CSCI 5922 Lab assignment 2

Jake Krol

March 2025

1 Report

1.1 Experimental design

1.1.1 What will the experiment study?

The experiment will study the effect of architectural and data/loss regularization of a feedforward neural network (FFN) used for a multiclass classification task to classify the number of days (e.g., 0-10 days, 11-20 days, ..., more than 100 days) that a COVID-19 patient will stay in at the hospital. Each sample is a patient case, and features include patient/hospital attributes (e.g., bed quality). There are 11 total date ranges in the dataset, and therefore 11 output logits in the FFN. The dataset was downloaded from Kaggle's COVID-19 Hospitals Treatment Plan (hosted by user arashnic on March, 2025. For context, the best accuracy I found for this dataset on Kaggle is 0.39.

Naturally, model evaluation is measured by accuracy. In this multiclass classification task, accuracy measures the fraction of samples where the output logit with the highest probability corresponds to the ground truth encoding of date ranges; note, date ranges are numerically encoded from $\{0 \dots 10\}$.

Architectural regularization will be tested by applying an 80% probability of **dropout** for each neuron in every hidden layer. While, **data/loss regularization** will be tested by 1) **adding momentum** to the stochastic gradient descent (SGD) optimizer or 2) **step decay** of the learning rate term.

1.1.2 How will it be conducted?

The effect of regularization techniques will be done by **Each regularization technique will be added in isolation** and train/test accuracy will be evaluated. Accuracy across the fixed number of epochs will be compared with other regularization techniques and the baseline.

When adding a single regularization technique, **all other model components, including hyperparameters, from the baseline will stay the same**: number of epochs, number of hidden layers, dimension of hidden layers,activation layer functions, weight initialization strategy, optimizer,batch size, and learning rate. See table 1 for the fixed, baseline hyperparameters.

Also, **various training size proportions (25%, 50%, 75%, and 100%)** will be used for the baseline and regularized models. This will give insight on data efficiency to see how (regularized)-models perform in low and high training data scenarios.

Data pre-processing steps will include one-hot encoding categorical features, scaling continuous feature values (using only the avaiable train proportion), nominal encoding target date ranges, and always stratifying any data subsets by the target vector class balance.

1.1.3 What is the baseline model?

The **baseline model is** an FFN with 4 hidden layers and an output layer; note, all layers have a **non-linear ReLU activation function applied to them, for a total of 5**. After one-hot encoding categorical features in the dataset, the dimension of input vectors are 125. Again, table 1 includes all the hyperparameters of the baseline model.

Hyperparameter	**Value/method**
Epochs	300
# of hidden layers	4
Dimension of each hidden layer	64
Activation function of each hidden layer	ReLU
Parameter initialization method	Xavier Uniform
Optimizer	SGD
Batch size	1024
Learning rate	0.001

Table 1: Baseline model hyperparameters

The **baseline model was trained for 300 epochs** until test accuracy asymptoted. Therefore, all regularized and training data reduced models will **also be trained for 300 epochs**.

1.1.4 Regularization method details

In greater detail, **Architectural regularization** is done by adding 80% probability of dropout of each neuron throughout all hidden layers. While, two different **data/loss** regularization methods will be tested. First, a **momentum=0.9** will add this fraction of the previous gradient step to the current gradient step. Second, **learning rate step decay** will reduce the learning rate by a factor of $\frac{1}{2}$ every 50 epochs.

1.2 Results

Regularization	$\mathbf{ACC_{E_{300}}}$	$\mathbf{ACC_{best}}$	$\mathbf{E_{bestACC}}$	**Train %**	$\mathbf{Avg\ precision_{E_{300}}}$	$\mathbf{Avg\ recall_{E_{300}}}$
None (Baseline)	**0.3839**	–	–	100	0.4180	0.3445
None (Baseline)	0.3836	–	–	75	0.3862	0.3811
None (Baseline)	0.3759	–	–	50	0.3612	0.3855
None (Baseline)	0.3604	–	–	25	0.3640	0.3582
Dropout ($p = 0.8$)	**0.3839**	–	–	100	0.4180	0.3445
Dropout ($p = 0.8$)	0.3836	–	–	75	0.3862	0.3811
Dropout ($p = 0.8$)	0.3759	–	–	50	0.3612	0.3582
Dropout ($p = 0.8$)	0.3604	–	–	25	0.3640	0.3582
Momentum	0.2963	0.3849	33	100	0.2878	0.3091
Momentum	0.3569	0.3864	41	75	0.3561	0.3578
Momentum	0.3540	0.3878	61	50	0.3552	0.3533
Momentum	0.3746	**0.3887**	129	25	0.3901	0.3587
Learning rate decay	0.3692	–	–	100	0.3548	0.3926
Learning rate decay	0.3594	–	–	75	0.3626	0.3547
Learning rate decay	0.3046	–	–	50	0.3178	0.2944
Learning rate decay	0.2834	–	–	25	0.3134	0.2793

