Student Name: Arash Sajjadi
Student ID: 400422096

Artificial Neural Network (ANN)
Assignment I

Question 1

Online Learning

Online learning is a **machine learning method**. In *online learning*, data becomes available in a particular sequential order and is used to update the best predictor for future data at each step, as opposed to batch learning techniques which generate the best predictor by learning on the entire training data set at once. *Online learning* is a common technique used in machine learning where it is computationally infeasible to train during the whole dataset, requiring out-of-core algorithms.

Online learning vs offline learning

Online learning typically means that you perform learning as the data comes in, while offline learning means that you use static data. Like offline learning methods, *online learning* techniques can be applied to solve a variety of tasks and problems of real-world applications.*Online learning* is used in the following cases.

- Supervised Learning Tasks
- Bandit Learning Task
- Unsupervised Learning Task

Computational Timing

In Offline mode, the complete training data is fed at once to the learning process to adjust weights and parameters automatically, which results in faster and cheaper learning. This mode can be used when we are not expecting a dataset from a different probability distribution means the data and its characteristics are the same in the deployed environment.

Progressive Learning

Progressive learning is a **deep learning framework** for continual learning, whereby tasks are learned in sequence with the ability to use prior knowledge from previously learned tasks to facilitate the learning and executing of new ones.

The beginning of progressive neural networks

A progressive neural network (prognets) is a neural algorithm developed by **Deepmind** in their paper Progressive Neural Networks (Rusu et al., 2016).

The difference between progressive learning and online learning

The existing online sequential learning methods do not require retraining when a “new data sample” is received. Nevertheless, it fails when encountering an unknown “new class of data” to the existing knowledge. The progressive learning technique overcomes this shortcoming by allowing the network to learn multiple new classes alien to existing knowledge encountered at any time. The progressive learning technique enables learning new classes dynamically on the run.

Meta-Learning

Meta-learning in machine learning refers to learning algorithms that learn from other learning algorithms. Most commonly, this means the use of machine learning algorithms that learn how to best combine the predictions from other machine learning algorithms in the field of ensemble learning.

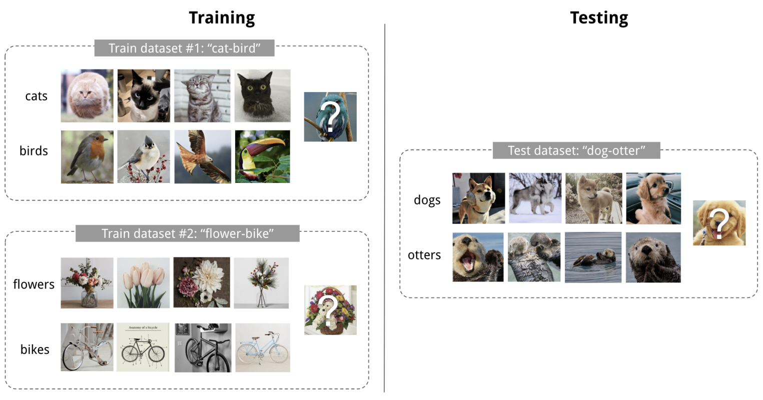
0.1 Define the Meta-Learning Problem

A good meta-learning model should be trained over a variety of learning tasks and optimized for the best performance on a distribution of tasks, including potentially unseen tasks. Each task is associated with a dataset $\mathcal{D}$, containing both feature vectors and true labels. The optimal model parameters are:

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{\mathcal{D}\sim p(\mathcal{D})} \left[\mathcal{L}_\theta(\mathcal{D})\right]$$

It looks very similar to a normal learning task, but one dataset is considered as one data sample.

Few-shot classification is an instantiation of meta-learning in the field of supervised learning. The dataset $\mathcal{D}$ is often split into two parts, a support set S for learning and a prediction set B for training or testing $\mathcal{D} = \langle S, B \rangle$, . Often we consider a K-shot N-class classification task: the support set contains K labelled examples for each of N classes.


Training
Testing
Train dataset #1: "cat-bird"
cats
birds
Train dataset #2: "flower-bike"
flowers
bikes
Test dataset: "dog-otter"
dogs
otters

meta-learning techniques

- Metric Learning
- Model-Agnostic *Meta-Learning* (MAML)
- Recurrent Neural Networks (RNNs)
- Stacking or Stacked Generalization
- Convolutional Siamese Neural Network
- Matching networks

Model-Agnostic Meta-Learning

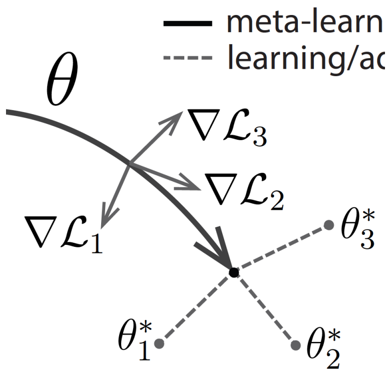
MAML, or Model-Agnostic Meta-Learning, is a model and task-agnostic algorithm for meta-learning that trains a model's parameters such that a small number of gradient updates will lead to fast learning on a new task.

