handled without any summation or averaging. This means the loss is computed individually for each sample in the batch. Later on, in the 'forward' function the nature of the obj function changes whether reduction is set to 'mean' or 'sum'. Thereby, loss would be determined by the final fomr of the function.

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:

Yes, it is correct to use log-softmax instead of the softmax for making predictions.

NLLLoss needs log probabilities for to measure the performance of a classification model by directly computing the negative log of the probabilities assigned to the true class labels. Thereby, log-sofmax provides those log probabilities for NLLLoss.

By softmax the logarithm function applied after conversion of probabilities. But Log-Softmax applies directly the natural logarithm to the softmax output. This case, log-softmax is more efficient and ensures a stable training of the network. Usage of log-softmax does not affect the accuracy of the predictions. Either with softmax or log-softmax the predictions would be the same. However, if we look at the computational efficiency, using log-softmax would be more numerically stable than using softmax followed by a log operation. And this approach would perform better on large-scale models and dataset.

1.2.2 2.2 Evaluation

[329]:
```
# 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="sum")
        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()
```

1.2.3 2.3 Training procedure

[330]:
```
# 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
```

1.2.4 2.4 Experiments

Initialize dataloaders

[331]:
```
# 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)
```

[332]:
```
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
```

[333]:
```
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

[334]:
```
# 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␣
 ↪longer than 20 epochs, it is stopped
```

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.

[335]:
```
# 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
    # Set the base directory to your specific folder
    base_dir = "/Users/selma/Desktop/computational int/results_dir"

    # Append the specific experiment name to the base directory to organize␣
 ↪results
    result_dir = os.path.join(base_dir, name + "/")

    # Check if the directory exists, and if not, create it


    # =========
```

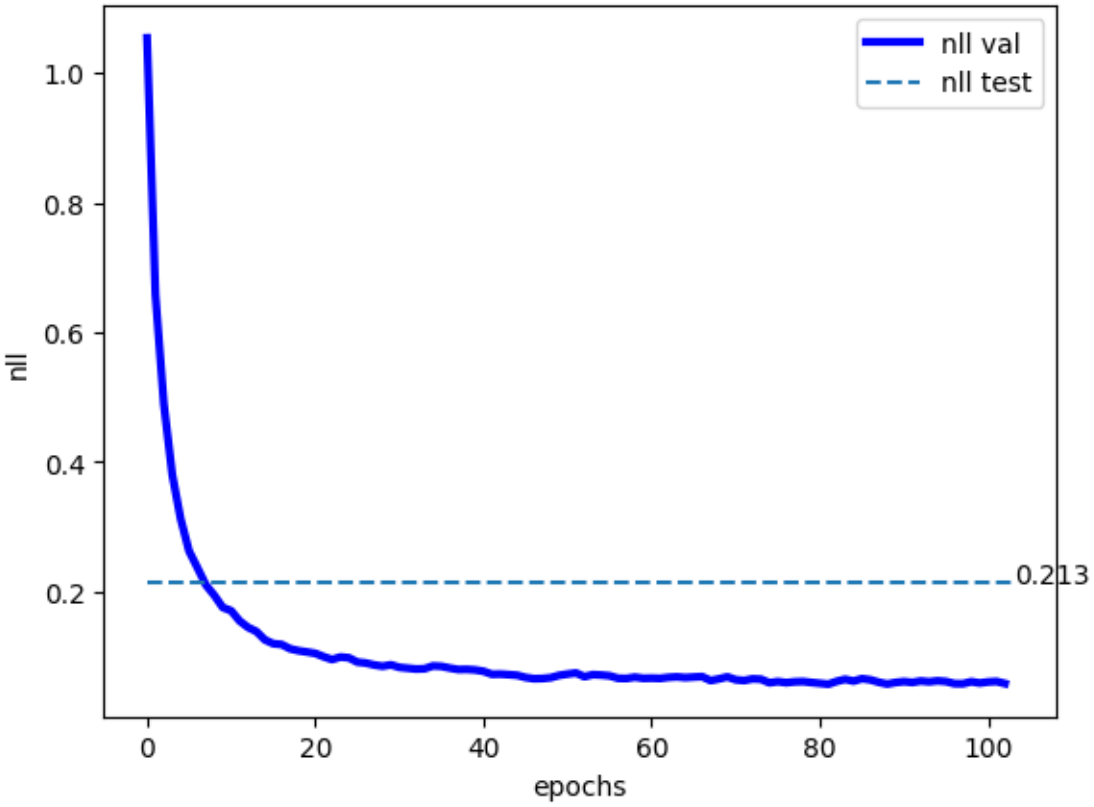
```
    # MAKE SURE THAT "result_dir" IS A PATH TO A LOCAL FOLDER OR A GOOGLE COLAB␣
↪FOLDER (DEFINED IN CELL 3)
    #result_dir = "./"  # (current folder)
    # =========
    if not (os.path.exists(result_dir)):
        os.mkdir(result_dir)

    # MLP
    if name[0:14] == "classifier_mlp":
        # =========
        # GRADING:
        # 0
        # 0.5pt if properly implemented
        # =========
        # ------
        # PLEASE FILL IN:
        # classnet = nn.Sequential(...)


        # D -> M -> M -> K
        #mlp
        classnet = nn.Sequential(
            nn.Linear(D, M),  #non-linearity
            nn.LeakyReLU(0.01),  # small negative slope coefficient, prevent␣
↪dead neurons
            nn.Dropout(0.5),  # Adding dropout to reduce overfitting
            nn.Linear(M, M),
            nn.LeakyReLU(0.01),
            nn.BatchNorm1d(M),  # Batch normalization for better convergence
            nn.Linear(M, K),
            nn.LogSoftmax(dim=1) #output
        )

        # You are asked here to propose your own architecture
        # NOTE: Please remember that the output must be LogSoftmax!
        # ------
        pass

    # CNN
    elif name[0:14] == "classifier_cnn":
        # =========
        # GRADING:
        # 0
        # 0.5pt if properly implemented
        # =========
        # ------
        classnet = nn.Sequential(
```

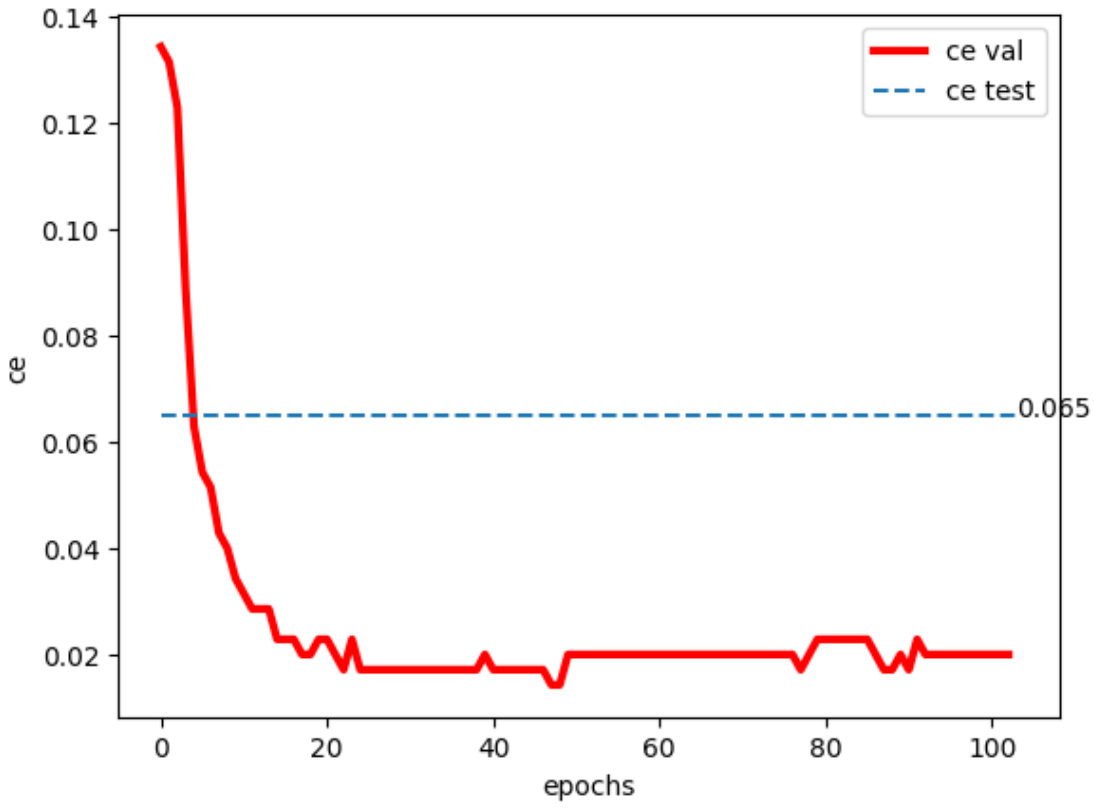
```
            nn.Unflatten(1, (1, 8, 8)), #make input in 2D

            #feature maps & extraction
            nn.Conv2d(1, num_kernels, kernel_size=3, padding=1), #feature␣
↪extraction
            nn.ReLU(), #adding non-linearity
            nn.MaxPool2d(2), #reduces the spatial dimensions

            nn.Conv2d(num_kernels, num_kernels * 2, kernel_size=3, padding=1),␣
↪#second convolution layer
            nn.BatchNorm2d(num_kernels * 2),  #batch normalization
            nn.ReLU(),
            nn.MaxPool2d(2),

            Flatten(), # !! flattens the multi-dimensional feature maps into a␣
↪single vector.