Table 2: Test accuracy (ACC) of various regularization methods and training set proportions. $E_{\#}$:= number of epochs. Therefore, E_{300} is the test ACC using the model trained for 300 epochs. And, $E_{bestACC}$ is the epoch associated with the best test accuracy. If $E_{bestACC} = 300$, then a "–" denotes that $ACC_{E_{300}} = ACC_{best}$ and $E_{bestACC} = 300$. Put more simply, if the best accuracy was at 300 epochs, then only that accuracy is reported. Finally, the average precision and recall across classes is reported for test data using the model trained for 300 epochs.

1.3 Analysis

1.3.1 Trends

Training with various regularization methods and proportions of training data indicates two **trends**: **1) test accuracy generally increases as more training data is seen, with some caveats**, and **2) adding momentum to the optimizer is a powerful, yet delicate method which can achieve high test accuracy in low epoch and low training proportion scenarios**.

Discussing trend one in more detail, test accuracy (and average precision/recall) increased as more training data was available in the baseline, dropout, and learning rate decay models, shown by Table 2. However, this effect has diminishing returns, and the model with momentum did not follow this trend. Notably, the top accuracy at 300 epochs ($ACC_{E_{300}}$) was 0.3839 ACC for the baseline and dropout models at 100% training data. Yet, with only 75% of the training data, both the baseline and dropout model perform within 0.0003 ACC of the 100% training data scenario. Meanwhile, the gap between 100% and 25% is quite large. For instance, the learning rate decay model at 100% of training data achieves 0.3692 ACC, but only 0.2834 ACC at 25% of training data (Table 2); this is approximately an 0.08 ACC difference. Similarly, an 0.03 ACC difference occurs at 100% train proportion and 25% train proportion for the baseline and dropout models. Overall, increasing training data positively effected all models, except momentum, but the effect was non-linearly associated with ACC. Where, most ACC gains are in the 2575% range, not 75-100%.

The second trend is that the model with gradient momentum performed dissimilarly to all others. The ACC of momentum peaked at 0.3887 during the 129th epoch with only 25% of the data (Table 2) to achieve the highest performance on test ACC across all methods. Therefore, **momentum was the most data efficient regularization method**. Although, this experiment focused on fixing the number of epochs, ignoring the performance of momentum at lower epochs would be misleading. Now focusing back to the 300 epoch scenario, the ACC of momentum continuously decreased as more training data was used for the model (0.3746 ACC at 25% and 0.2963% at 100%, Table 2). The hypothesis here is that the model was likely overfitting with large gradient due to the added momentum, and the training curves in the appendix section support this. The test ACC of the momentum model seemed prone to overfitting, and it would likely benefit from early stopping or reducing the momentum scaling factor hyperparameter.

Overall, these trends suggest model trainers should try various hyperparameters to explore which methods are efficient. In this study momentum was the most efficient, but it was also heavily prone to overfitting. Furthermore, it's possible the high test accuracy at lower epochs was spurious. Future experiments should include multiple repetitions and a proper validation set to ensure the model can generalize to new data while being "efficient" in the amount of training data used.

1.3.2 Miscellaneous results

- Train loss and train/test accuracy curves as a function of epochs for each model is included in the appendix section.
- The top ACC in this experiment is within 0.01 the ACC of a baseline model associated with this Kaggle dataset.