Consider a model represented by a parametrized function f_θ with parameters θ. When adapting to a new task $\mathcal{T}_i$, the model's parameters θ become θ'_i . With MAML, the updated parameter vector θ'_i is computed using one or more gradient descent updates on task $\mathcal{T}_i$. For example, when using one gradient update,

$$\theta'_i = \theta = \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$$

The step size α may be fixed as a hyperparameter or metalearned. The model parameters are trained by optimizing for the performance of $f_{\theta'_i}$ with respect to θ across tasks sampled from $p(\mathcal{T}_i)$. More concretely the meta-objective is as follows:

$$\min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i}) = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta))$$



Note that the meta-optimization is performed over the model parameters θ, whereas the objective is computed using the updated model parameters θ' . In effect MAML aims to optimize the model parameters such that one or a small number of gradient steps on a new task will produce maximally effective behavior on that task. The meta-optimization across tasks is performed via stochastic gradient descent (SGD), such that the model parameters θ are updated as follows:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta'_i})$$

where β is the meta step size.

meta-learning advantages

Meta-learning algorithms help enhance machine learning solutions. Here are the most significant advantages of *meta-learning*:

- Increased model prediction accuracy
- It speeds up the training process and makes it more economical
- Helps build generalized models

Self-supervised Learning

Self-supervised learning is a machine learning process where the model trains itself to learn one part of the input from another part of the input. It is also known as predictive or pretext learning. This process transforms the unsupervised problem into a supervised problem by auto-generating the labels.

The process of the self-supervised learning method is to identify any hidden part of the input from any unhidden part of the input.

Self-supervised Learning types

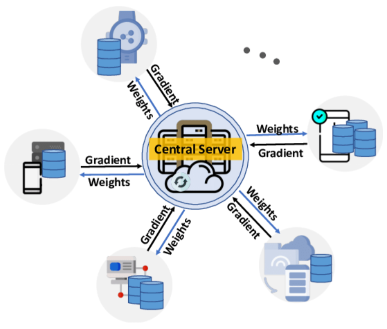
1. **Contrastive self-supervised learning:** it uses both positive and negative examples. Contrastive learning's loss function minimizes the distance between positive samples while maximizing the distance between negative samples.
2. **Non-contrastive self-supervised learning:** NCSSL uses only positive examples. Counterintuitively, NCSSL converges on a useful local minimum rather than reaching a trivial solution, with zero loss.

difference between self-supervised and unsupervised learning

Self-supervised and unsupervised learning methods can be considered complementary learning techniques as both do not need labeled datasets. Unsupervised learning can be considered the superset of *self-supervised learning* as it does not have any feedback loops. On the contrary, *self-supervised learning* has a lot of supervisory signals that act as feedback in the training process.

Federated Learning

Federated learning is a machine learning technique that trains an algorithm across multiple decentralized edge devices or servers holding local data samples without exchanging them. This approach stands in contrast to traditional centralized machine learning techniques, where all the local datasets are uploaded to one server, as well as to more classical decentralized approaches, which often assume that local data samples are identically distributed.



Federated learning types

1. Centralized federated learning
2. Decentralized federated learning
3. Heterogeneous federated learning

Federated learning advantages

Federated learning also enables learning at the edge, bringing model training to the data distributed on millions of devices. At the same time, it allows you to enhance results obtained at the periphery, in the central location.

Federated learning uses

Federated learning is essential in supporting privacy-sensitive data applications where training data is distributed. Theoretically, it sounds like a perfect plan that helps solve problems we face in traditional machine learning models with data on one central server or location.