            # fully connected layer & classification
            nn.Linear(num_kernels * 2 * 2 * 2, M),
            nn.Dropout(0.5),  #dropout to reduce overfitting
            nn.ReLU(),
            nn.Linear(M, K),
            nn.LogSoftmax(dim=1) #output
        )


        # You are asked here to propose your own architecture
        # NOTE: Plese note that the images are represented as vectors, thus,␣
↪you must
        # use Reshape(size) as the first layer, and Flatten() after all␣
↪convolutional
        # layers and before linear layers.
        # NOTE: Please remember that the output must be LogSoftmax!
        # ------
        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_loader)
    # 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,
    )
```

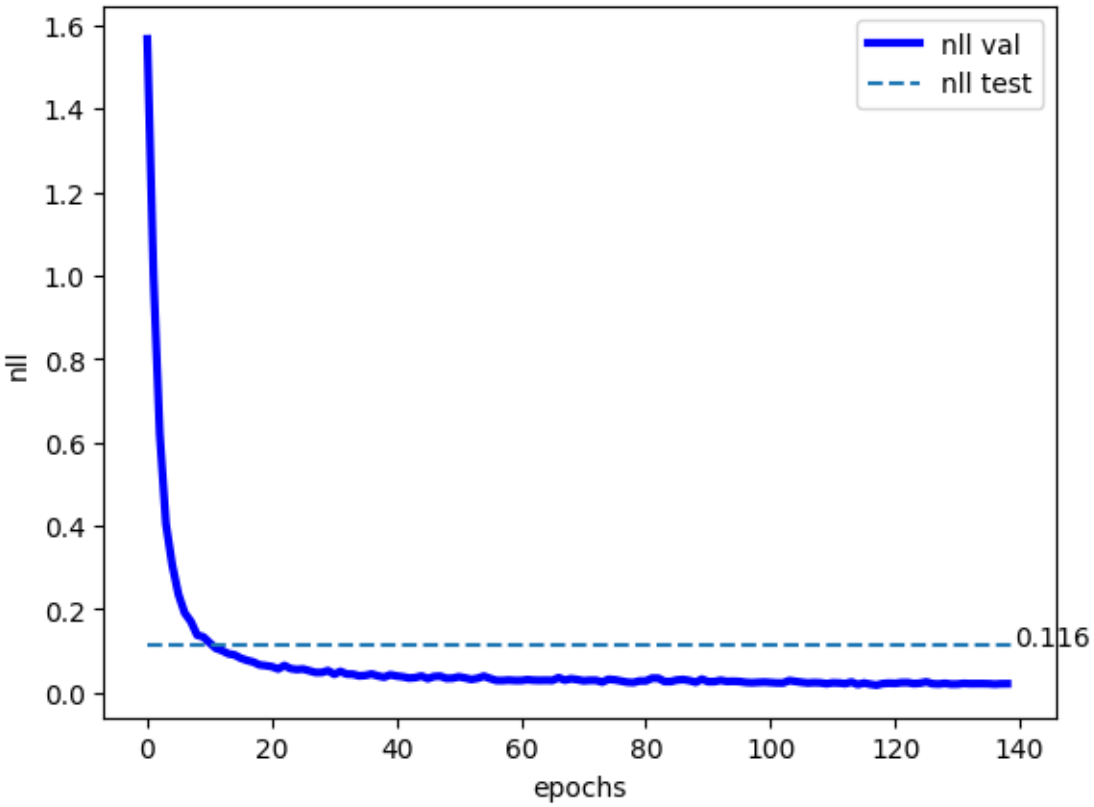
```
-> START classifier_mlp
Epoch: 0, val nll=1.0539277430943081, val ce=0.13428571428571429
Epoch: 10, val nll=0.17072769846235003, val ce=0.03142857142857143
Epoch: 20, val nll=0.10497098582131523, val ce=0.022857142857142857
Epoch: 30, val nll=0.08339515992573329, val ce=0.017142857142857144
Epoch: 40, val nll=0.07737104143415179, val ce=0.017142857142857144
Epoch: 50, val nll=0.07307283571788242, val ce=0.02
Epoch: 60, val nll=0.06680194990975516, val ce=0.02
Epoch: 70, val nll=0.06425027694020953, val ce=0.02
Epoch: 80, val nll=0.05878981837204524, val ce=0.022857142857142857
Epoch: 90, val nll=0.06146417643342699, val ce=0.017142857142857144
Epoch: 100, val nll=0.060923955270222256, val ce=0.02
```

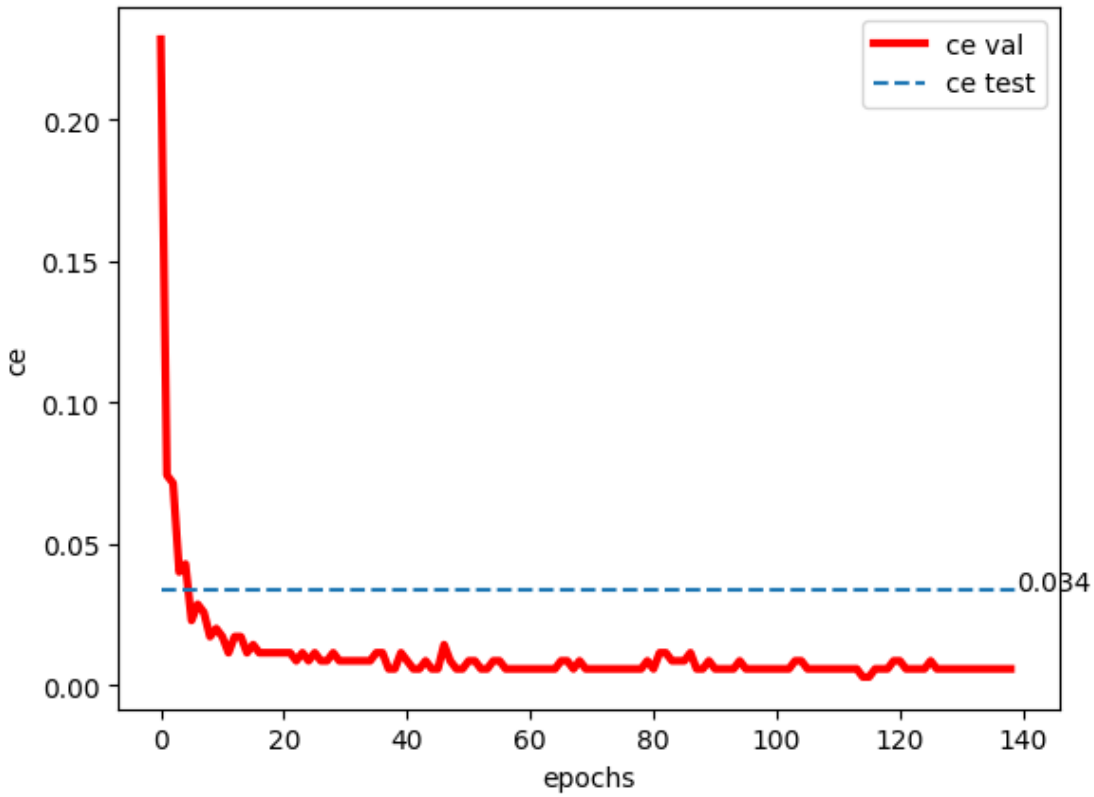
-> FINAL PERFORMANCE: nll=0.21294166684417384, ce=0.06487695749440715