2 Code

```
#!/usr/bin/env python3
import pandas as pd
import numpy as np
import os
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import torch
from torch import nn
import torch.optim as optim
import sys
# import dataloader
from torch.utils.data import DataLoader
from sklearn.utils import resample
from torch.optim.lr_scheduler import StepLR

# CONST
TRAIN_PROP = 0.50
EPOCHS=300
DROPOUT=True
NAME='dropout'
LR_DECAY=False
MOMENTUM=None
torch.manual_seed(42)

print("CONSTANTS")
print(f"TRAIN_PROP: {TRAIN_PROP}")
print(f"EPOCHS: {EPOCHS}")
print(f"DROPOUT: {DROPOUT}")
print(f"NAME: {NAME}")
print(f"LR_DECAY: {LR_DECAY}")
print(f"MOMENTUM: {MOMENTUM}")


# df = pd.read_csv('host_train.csv')
# df = df.dropna().reset_index(drop=True)
# print(df.head())

# # encoding the categorical columns
# col_conv = [
#     'Department',
#     'Ward_Facility',
#     'Ward_Type',
```

```
#     'Type of Admission',
#     'Illness_Severity',
#     'Age',
#     'Stay_Days'
# ]
# file='lab_assignment2.categories.txt'
# try:
#     os.remove(file)
# except FileNotFoundError:
#     pass
# for col in col_conv:
#     label_encoder = LabelEncoder()
#     df.loc[:,col] = label_encoder.fit_transform(df.loc[:,col])
#     for i in range(len(label_encoder.classes_)):
#         with open(file, 'a') as f:
#             f.write(f'{col},{i},{label_encoder.classes_[i]}\n')
# df.to_csv('host_train_encoded.tsv', index=False,sep='\t')
# df = df.drop(columns=['case_id','patientid'])

# # one-hot encode nominal
# onehot = True
# if onehot:
#     col_1hot = ['Hospital', 'Hospital_type', 'Hospital_city', 'Hospital_region', 'Department', 'Wa
#     df = pd.get_dummies(df, columns=col_1hot, dtype=np.int64)
#     df.to_csv('host_train_encoded_onehot.tsv', index=False,sep='\t')

# # split
df = pd.read_csv('host_train_encoded_onehot.tsv', sep='\t')
X = df.drop(columns=['Stay_Days'])
y = df['Stay_Days']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42
X_train = X_train.reset_index(drop=True)
X_test.reset_index(drop=True)
y_test.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)

if TRAIN_PROP < 1:
    print(f'Subsetting to {int(TRAIN_PROP * 100)}%')
    print('Before', y_train.shape)
    y_train, _ = train_test_split(y_train, test_size=1-TRAIN_PROP, stratify=y_train, random_state=42
    print('After', y_train.shape)
    X_train = X_train.iloc[y_train.index,:]
    print('After', X_train.shape)

# scale continuous columns
file = f'lab_assignment2.train_scalers.{TRAIN_PROP}.txt'
```

```
try:
    os.remove(file)
except FileNotFoundError:
    pass
col_cont = ['Available_Extra_Rooms_in_Hospital', 'Patient_Visitors', 'Admission_Deposit']
d = {}
for col in col_cont:
    print(col)
    mu = X_train.loc[:,col].mean()
    sigma = X_train.loc[:,col].std()
    X_train.loc[:,col] = (X_train.loc[:,col] - sigma) / mu
    d[col] = [mu, sigma]
    with open(file, 'a') as f:
        f.write(f'{col},{mu},{sigma}\n')
X_train.to_csv(f'covid_X_train_{TRAIN_PROP}.tsv', index=False,sep='\t')

# scale test data
X_test.loc[:,'Available_Extra_Rooms_in_Hospital'] = \
    (X_test.loc[:,'Available_Extra_Rooms_in_Hospital'] - \
        d['Available_Extra_Rooms_in_Hospital'][0]) / \
            d['Available_Extra_Rooms_in_Hospital'][1]
X_test.loc[:,'Patient_Visitors'] = \
    (X_test.loc[:,'Patient_Visitors'] - \
        d['Patient_Visitors'][0]) / \
            d['Patient_Visitors'][1]
X_test.loc[:,'Admission_Deposit'] = \
    (X_test.loc[:,'Admission_Deposit'] - \
        d['Admission_Deposit'][0]) / \
            d['Admission_Deposit'][1]
# X_test depends on train prop since we scale based on train data
X_test.to_csv(f'covid_X_test_{TRAIN_PROP}.tsv', index=False,sep='\t')
y_train.to_csv(f'covid_y_train_{TRAIN_PROP}.tsv', index=False,sep='\t')
# y test is invariant to train prop
y_test.to_csv(f'covid_y_test.tsv', index=False,sep='\t')

### network
if DROPOUT:
    p_drop=0.8
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# build MLP with 5 hidden layers and ReLU activation
class FFN(nn.Module):
    def __init__(self, input_size, output_size):
        super(FFN, self).__init__()
        hidden_size = 64
```

```
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu1 = nn.ReLU()
        if DROPOUT:
            self.drop1 = nn.Dropout(p=p_drop)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.relu2 = nn.ReLU()
        if DROPOUT:
            self.drop2 = nn.Dropout(p=p_drop)
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        self.relu3 = nn.ReLU()
        if DROPOUT:
            self.drop3 = nn.Dropout(p=p_drop)
        self.fc4 = nn.Linear(hidden_size, hidden_size)
        self.relu4 = nn.ReLU()
        if DROPOUT:
            self.drop4 = nn.Dropout(p=p_drop)
        self.fc5 = nn.Linear(hidden_size, output_size)
        self.relu5 = nn.ReLU()
        # Apply Xavier initialization
        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                torch.nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    torch.nn.init.zeros_(m.bias)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        x = self.relu3(x)
        x = self.fc4(x)
        x = self.relu4(x)
        x = self.fc5(x)
        x = self.relu5(x)
        return x
# input_size = 15
input_size = 125
num_classes = 11

model = FFN(input_size, num_classes).to(device)
criterion = nn.CrossEntropyLoss()  # Automatically applies softmax
```