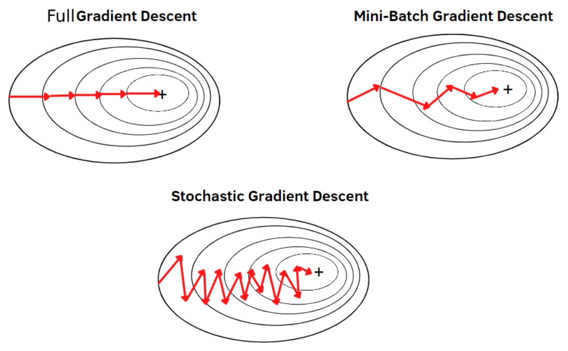
The security of Federated learning

Federated learning is known to be vulnerable to both security and privacy issues. Existing research has focused either on preventing poisoning attacks from users or on concealing the local model updates from the server, but not both.

Question 2

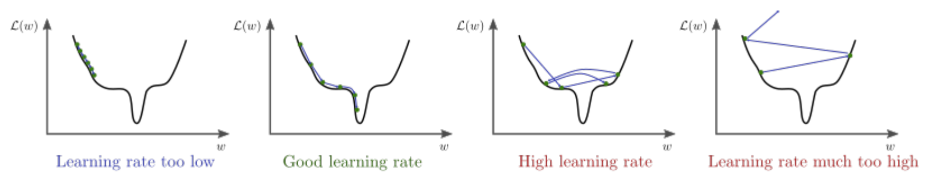
Gradient descent is an optimization algorithm commonly used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the slightest possible error.

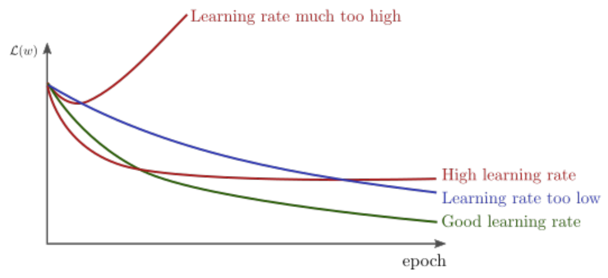
As stated in the question, we have three different types of gradient descent algorithms, and I will review all three.



Let me first make some basic definitions.

- The **learning rate** (also called step size or alpha) is the size of the steps taken to reach the minimum. This is typically a small value and is evaluated and updated based on the behavior of the cost function. High learning rates result in more significant steps but risk overshooting the minimum. Conversely, a low learning rate has small step sizes. While it has the advantage of more precision, the number of iterations compromises overall efficiency as this takes more time and computations to reach the minimum.
- **The cost (or loss) function** measures the difference, or error, between actual y and predicted y at its current position. This improves the machine learning model's efficacy by providing feedback to the model to adjust the parameters to minimize the error and find the local or global minimum. It continuously iterates, moving along the direction of the steepest descent (or the negative gradient) until the cost function is close to or at zero. At





this point, the model will stop learning. Additionally, while the terms cost function and loss function are considered synonymous, there is a slight difference between them. It is worth noting that a loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

- **Convergence in deep learning** In simple words, we can say that convergence of neural networks is a point of training a model, after which changes in the learning rate become lower, and the errors produced by the model in training come to a minimum.

  You can imagine how Gradient Descent (GD) works, thinking that you throw marble inside a bowl and you start taking photos. The marble will oscillate till friction stops it at the bottom. Now imagine that you are in an environment where friction is so tiny that the marble takes a long time to stop completely, so we can assume that when the oscillations are small enough, the marble has reached the bottom (although it could continue oscillating). If we continue taking photos, the marble makes not appreciable movements. So reaching a point where GD makes tiny changes in your objective function is called convergence, which does not mean it has reached the optimal result (but it is pretty near, if not on it).

Types of Gradient Descent

Full (or batch) gradient descent

Full gradient descent sums the error for each point in a training set, updating the model only after all training examples have been evaluated. This process is referred to as a training epoch. While this batching provides computation efficiency, it can still have an extended processing time for extensive training datasets as it still needs to store all of the data in memory. Batch gradient descent also usually produces a stable error gradient and convergence, but sometimes that convergence point is not ideal, finding the local minimum versus the global one.

GD's practical calculation, a balanced error gradient, and stable convergence are some of the benefits of GD. Some weaknesses are that the steady error gradient can often lead to a convergence condition that is not the best the model can accomplish.

Stochastic gradient descent

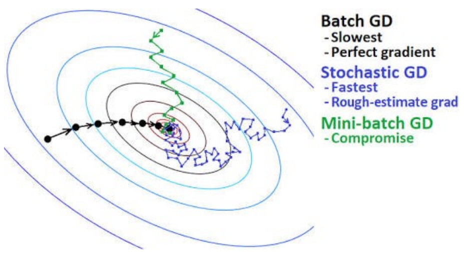
Stochastic gradient descent (SGD) runs a training epoch for each example within the dataset and updates each training example's parameters one at a time. Since you only need to hold one training example, they are easier to store in memory. While these frequent updates can offer more detail and speed, they can result in losses in computational efficiency compared to batch gradient descent. Its frequent updates can result in noisy gradients, which can also help escape the local minimum and find the global one.