```
-> START classifier_cnn
Epoch: 0, val nll=1.5668578774588449, val ce=0.22857142857142856
Epoch: 10, val nll=0.11948375156947545, val ce=0.017142857142857144
Epoch: 20, val nll=0.06232196467263358, val ce=0.011428571428571429
Epoch: 30, val nll=0.0440256861277989, val ce=0.008571428571428572
Epoch: 40, val nll=0.039859893662588936, val ce=0.008571428571428572
Epoch: 50, val nll=0.038165772472109116, val ce=0.008571428571428572
Epoch: 60, val nll=0.029203495042664663, val ce=0.005714285714285714
Epoch: 70, val nll=0.02845051084245954, val ce=0.005714285714285714
Epoch: 80, val nll=0.02807743285383497, val ce=0.005714285714285714
Epoch: 90, val nll=0.026337989824158806, val ce=0.005714285714285714
Epoch: 100, val nll=0.024075352251529694, val ce=0.005714285714285714
Epoch: 110, val nll=0.023602229739938464, val ce=0.005714285714285714
Epoch: 120, val nll=0.022171095567090172, val ce=0.008571428571428572
Epoch: 130, val nll=0.020524099682058608, val ce=0.005714285714285714
-> FINAL PERFORMANCE: nll=0.11565651109554624, ce=0.03355704697986577
```


nll val
nll test
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
nll
0.116
0
20
40
60
80
100
120
140
epochs



1.3 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:

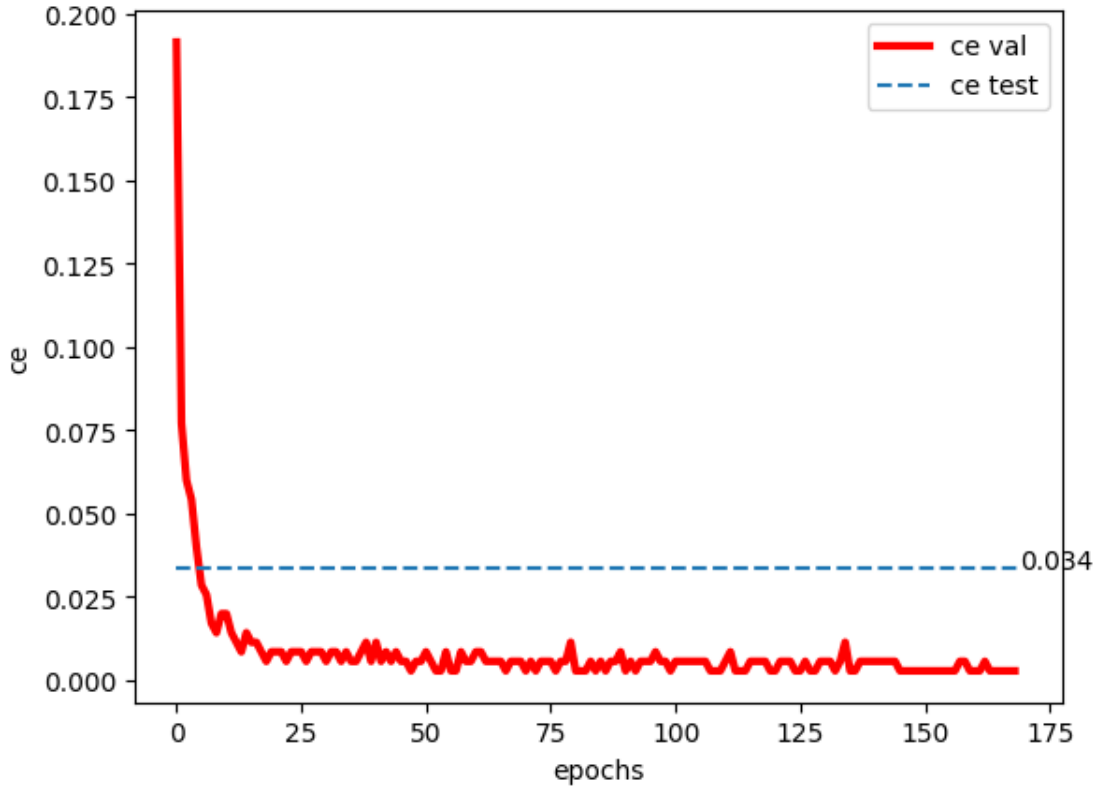
Generally, for NLL and CE the lower the values the better. Additionally, NLL function is used for minimization problems and aims to minize. NLL should decrease as epochs increase. This would indicate that the model is learning and improving its predictions over time, which gives then a lower classification error along with lower nll values. Models are trained over several epochs. As a result, the graphs show the validation and testing process results.

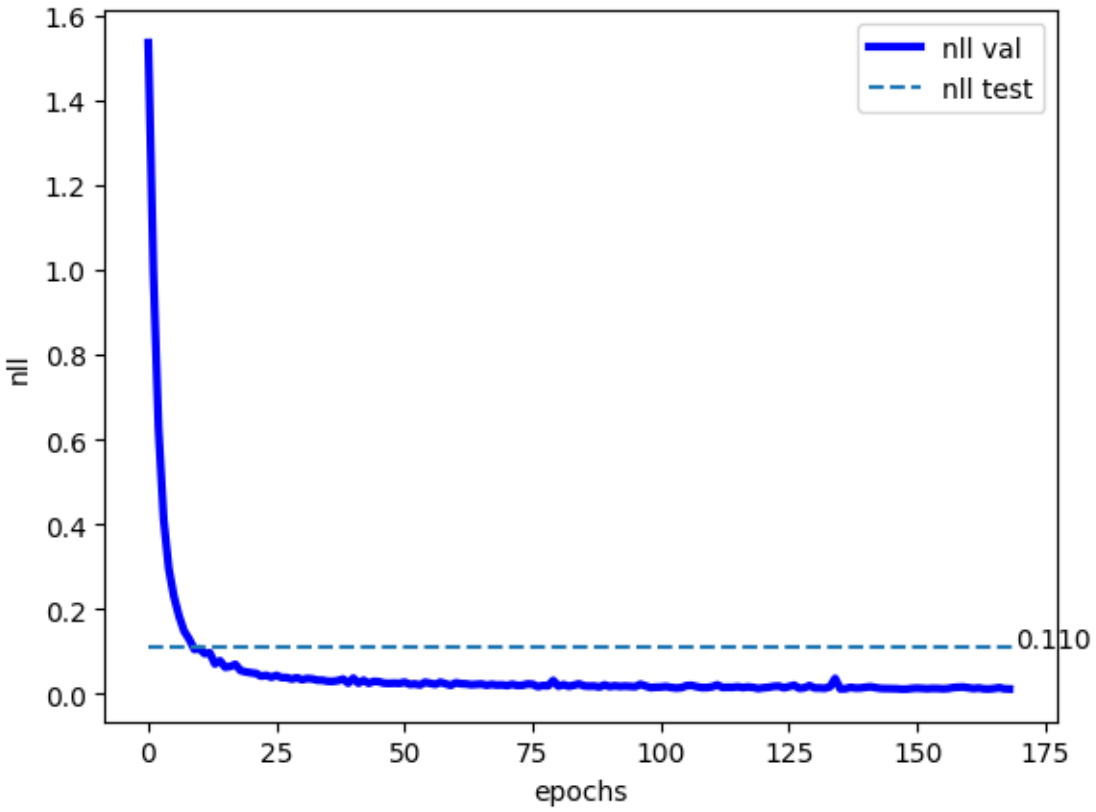
NLL Plot: - For NLL-epochs graphs, the decreasing of the curve is almost the same for both. Around 10 epochs, the curve for both CNN and MLP decreases sharply. This could mean that the models learn quickly. Approximately after 20th epochs models' validaiton sets beginning to stabilizing. MLP model has 0.216 as nll test from evaluation, which indicates a significant divergence between validaiton curve and test set line. This could mean that model is not performing well or its overfitting. On the other hand, CNN one has 0.110 for test set. In this case validation and test sets does not have a significant divergence, closer than MLP ones. Additionally, test set of CNN one has a lower value. Considering the idea the lower the values the better for NLL, since the main idea is to minimize, CNN model performed better with a lower value as final evaluation with test set compared to MLP one.

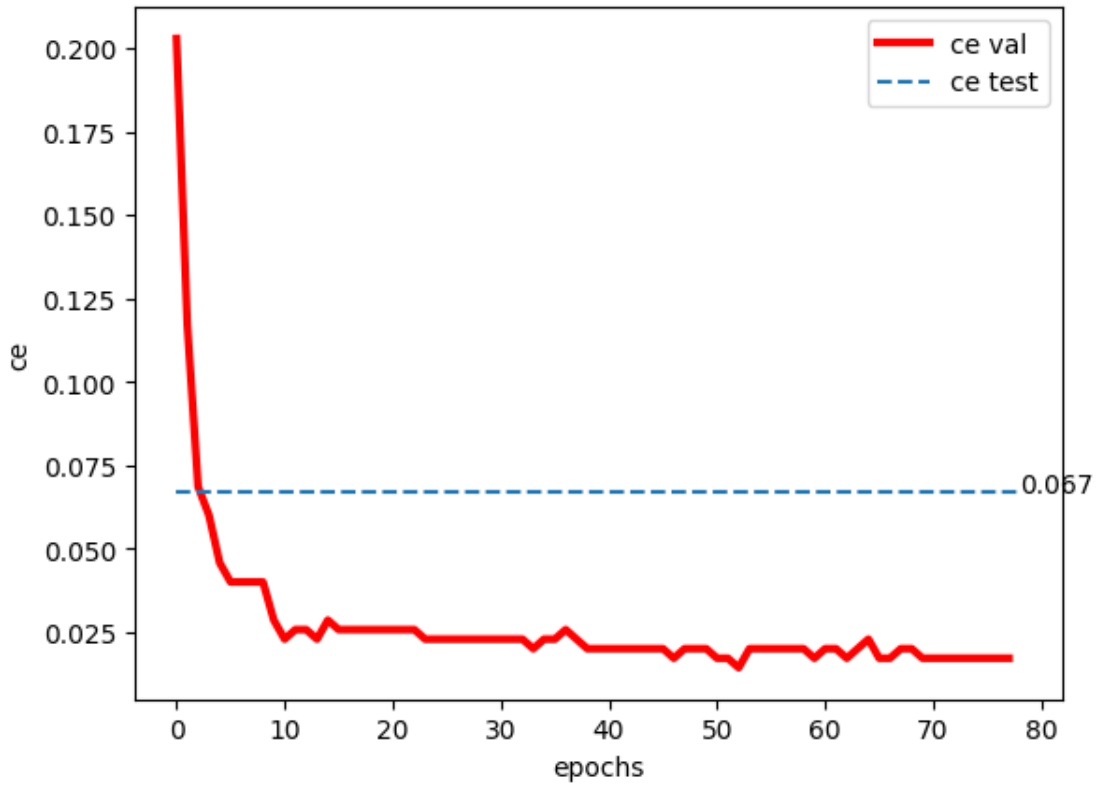
CE Plot: - Similar to the NLL one, around 10 epochs, the curve for both CNN and MLP decreases sharply and approximately after 20th epochs both models' validaiton sets beginning to stabilizing. MLP model has 0.067 as nll test, which indicates again a significant divergence between validaiton curve and test set line during the training and evaluation. Then again this could indicate that model is not performing well. HOwever, CNN one has 0.034 as test set and does not have considerable divergence. Addition to that CNN one has a lower CE value approximately 0.01 for val set, which is lower than MLP one (around 0.02 for val set).

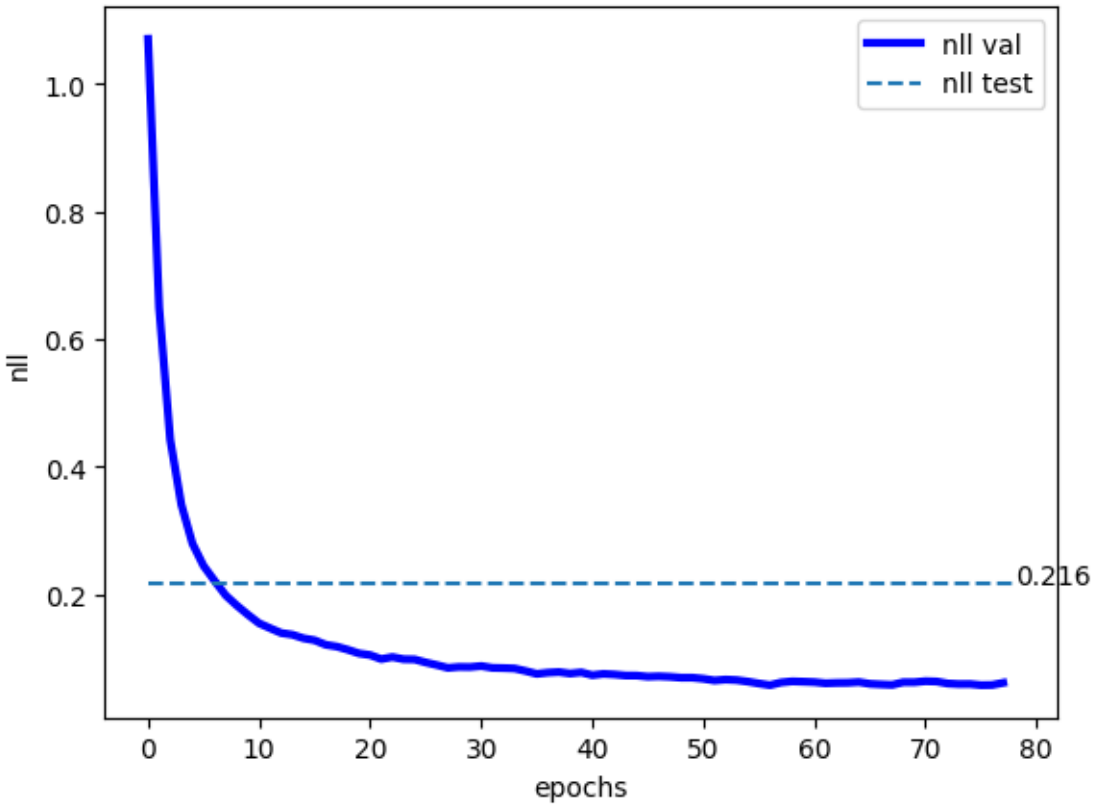
In both CNN and MLP CE-Plots, graphs fluctuate between a certain CE range after the 10th epochs compared to NLL ones (for CNN its approximately between 0.020-0.000 and for MLP around 0.025-0.010). This is because models essentially improve their predictions over time. However, this may not always result in simple improvements in every epoch due to the changing nature of mini-batches and the inherent stochasticity in the optimization process. As a result, it is not fixed on a certain CE value. This fluctuates in MLP seems much more compared to CNN, which could be due to the nature of their architecture structure.

MLP model stops around 100 epochs. CNN model stops around 140 epochs. Hereby, training doesn't improve for longer than 20 epochs, so it is stopped at 100 and 140 epochs respectively for MLP and CNN (since max_patience = 20). So for MLP 100 epochs and for CNN 140 epochs is enough for models to learning and training.




nll val
nll test
nll
epochs
0.110
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175


ce val
ce test
ce
epochs
0.067
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
10
20
30
40
50
60
70
80



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:

Yes, as i stated in the previous question, CE and NLL final test values for CNN model are lower than MLP. So CNN performed better for the loss value and classification error of the minimization task.

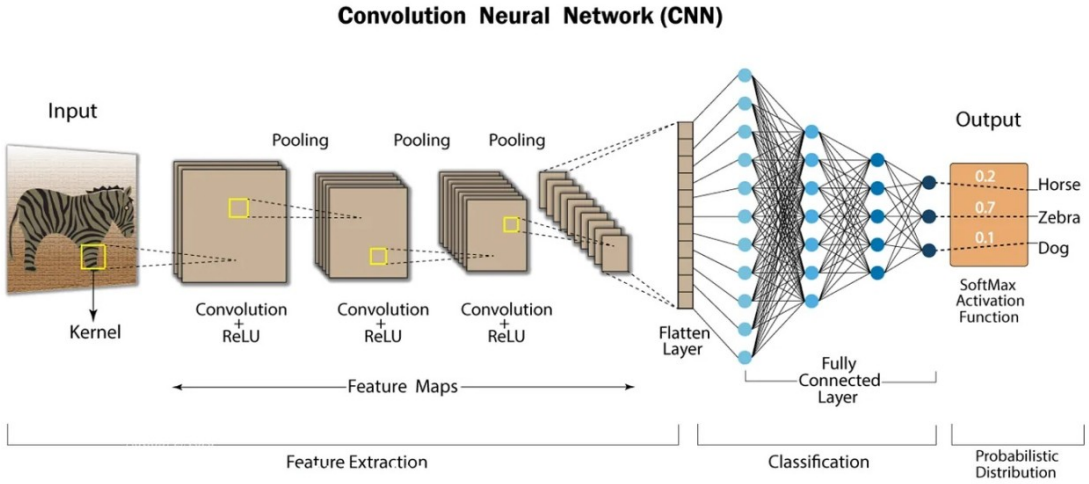