For several optimization problems, these three GD algorithms perform well. They can all converge to a promising optimum (local or global), but they still suffer from many challenges, such as the selection of the learning rate, adjustment of the learning rate, etc.

The learning rate η may significantly affect the convergence of GD algorithms. There is a trade-off between the rate of convergence and overshooting when setting a learning rate. If the learning rate is too high, we could OVERSHOOT the minimum and begin to bounce without meeting the minimum. On the other hand, if the learning rate is too poor, the training could take too long.

Mini-batch gradient descent

Mini-batch gradient descent combines concepts from both batch gradient descent and stochastic gradient descent. It splits the training dataset into small batch sizes and performs updates on each of those batches. This approach balances batch gradient descent's computational efficiency and stochastic gradient descent's speed. This provides a balance between SGD robustness and batch GD efficiency



Parameters	Full Gradient Descent	Stochastic Gradient Descent	Mini-Batch Gradient Descent
Accuracy	High	Low	Moderate
Time consuming	More	Less	Moderate

Full gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is usually much faster and can also be used to learn **online**.

Question 3

[1]:
```python
import numpy as np
import math
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
from torchvision import datasets
import matplotlib.pyplot as plt
```

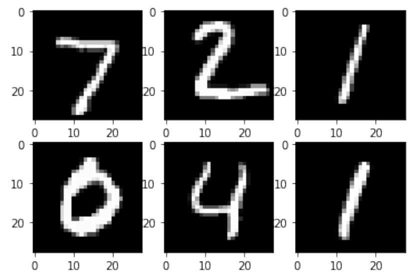
[2]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
input_size = 784 # 28x28
hidden_size = 500
num_classes = 10
num_epochs = 15
batch_size = 128
learning_rate = 0.0001
```

[3]:
```
# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)
test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToTensor())
# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
###########################################
def create_datasets(batch_size):
    # percentage of training set to use as validation
    valid_size = 0.2
    # convert data to torch.FloatTensor
    transform = transforms.ToTensor()
    # choose the training and test datasets
    train_data = datasets.MNIST(root='data',
                                train=True,
                                download=True,
                                transform=transform)
    test_data = datasets.MNIST(root='data',
                               train=False,
                               download=True,
                               transform=transform)
    # obtain training indices that will be used for validation
    num_train = len(train_data)
    indices = list(range(num_train))
    np.random.shuffle(indices)
    split = int(np.floor(valid_size * num_train))
    train_idx, valid_idx = indices[split:], indices[:split]
    # define samplers for obtaining training and validation batches
    train_sampler = SubsetRandomSampler(train_idx)
    valid_sampler = SubsetRandomSampler(valid_idx)
    # load training data in batches
    train_loader = torch.utils.data.DataLoader(train_data,
                                               batch_size=batch_size,
                                               sampler=train_sampler,
                                               num_workers=0)
    # load validation data in batches
    valid_loader = torch.utils.data.DataLoader(train_data,
                                               batch_size=batch_size,
                                               sampler=valid_sampler,
                                               num_workers=0)
    # load test data in batches
    test_loader = torch.utils.data.DataLoader(test_data,
                                              batch_size=batch_size,
                                              num_workers=0)
    return train_loader, test_loader, valid_loader
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/9912422 [00:00<?, ?it/s]
```

```
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./data/MNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]


Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]


Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/4542 [00:00<?, ?it/s]


Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

[4]:
```
examples = iter(test_loader)
example_data, example_targets = examples.next()

for i in range(6):
    plt.subplot(2,3,i+1)
    plt.imshow(example_data[i][0], cmap='gray')
plt.show()
```



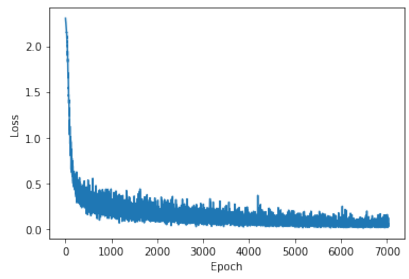
[5]:
```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, math.floor(hidden_size/2))
        self.l3 = nn.Linear(math.floor(hidden_size/2), num_classes)
        self.N=nn.Softmax()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        # no activation and no softmax at the end
        return out

model = NeuralNet(input_size, hidden_size, num_classes).to(device)
print(model)
```

```
NeuralNet(
  (l1): Linear(in_features=784, out_features=500, bias=True)
  (relu): ReLU()
  (l2): Linear(in_features=500, out_features=250, bias=True)
  (l3): Linear(in_features=250, out_features=10, bias=True)
  (N): Softmax(dim=None)
)
```