Generally, CNN models are used for image recognition because of their unique architecture system that works consist of convolutional layers. Its layers and working system for the image recognition is much more advanced than the MLP ones.

MLP treats input data as a flat vector and loses spatial information, so it changes the positioning of the pixels of the input image through the running process. HOwever, CNN maintain the spatial structure of the image throughout the network by applying filters within the layers that capture spatial hierarchies which allows CNNs to learn more efficiently from image data. Additionally, it requires fewer parameters due to weight sharing in convolutional layers. Pooling layers in CNNs reduce the spatial dimensions of the data, helping to make the feature detection more robust and computation more efficient.

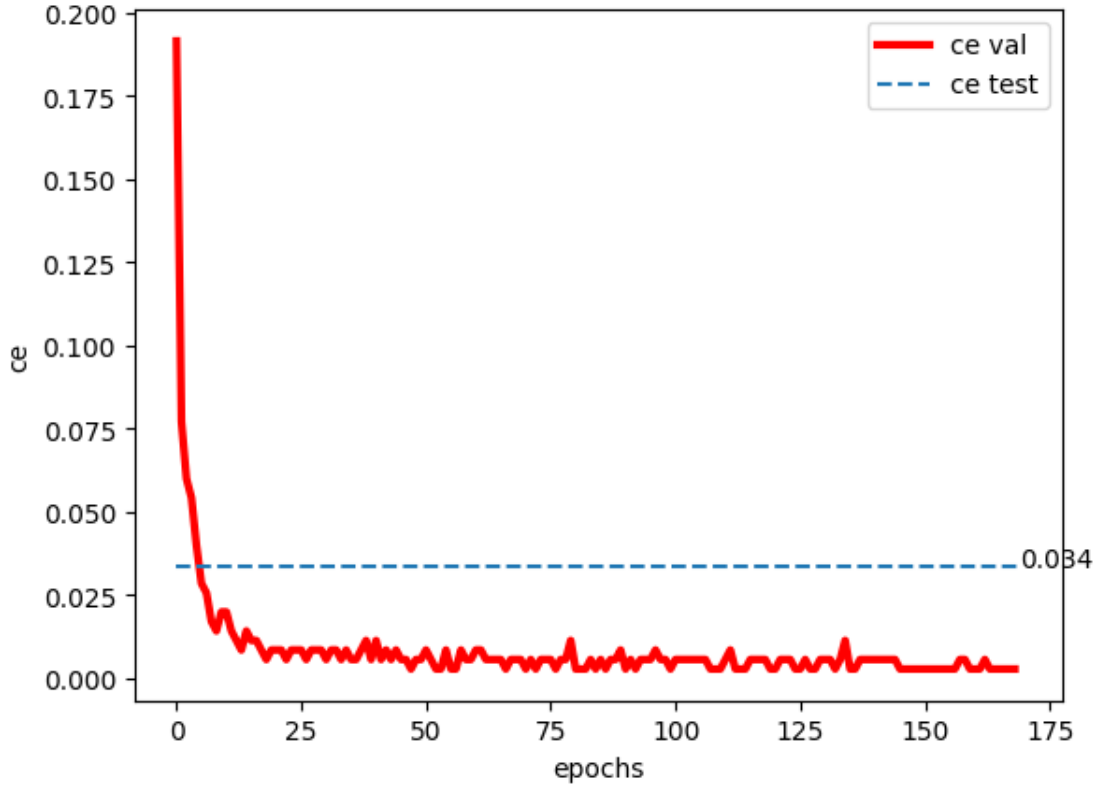
This also can be seen from the code above, the architectures of the both models are quite different. CNN ones uses convolutional layers along with pooling and regularizations (ReLU, batch and dropout) and does contain a Linarity for the fully connected part. In contrast, MLP model only has the layers of fully connected part, which could not be enough for complex data.

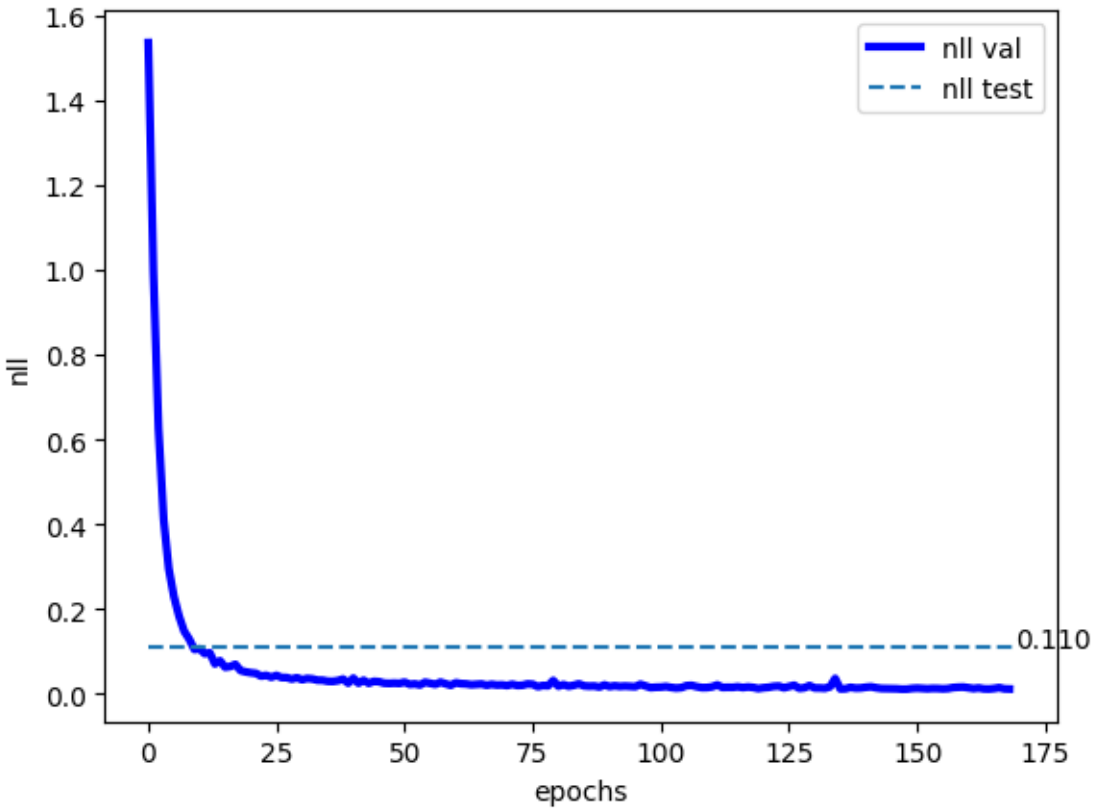
(referance: https://www.researchgate.net/post/Why_CNN_is_better_than_SVM_for_image_classification_ar
classification_machine_learning_or_deep_learning#:~:text=The%20Convolutional%20Neural%20Network%20(

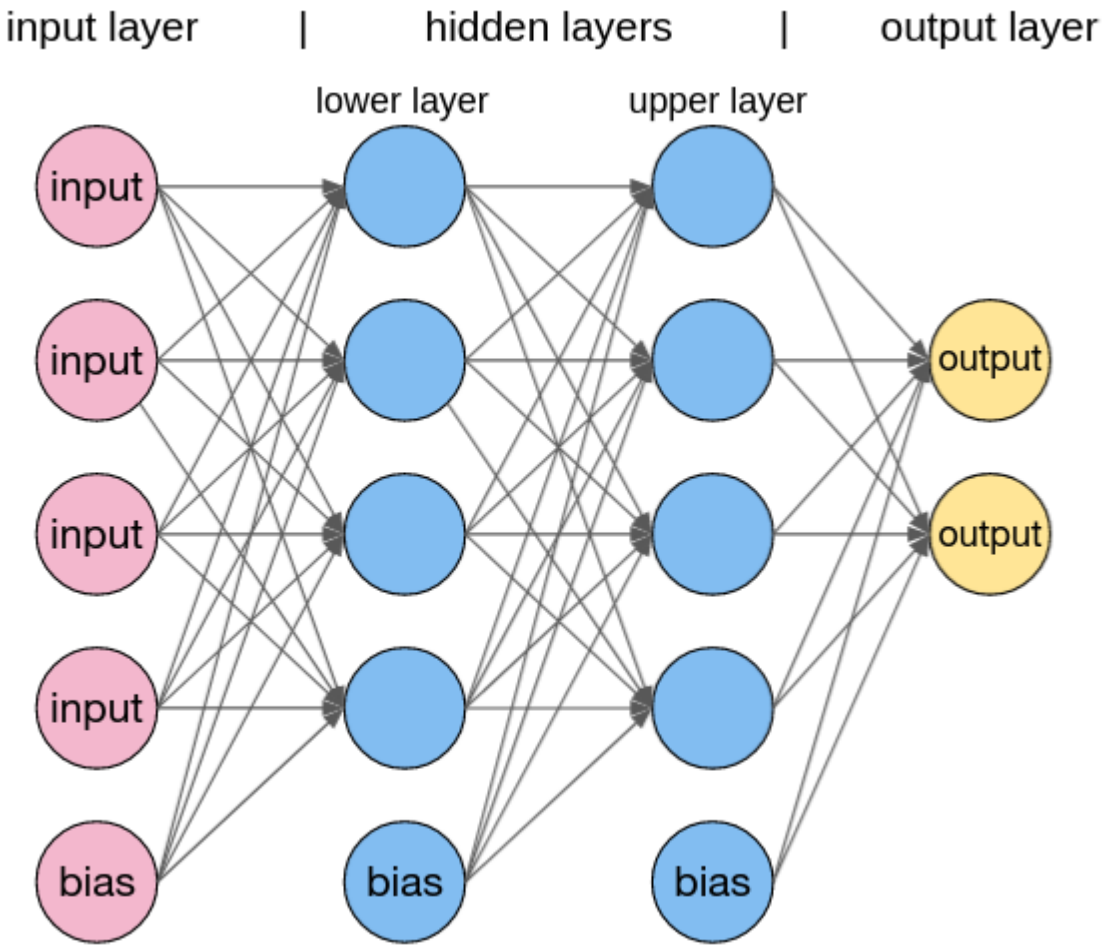
To illustrate the different architecture of CNN and MLP, hereby i placed revelant images.



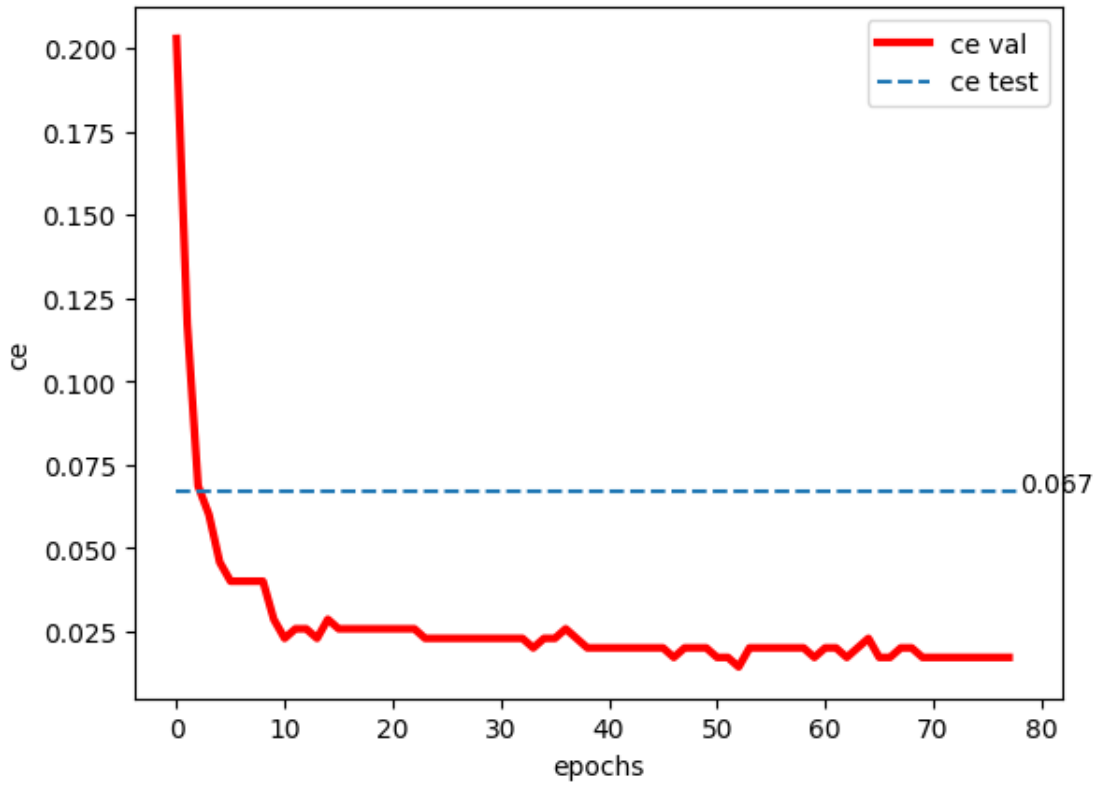
CNN

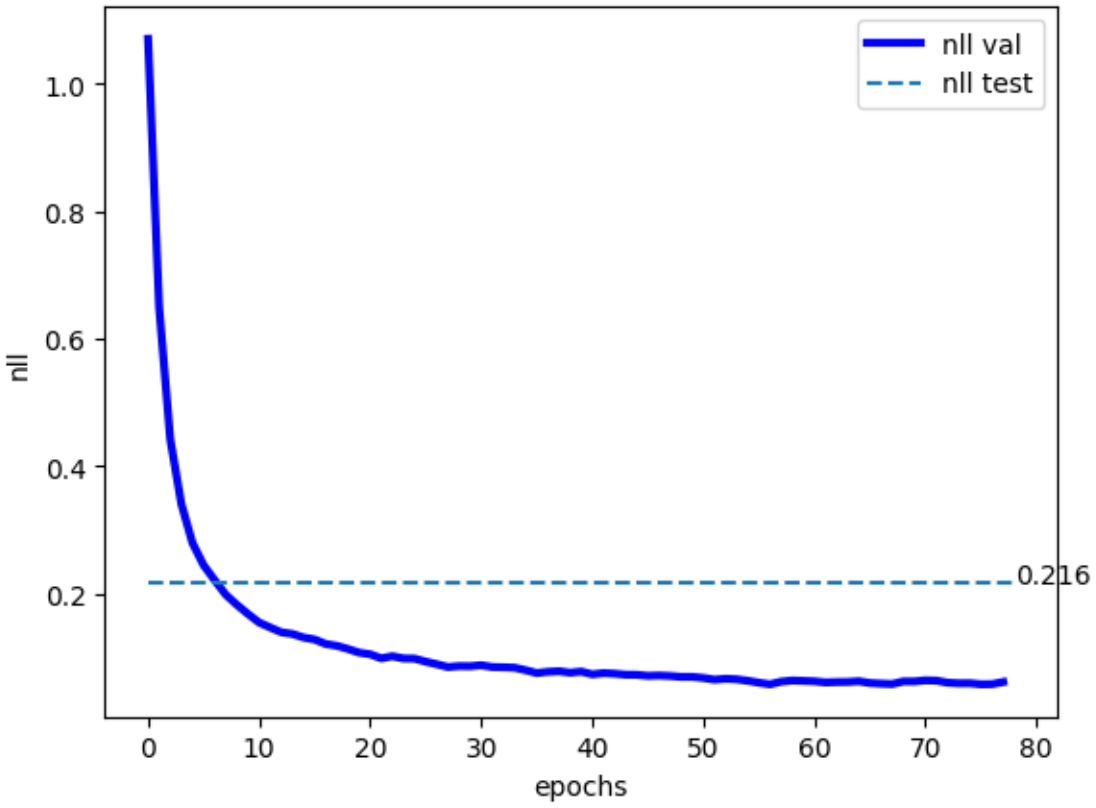



nll val
nll test
nll
epochs
0.110
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175


input layer
|
hidden layers
|
output layer
lower layer
upper layer
input
input
input
input
bias
bias
bias
output
output

MLP


ce val
ce test
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
ce
0.067
0
10
20
30
40
50
60
70
80
epochs


nll val
nll test
1.0
0.8
0.6
0.4
0.2
nll
0.216
0
10
20
30
40
50
60
70
80
epochs

1.4 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%5B8,set%20of%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.

1.5 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!

[312]:
```
import torch
from torch.utils.data import Dataset
import numpy as np
import scipy.io