[6]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[7]:
```
Loss=[]
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # origin shape: [100, 1, 28, 28]
        # resized: [100, 784]
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        Loss.append(loss.item())
        if (i+1) % 200 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}], Loss: {loss.item():.
↪4f}')

plt.plot(Loss)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

```
Epoch [1/15], Step [200/469], Loss: 0.5328
Epoch [1/15], Step [400/469], Loss: 0.3397
Epoch [2/15], Step [200/469], Loss: 0.2381
Epoch [2/15], Step [400/469], Loss: 0.2366
Epoch [3/15], Step [200/469], Loss: 0.2164
Epoch [3/15], Step [400/469], Loss: 0.2223
Epoch [4/15], Step [200/469], Loss: 0.1923
Epoch [4/15], Step [400/469], Loss: 0.2092
Epoch [5/15], Step [200/469], Loss: 0.1598
Epoch [5/15], Step [400/469], Loss: 0.0975
Epoch [6/15], Step [200/469], Loss: 0.1384
Epoch [6/15], Step [400/469], Loss: 0.0872
Epoch [7/15], Step [200/469], Loss: 0.0675
Epoch [7/15], Step [400/469], Loss: 0.1324
Epoch [8/15], Step [200/469], Loss: 0.1658
Epoch [8/15], Step [400/469], Loss: 0.0707
Epoch [9/15], Step [200/469], Loss: 0.0923
Epoch [9/15], Step [400/469], Loss: 0.0886
Epoch [10/15], Step [200/469], Loss: 0.0367
Epoch [10/15], Step [400/469], Loss: 0.1078
Epoch [11/15], Step [200/469], Loss: 0.0705
Epoch [11/15], Step [400/469], Loss: 0.0351
Epoch [12/15], Step [200/469], Loss: 0.0748
Epoch [12/15], Step [400/469], Loss: 0.0669
Epoch [13/15], Step [200/469], Loss: 0.0965
Epoch [13/15], Step [400/469], Loss: 0.0505
Epoch [14/15], Step [200/469], Loss: 0.0177
Epoch [14/15], Step [400/469], Loss: 0.0536
Epoch [15/15], Step [200/469], Loss: 0.0147
Epoch [15/15], Step [400/469], Loss: 0.0766
```



[8]:
```
# Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')
```

```
Accuracy of the network on the 10000 test images: 97.67 %
```

[9]:
```
del model
```

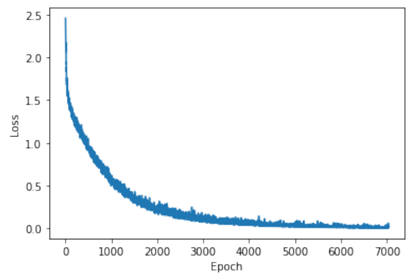
[10]:
```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, math.floor(hidden_size/2))
        self.BN= nn.BatchNorm1d(math.floor(hidden_size/2))
        self.D0=nn.Dropout(0.05)
        self.D1=nn.Dropout(0.15)
        self.l3 = nn.Linear(math.floor(hidden_size/2), num_classes*2)
        self.BN2= nn.BatchNorm1d(num_classes*2)
        self.l4 = nn.Linear(num_classes*2, num_classes)
        self.N=nn.Softmax()
    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out= self.D0(out)
        out = self.l2(out)
        out= self.BN(out)
        out = self.relu(out)
        out= self.D1(out)
        out = self.l3(out)
        out = self.relu(out)
        out= self.BN2(out)
        out = self.relu(out)
        out = self.l4(out)
        # no activation and no softmax at the end
        return out
model = NeuralNet(input_size, hidden_size, num_classes).to(device)
print(model)
```

```
NeuralNet(
  (l1): Linear(in_features=784, out_features=500, bias=True)
  (relu): ReLU()
  (l2): Linear(in_features=500, out_features=250, bias=True)
  (BN): BatchNorm1d(250, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (D0): Dropout(p=0.05, inplace=False)
  (D1): Dropout(p=0.15, inplace=False)
  (l3): Linear(in_features=250, out_features=20, bias=True)
  (BN2): BatchNorm1d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (l4): Linear(in_features=20, out_features=10, bias=True)
  (N): Softmax(dim=None)
)
```

[11]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=0.00001)
```