class SVHNDataset(Dataset):
    """Street View House Numbers (SVHN) Dataset."""

    def __init__(self, mode="train", transforms=None):
        # file paths
        train_file = '/Users/selma/Desktop/computational int/train_32x32.mat'
        test_file = '/Users/selma/Desktop/computational int/test_32x32.mat'

        # loading the data
        # for training and validation train file will be used
        # for testing, test file will be used
        if mode in ['train', 'val']:
            data = scipy.io.loadmat(train_file)
        elif mode == 'test':
            data = scipy.io.loadmat(test_file)

        images = np.transpose(data['X'], axes=(3, 2, 0, 1))  # Reordering␣
↪dimensions to NCHW for architecture
        labels = data['y'].astype(np.int64).squeeze()
        labels[labels == 10] = 0  # Change labels from 10 to 0 for digit '0'
```

```
        # Split the data according to the mode
        if mode == 'train':
            self.images = images[:int(0.9 * len(images))]  # 90% for training
            self.labels = labels[:int(0.9 * len(labels))]
        elif mode == 'val':
            self.images = images[int(0.9 * len(images)):]  # Last 10% for␣
 ↪validation
            self.labels = labels[int(0.9 * len(labels)):]
        elif mode == 'test': # test file
            self.images = images
            self.labels = labels

        self.transforms = transforms

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

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        # Convert numpy array to a float tensor and normalize
        image = torch.from_numpy(image).float() / 255.0

        if self.transforms:
            image = self.transforms(image)

        return image, label
```

VISUALIZATION OF THE DATA

[311]:
```
import torch
import matplotlib.pyplot as plt

# Initialize your dataset class
dataset = SVHNDataset( mode="test") #chnage to see val and train too

# Setup figure for visualization
fig, axs = plt.subplots(4, 4, figsize=(8, 8))  # this part increases the␣
 ↪quality
for i in range(4):
    for j in range(4):
        index = 4 * i + j
        image, label = dataset[index]
        image = image.permute(1, 2, 0).numpy()  # Converting from CxHxW to␣
 ↪HxWxC for matplotlib
```

```
        axs[i, j].imshow(image)
        axs[i, j].set_title(f"Label: {label}")
        axs[i, j].axis('off')

plt.show()
```

NN CLASSIFIER

[313]:
```
# 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)
```

[314]:
```
# =========
# 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):
        # ------
        # Compute the logits by passing input x through classnet
        logits = self.classnet(x)
        # Convert logits to class probabilities using log_softmax
        log_probs = F.log_softmax(logits, dim=1)
        # Find the class with the maximum probability
```

```
        # this is useful and efficient for later training phrase
        y_pred = log_probs.argmax(dim=1)

        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"):
        # ------
        # Pass input x through the classnet to get the logits
        logits = self.classnet(x)
        # Calculate log-softmax of logits for NLLLoss
        log_probs = F.log_softmax(logits, dim=1)

        # Calculate the loss using NLLLoss
        loss = self.nll(log_probs, y)
        # ------
        if reduction == "sum":
            return loss.sum()
        else:
            return loss.mean()
```

evaluation

[315]:
```
# 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="sum")
        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()
```

training procedure

[316]:
```
# 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
```

[318]:
```
from torch.utils.data import DataLoader

train_data = SVHNDataset(mode="train")
val_data = SVHNDataset(mode="val")
test_data = SVHNDataset(mode="test")

training_loader = DataLoader(train_data, batch_size=128, shuffle=True)
val_loader = DataLoader(val_data, batch_size=128, shuffle=False)
test_loader = DataLoader(test_data, batch_size=128, shuffle=False)


# 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␣
 ↪longer than 20 epochs, it is stopped
```

[319]:
```
# Fetch a sample to see data format
print("Feature example from the dataset:")
feature_example, label_example = train_data[1]
print(f"Feature shape: {feature_example.shape}, Label: {label_example}")
print("Shape of train_features:", train_features.size())

# Fetch a batch and see its shape
train_features, train_labels = next(iter(training_loader))
print("\nFeature batch shape from DataLoader: ", train_features.size())
print("Labels batch shape from DataLoader: ", train_labels.size())
```

```
# Flatten operation (if necessary after all convolutional operations)
flatten = Flatten()
train_features_flattened = flatten(train_features)
print("\nFeature batch shape after flatten operation: ",␣
 ↪train_features_flattened.size())
```

```
Feature example from the dataset:
Feature shape: torch.Size([3, 32, 32]), Label: 9
Shape of train_features: torch.Size([64, 64])

Feature batch shape from DataLoader:  torch.Size([128, 3, 32, 32])
Labels batch shape from DataLoader:  torch.Size([128])

Feature batch shape after flatten operation:  torch.Size([128, 3072])
```

1.6 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%20datas

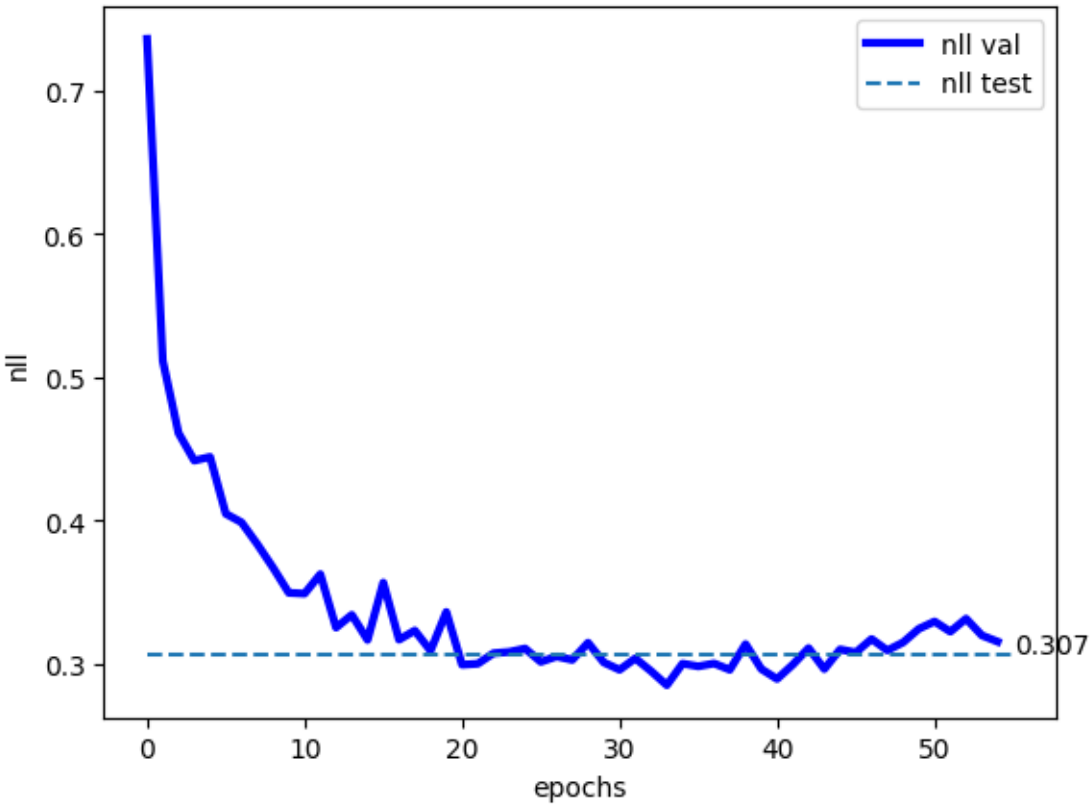
1.7 3. (1pt) Think of an extra component that could improve the performance (e.g., a regularization, specific activation functions).

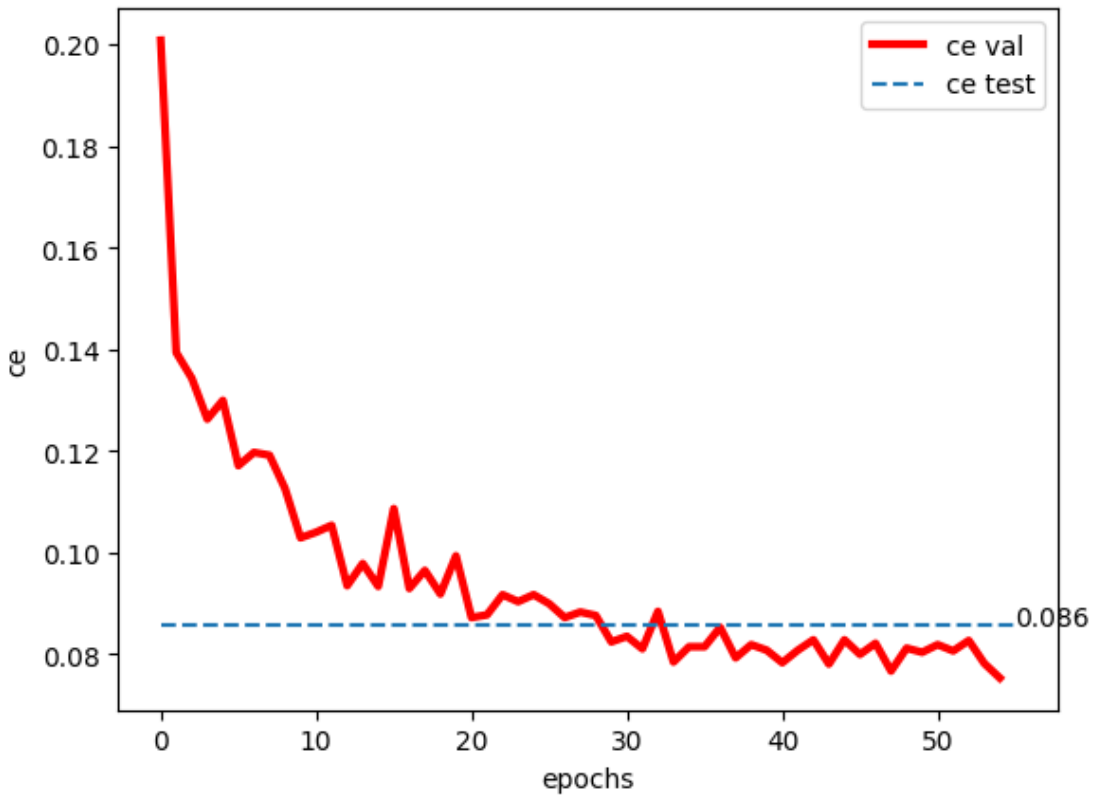
[320]:
```
#Took this code from part 2.
# I only changed the necessery parts.