[12]:
```
Loss2=[]
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # origin shape: [100, 1, 28, 28]
        # resized: [100, 784]
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        outputsval = model(images)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        Loss2.append(loss.item())
        if (i+1) % 200 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}], Loss: {loss.item():.
↪4f}')

plt.plot(Loss2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

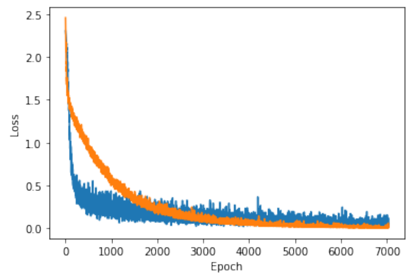
```
Epoch [1/15], Step [200/469], Loss: 1.2548
Epoch [1/15], Step [400/469], Loss: 1.0416
Epoch [2/15], Step [200/469], Loss: 0.8136
Epoch [2/15], Step [400/469], Loss: 0.6774
Epoch [3/15], Step [200/469], Loss: 0.4879
Epoch [3/15], Step [400/469], Loss: 0.4233
Epoch [4/15], Step [200/469], Loss: 0.3841
Epoch [4/15], Step [400/469], Loss: 0.3035
Epoch [5/15], Step [200/469], Loss: 0.2315
Epoch [5/15], Step [400/469], Loss: 0.1693
Epoch [6/15], Step [200/469], Loss: 0.1383
Epoch [6/15], Step [400/469], Loss: 0.1071
Epoch [7/15], Step [200/469], Loss: 0.1324
Epoch [7/15], Step [400/469], Loss: 0.0794
Epoch [8/15], Step [200/469], Loss: 0.1225
Epoch [8/15], Step [400/469], Loss: 0.0515
Epoch [9/15], Step [200/469], Loss: 0.0657
Epoch [9/15], Step [400/469], Loss: 0.0355
Epoch [10/15], Step [200/469], Loss: 0.0242
Epoch [10/15], Step [400/469], Loss: 0.0288
Epoch [11/15], Step [200/469], Loss: 0.0225
Epoch [11/15], Step [400/469], Loss: 0.0264
Epoch [12/15], Step [200/469], Loss: 0.0469
Epoch [12/15], Step [400/469], Loss: 0.0230
Epoch [13/15], Step [200/469], Loss: 0.0126
Epoch [13/15], Step [400/469], Loss: 0.0115
Epoch [14/15], Step [200/469], Loss: 0.0139
Epoch [14/15], Step [400/469], Loss: 0.0090
Epoch [15/15], Step [200/469], Loss: 0.0085
Epoch [15/15], Step [400/469], Loss: 0.0077
```



[13]:
```
# Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')
```

```
Accuracy of the network on the 10000 test images: 98.27 %
```

[14]:
```
plt.plot(Loss)
plt.plot(Loss2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



[15]:
```
del model
model = NeuralNet(input_size, hidden_size, num_classes).to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=0.00001)
```

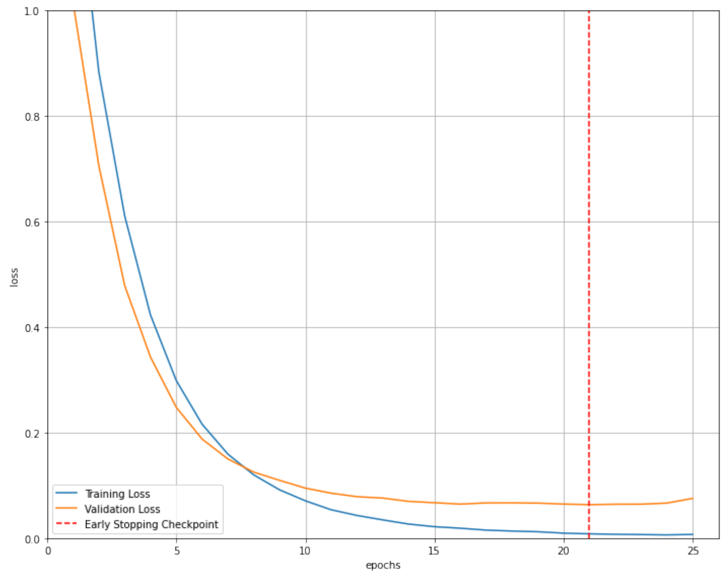
[16]:
```
# import EarlyStopping
from pytorchtools import EarlyStopping
```