names = ["test_model"]  #changed names

# same file logic as above
for name in names:
    print("\n-> START {}".format(name))
    model_name = name + "_M_" + str(M) + "_kernels_" + str(num_kernels)

    # Create a folder if necessary
    result_dir = os.path.join(base_dir, model_name + "/")


    if not (os.path.exists(result_dir)):
        os.mkdir(result_dir)


    classnet = nn.Sequential(
    # First convolutional layer
    nn.Conv2d(3, 32, kernel_size=5, stride=1, padding=2),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
```

```
    # Second convolutional layer
    nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_kernels * 2),  #batch normalization
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    # Flatten the output for the dense layers
    Flatten(),

    # Fully connected layer
    nn.Linear(64 * 8 * 8, 256),
    nn.Dropout(0.5),  #dropout to reduce overfitting
    nn.ReLU(),

    # Output layer
    nn.Linear(256, 10),
    nn.LogSoftmax(dim=1)
)

    # 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_loader)
    # write the results to a file
```

```
    with open(os.path.join(result_dir, model_name + "_test_loss.txt"), "w") as␣
↪f:
        f.write("NLL: {}\nCE: {}".format(test_loss, 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 test_model
Epoch: 0, val nll=0.7362469973344149, val ce=0.20079170079170078
Epoch: 10, val nll=0.3491705024369323, val ce=0.10401310401310401
Epoch: 20, val nll=0.2995599317902136, val ce=0.08722358722358722
Epoch: 30, val nll=0.29603268375756014, val ce=0.08353808353808354
Epoch: 40, val nll=0.28962747574893954, val ce=0.07835107835107835
Epoch: 50, val nll=0.3297235078574724, val ce=0.0819000819000819
-> FINAL PERFORMANCE: nll=0.30726114027179235, ce=0.08570221266133989
```


nll val
nll test
0.7
0.6
0.5
0.4
0.3
nll
0.307
0
10
20
30
40
50
epochs



1.8 4. (1pt) Provide a good explanation of the applied architecture and a description of all components.

SVHNDataset and relevant codes

- SVHNDataset(Dataset) class is responsible for creating training, validation and test datasets and the labels for 32x32 data. The code reorders the dimensions to NCHW format since CNN use convolutional layers which expect input tensors in this format. In PyTorch N is for batch size, C is number of channels, H is height, and W is for width. Hereby, i did not used Reshape() function but used 'np.transpose' in SVHNDataset(Dataset). Additionally, for loading the data files, I used paths. Other than that the logc of the code is same as the one provided at the beginning of this asgm. I used the same structure, but chnaged data splitting part, since my data is different (e.g this part images[:int(0.9 * len(images))]).
- For Reshape, Flatten, ClassifierNeuralNet, evaluation, and training parts i used the same function that provided in the second part.

- The initialization of the dataloaders has been implemented according to the SVHN-Dataset(Dataset) class.
- I used the same hyperparameter structure from second part but did not used all of them, since my data was different.

The architecture and extra components for improvements:

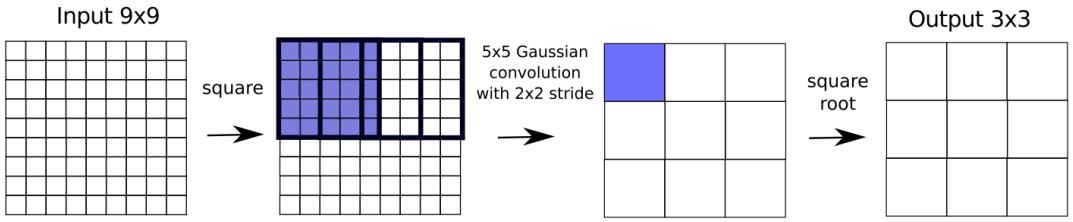
```
classnet = nn.Sequential(

# First convolutional layer
nn.Conv2d(3, 32, kernel_size=5, stride=1, padding=2),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Second convolutional layer
nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
nn.BatchNorm2d(num_kernels * 2), #batch normalization
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Flatten the output for the dense layers
Flatten(),