[17]:
```python
def train_model(model, batch_size, patience, n_epochs):
    # to track the training loss as the model trains
    train_losses = []
    # to track the validation loss as the model trains
    valid_losses = []
    # to track the average training loss per epoch as the model trains
    avg_train_losses = []
    # to track the average validation loss per epoch as the model trains
    avg_valid_losses = []
    # initialize the early_stopping object
    early_stopping = EarlyStopping(patience=patience, verbose=True)
    for epoch in range( n_epochs ):
        ###################
        # train the model #
        ###################
        model.train() # prep model for training
        #for i, (images, labels) in enumerate(train_loader):
        for batch, (data, target) in enumerate(train_loader,1):
            # clear the gradients of all optimized variables
            data = data.reshape(-1, 28*28).to(device)
            target = target.to(device)
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # record training loss
            train_losses.append(loss.item())
        ######################
        # validate the model #
        ######################
        model.eval() # prep model for evaluation
        for data, target in valid_loader:
            data = data.reshape(-1, 28*28).to(device)
            target = target.to(device)
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the loss
            loss = criterion(output, target)
            # record validation loss
            valid_losses.append(loss.item())
        # print training/validation statistics
        # calculate average loss over an epoch
        train_loss = np.average(train_losses)
        valid_loss = np.average(valid_losses)
        avg_train_losses.append(train_loss)
        avg_valid_losses.append(valid_loss)
        epoch_len = len(str(n_epochs))
        print_msg = (f'[{epoch:>{epoch_len+1}}/{n_epochs:>{epoch_len}}] ' +
                     f'train_loss: {train_loss:.5f} ' +
                     f'valid_loss: {valid_loss:.5f}')
        print(print_msg)
        # clear lists to track next epoch
        train_losses = []
        valid_losses = []
        # early_stopping needs the validation loss to check if it has decresed,
        # and if it has, it will make a checkpoint of the current model
        early_stopping(valid_loss, model)
        if early_stopping.early_stop:
            print("Early stopping")
            break
    # load the last checkpoint with the best model
    model.load_state_dict(torch.load('checkpoint.pt'))
    return  model, avg_train_losses, avg_valid_losses
```

[18]:
```python
batch_size = batch_size
n_epochs = num_epochs+10

train_loader, test_loader, valid_loader = create_datasets(batch_size)

# early stopping patience; how long to wait after last time validation loss improved.
patience = 20

model, train_loss, valid_loss = train_model(model, batch_size, patience, n_epochs)
```

```
[  0/25] train_loss: 1.32480 valid_loss: 1.01598
Validation loss decreased (inf --> 1.015984).  Saving model ...
[  1/25] train_loss: 0.88337 valid_loss: 0.70585
Validation loss decreased (1.015984 --> 0.705851).  Saving model ...
[  2/25] train_loss: 0.61095 valid_loss: 0.47881
Validation loss decreased (0.705851 --> 0.478811).  Saving model ...
[  3/25] train_loss: 0.42300 valid_loss: 0.34329
Validation loss decreased (0.478811 --> 0.343290).  Saving model ...
[  4/25] train_loss: 0.29861 valid_loss: 0.24792
Validation loss decreased (0.343290 --> 0.247920).  Saving model ...
[  5/25] train_loss: 0.21595 valid_loss: 0.18777
Validation loss decreased (0.247920 --> 0.187768).  Saving model ...
[  6/25] train_loss: 0.15921 valid_loss: 0.15009
Validation loss decreased (0.187768 --> 0.150086).  Saving model ...
[  7/25] train_loss: 0.11996 valid_loss: 0.12526
Validation loss decreased (0.150086 --> 0.125260).  Saving model ...
[  8/25] train_loss: 0.09164 valid_loss: 0.10963
Validation loss decreased (0.125260 --> 0.109627).  Saving model ...
[  9/25] train_loss: 0.07079 valid_loss: 0.09484
Validation loss decreased (0.109627 --> 0.094843).  Saving model ...
[ 10/25] train_loss: 0.05382 valid_loss: 0.08527
Validation loss decreased (0.094843 --> 0.085267).  Saving model ...
[ 11/25] train_loss: 0.04308 valid_loss: 0.07861
Validation loss decreased (0.085267 --> 0.078613).  Saving model ...
[ 12/25] train_loss: 0.03464 valid_loss: 0.07597
Validation loss decreased (0.078613 --> 0.075972).  Saving model ...
[ 13/25] train_loss: 0.02678 valid_loss: 0.06975
Validation loss decreased (0.075972 --> 0.069752).  Saving model ...
[ 14/25] train_loss: 0.02180 valid_loss: 0.06716
Validation loss decreased (0.069752 --> 0.067161).  Saving model ...
[ 15/25] train_loss: 0.01889 valid_loss: 0.06454
Validation loss decreased (0.067161 --> 0.064544).  Saving model ...
[ 16/25] train_loss: 0.01534 valid_loss: 0.06684
EarlyStopping counter: 1 out of 20
[ 17/25] train_loss: 0.01356 valid_loss: 0.06695
EarlyStopping counter: 2 out of 20
[ 18/25] train_loss: 0.01226 valid_loss: 0.06644
EarlyStopping counter: 3 out of 20
[ 19/25] train_loss: 0.00955 valid_loss: 0.06467
EarlyStopping counter: 4 out of 20
[ 20/25] train_loss: 0.00827 valid_loss: 0.06346
Validation loss decreased (0.064544 --> 0.063459).  Saving model ...
[ 21/25] train_loss: 0.00742 valid_loss: 0.06431
EarlyStopping counter: 1 out of 20
[ 22/25] train_loss: 0.00691 valid_loss: 0.06441
EarlyStopping counter: 2 out of 20
[ 23/25] train_loss: 0.00622 valid_loss: 0.06649
EarlyStopping counter: 3 out of 20
[ 24/25] train_loss: 0.00720 valid_loss: 0.07544
EarlyStopping counter: 4 out of 20
```

[19]:
```
# visualize the loss as the network trained
fig = plt.figure(figsize=(10,8))
plt.plot(range(1,len(train_loss)+1),train_loss, label='Training Loss')
plt.plot(range(1,len(valid_loss)+1),valid_loss,label='Validation Loss')

# find position of lowest validation loss
minposs = valid_loss.index(min(valid_loss))+1
plt.axvline(minposs, linestyle='--', color='r',label='Early Stopping Checkpoint')

plt.xlabel('epochs')
plt.ylabel('loss')
plt.ylim(0, 1) # consistent scale
plt.xlim(0, len(train_loss)+1) # consistent scale
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
fig.savefig('loss_plot.png', bbox_inches='tight')
```



[21]:
```
# initialize lists to monitor test loss and accuracy
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

model.eval() # prep model for evaluation

for data, target in test_loader:
    if len(target.data) != batch_size:
        break
    data = data.reshape(-1, 28*28).to(device)
    target = target.to(device)
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct = np.squeeze(pred.eq(target.data.view_as(pred)))
    # calculate test accuracy for each object class
    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# calculate and print avg test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            str(i), 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %5d%% (%5d/%5d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 0.059291

Test Accuracy of     0: 99% (974/979)
Test Accuracy of     1: 98% (1120/1133)
Test Accuracy of     2: 98% (1011/1030)
Test Accuracy of     3: 98% (996/1008)
Test Accuracy of     4: 98% (964/980)
Test Accuracy of     5: 97% (870/890)
Test Accuracy of     6: 98% (946/956)
Test Accuracy of     7: 98% (1009/1027)
Test Accuracy of     8: 97% (949/973)
Test Accuracy of     9: 97% (978/1008)

Test Accuracy (Overall):    98% ( 9817/ 9984)
```

The exact number of accuracy is 98.327323718 percent.

The demands of the question

1. Report the depth effect of different hidden layers.
2. Analyze the dropout technique and report its results.
3. Use early stopping criteria.
4. Become familiar with batch normalization and report its effects.

5. The model should be tested for L1 and L2 regularization.
6. Add a regularization term for the weight parameter using the following sample code

Report to the demands of the question

1. This is a simple dataset, so the depth does not have much effect on the learning process. Nevertheless, it is evident that more depth can include more complexity, and also the learning process will be longer.
2. I implemented three neural networks, the first neural network without dropout, the second neural network with dropout, and the third neural network with dropout and early stopping criteria. In general, regarding the effect of dropout, I must say that the speed of decreasing the loss function value reduces later. However, Finally, it is an excellent option to escape from overfitting.
3. Early stopping criteria were used in the training of the third neural network. For this purpose, the dataset should be divided into three parts. The requirements of which were considered from the beginning of the code.
4. Early stopping criteria were used in the training of the third neural network. For this purpose, the dataset should be divided into three parts. The requirements of which were considered from the beginning of the code. However, I used a deficient proportion of dropouts to generalize the learning.
5. I used L2 regularization (using the `weight_decay` command) It should be noted that I also checked L1, and the result of L2 was better than L1. Therefore, I only included the code related to L2 in this report.
6. As I said, I put the regulation in a different way.