# Fully connected layer
nn.Linear(64 * 8 * 8, 256),
nn.Dropout(0.5), #dropout to reduce overfitting
nn.ReLU(),

# Output layer
nn.Linear(256, 10),
nn.LogSoftmax(dim=1))
```

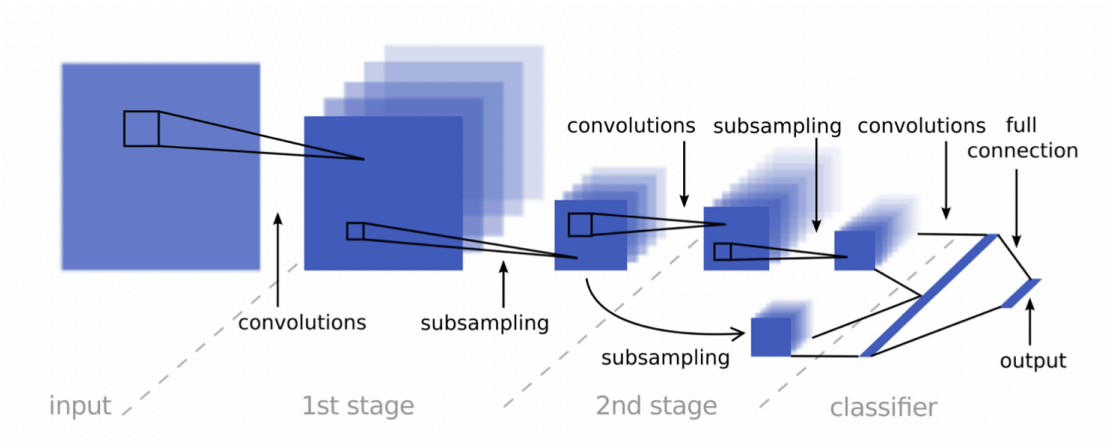
In this architecture two convolutional layers is used, each followed by a ReLU activation for non-linearity and a max pooling layer to reduce spatial dimensions and parameter counts. After the second convolutional layer batch normalization is applied. This improves the stability of the network. After the feature extraction and learning part, for the classification part; Flatten is applied. This converts the feature maps of 2D into a 1D feature vector which is necessary for the fully connected layers, the second part of the CNN. For fully connected layers two Linear layers are used followed by a ReLU activation for non-linearity and Dropout for preventing overfitting. As for the output layer LogSoftmax is used which provides probabilities of each class. Hereby, I considered, Batch Normalization and Dropout as an extra components.

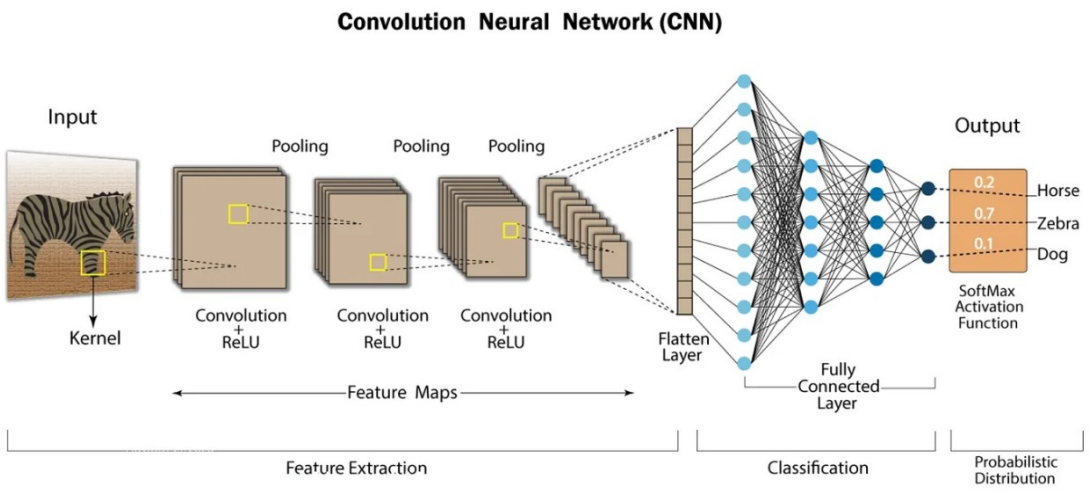
- nn.Conv2d(3 and 32, output = 32 and 64, kernel_size=5, stride=1, padding=2): To build a convolutional neural network this function is applied. In PyTorch it represents a two-dimensional convolutional layer which extractes features from the input image. The input planes/channels are used to produce feature maps. Hereby, input_channels, output_channels, kernel_size, stride and padding is used as parameters. According to this (https://stats.stackexchange.com/a/292064/82135) source, since data has colors channel input set to 3, and output to be 32 (since its 32x32).
  In the second convolutional layer channel input is set to 32, this is because the first layers output was 32 and output for this layer would be 64 whihc doubles the depth to focus on more complex features. As for the other parameters, 5 for kernel size is a general use, which is for the size of the filter. Stride is for pixel and padding is for keeping the output size same as input by adding padding to the input.(https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html).

- nn.ReLU(): Convolutional layers, without any activation functions, perform as linear. However, ReLU, regularization, is responsible for the non-linearity which allows model to learn complex patterns.
- nn.MaxPool2d(kernel_size=2, stride=2): This function takes the maximum value fromt he input and populates it. This helps model to focus on important features and increase the performance by reducing complexity and improving computational efficiency.
- nn.BatchNorm2d(num_kernels * 2): Normalizes the input layer by centering and scaling. Can be considered as regularizer too (https://medium.com/analytics-vidhya/everything-you-need-to-know-about-regularizer-eb477b0c82ba). This is important because it stabilizes the learning of the model by normalizing and reduces internal covariate shift which also prevent overfitting and perform better on deep complex netwroks.
- Flatten(): Before going into fully connected layers, this function transforms 2D feature maps into 1D feature vectors.
- nn.Linear(64 * 8 * 8, 256): In. this case total number of features are the first parameter, and second parameter is the number of neurons in the layer. So it maps the flattened features to a hiddden layer of 256 units.
- nn.Dropout(0.5): This function is responsible for preventing overfitting and relying on specific features. It randomizes some of the elements of the input tensor with probability 50%.

As for the order of the layers and the structure of the architecture, I took the following images and the lectures as example too.







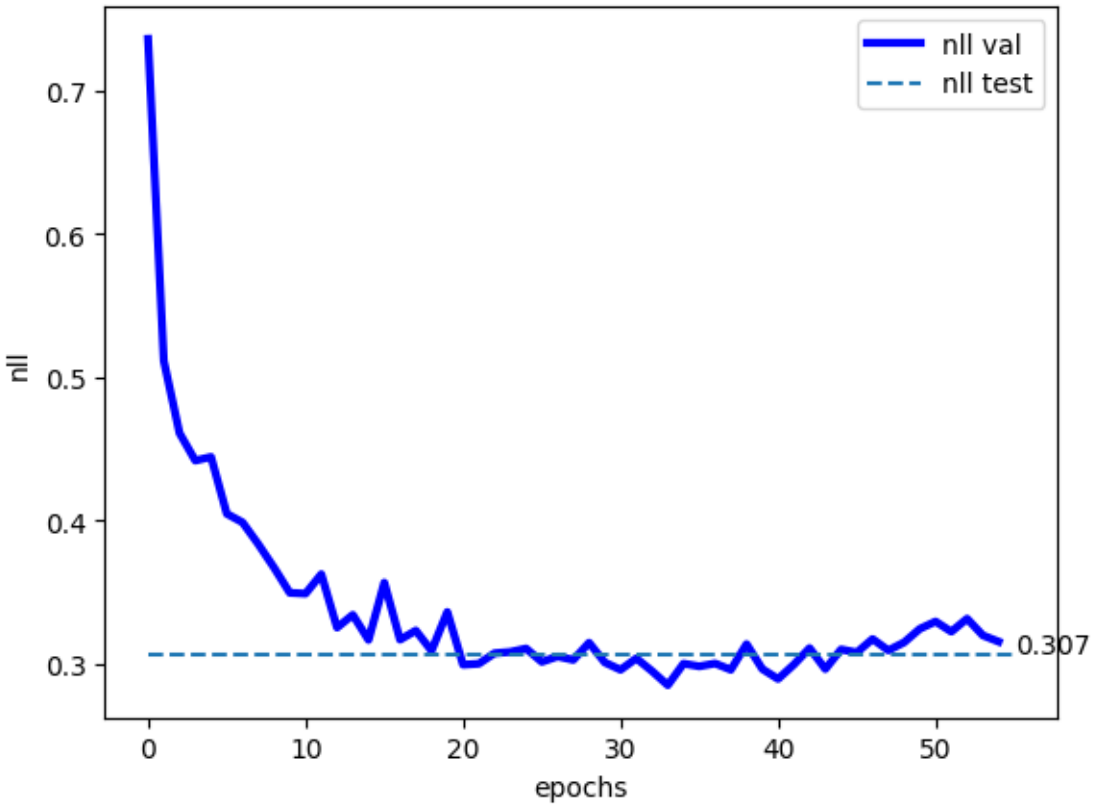
1.9 5. (2pt) Analyze the results.

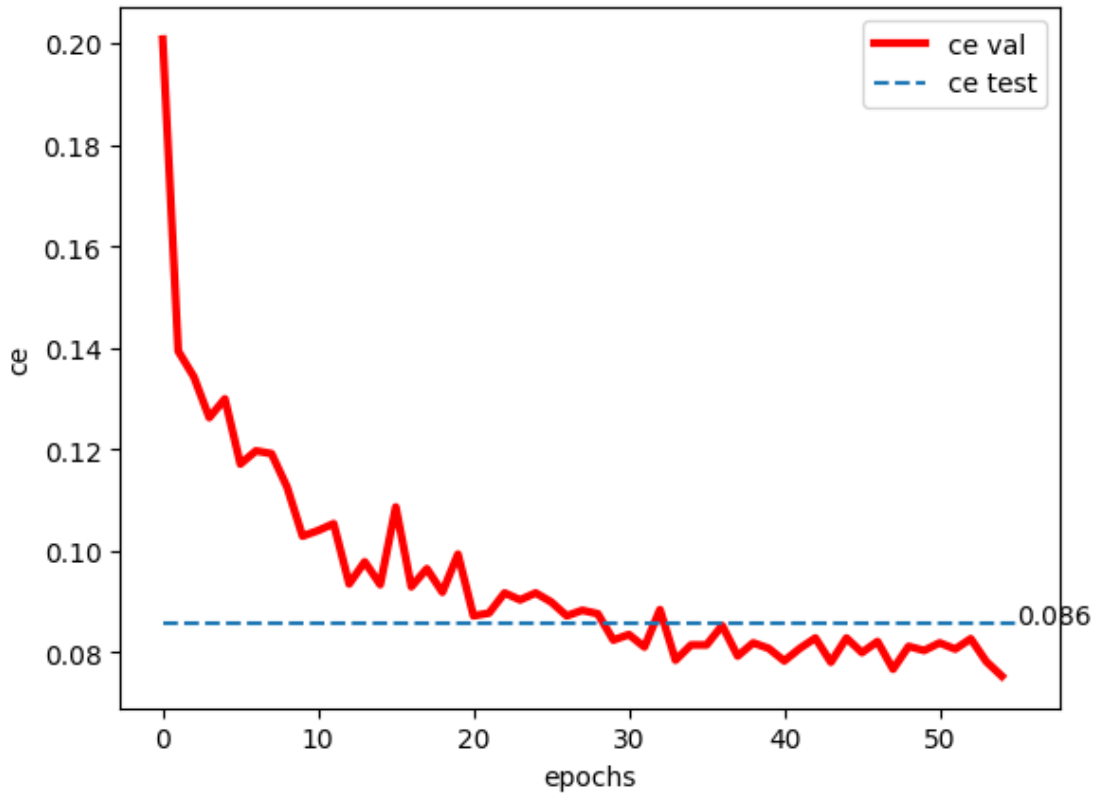
NLL Plot: During the initial epochs' validation curve decreases sharply. This demonstrates that the model quickly learns from the training data which could also mean that chosen parameters are optimized effectively to reduce prediction errors. Approximately after 10 epochs, NLL becomes more stabilized. When the validation NLL settles approximately between some values, it could mean that there is no longer any benefit for model to learn more patterns from the training set without overfitting. As a final value for the test set, 0.307, and the stabilized validation set approximately between 0.325-0.250, there seems to be no gap. So, this could mean that the model generalizes well without overfitting to the training data.

CE Plot: Like the NLL plot classification error decreases within the initial epochs starting of the training rapidly, which also states that model learns quickly from the training data. Approximately after 10 epochs CE becomes more stabilized. Hereby, as a result, the test set has around 0.086 which is lower than 0.1 as asked and could indicate that model performs well. Considering there is almost no significant divergence between the validation curve and test set line, I believe it is true to assume that there is no overfitting.

The model's graphs fluctuate between a certain CE and NLL range after the 20th epochs (approximately between 0.325-0.250 for NLL and 0.9-0.07 for CE). This is because CNN essentially improve its predictions over time. However, this may not always result in simple improvements in every epoch due to the changing nature of mini-batches and the inherent stochasticity in the optimization process. As a result, it is not fixed on a certain CE or NLL value.

The model stops around 50 epochs, which could be considered as relatively quick, indicating efficient learning dynamic of it. This is also because of the early stop mechanism 'max_patience' which set to 20 (this was already provided) epochs. Hereby, if models are no longer improved by 20 epochs it stops. Thus, since my model reached the min level by 50 epochs, it seems well-tuned and training beyond 50 epochs could not make any important improvements for the model.


nll val
nll test
0.7
0.6
0.5
0.4
0.3
nll
0.307
0
10
20
30
40
50
epochs


ce val
ce test
0.20
0.18
0.16
0.14
0.12
0.10
0.08
ce
0.086
0
10
20
30
40
50
epochs