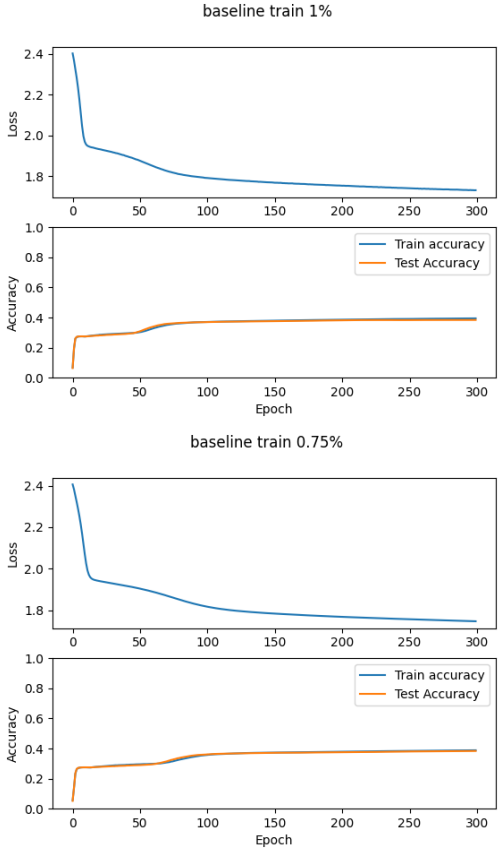
```
if MOMENTUM:
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=MOMENTUM)
else:
    optimizer = optim.SGD(model.parameters(), lr=0.001)

### minimal example
# Load data
# X_train = torch.tensor(pd.read_csv('covid_X_train.tsv', sep='\t').iloc[0,:].values.reshape(1,15),
# y_train = torch.tensor(pd.read_csv('covid_y_train.tsv', sep='\t').iloc[0,:].values, dtype=torch.lo
# print('X_train')
# print(X_train)
# print('y_train')
# print(y_train)

# # Forward pass
# print('Forward pass')
# outputs = model(X_train)  # Raw logits
# print("Logits:", outputs)  # Raw scores (not probabilities)
# loss = criterion(outputs, y_train)  # No need for softmax
# loss.backward()
# optimizer.step()
# print("Logits:", outputs)  # Raw scores (not probabilities)
# print("Predicted class:", torch.argmax(outputs, dim=1))  # Predicted class
# print("Ground truth:", y_train[0])
# sys.exit()

# load data
# train
X_train = pd.read_csv(f'covid_X_train_{TRAIN_PROP}.tsv', sep='\t')
X_train = X_train.dropna()
idx = X_train.index.values
y_train = pd.read_csv(f'covid_y_train_{TRAIN_PROP}.tsv', sep='\t')
y_train = y_train.iloc[idx,:]
X_train = torch.tensor(X_train.values, dtype=torch.float32).to(device)
y_train = torch.tensor(y_train.values, dtype=torch.long).to(device)
# test
X_test = pd.read_csv(f'covid_X_test_{TRAIN_PROP}.tsv', sep='\t')
X_test = X_test.dropna()
idx = X_test.index.values
y_test = pd.read_csv(f'covid_y_test.tsv', sep='\t')
y_test = y_test.iloc[idx,:]
X_test = torch.tensor(X_test.values, dtype=torch.float32).to(device)
y_test = torch.tensor(y_test.values, dtype=torch.long).to(device)
tensor_dataset = torch.utils.data.TensorDataset(X_train, y_train)
train_loader = DataLoader(tensor_dataset, batch_size=1024, shuffle=True)
```

```
# Start training
if LR_DECAY:
    scheduler = StepLR(optimizer, step_size=50, gamma=0.5)
epochs = [] # x axis for histry
losses = [] # loss histry
accuracies = [] # acc histry
accuracies_test = [] # test acc histry
for epoch in range(EPOCHS):
    epoch_loss = 0
    correct = 0
    correct_test = 0
    total = 0
    total_test = 0
    for i, (x, y) in enumerate(train_loader):
        if torch.isnan(x).any() or torch.isnan(y).any():
            print(f"NaN found in data at batch {i}")
        optimizer.zero_grad()  # Reset gradients
        o = model(x)  # Forward pass
        y = y.squeeze()
        loss = criterion(o, y)  # Compute loss
        loss.backward()  # Backpropagation
        optimizer.step()  # Update weights

        # Accumulate loss and accuracy
        epoch_loss += loss.item()
        yhat = torch.argmax(o, dim=1)
        correct += (yhat == y).sum().item()
        total += y.size(0)

        # test accuracy
        o = model(X_test)
        y = y_test.squeeze()
        yhat = torch.argmax(o, dim=1)
        correct_test += (yhat == y).sum().item()
        total_test += y.size(0)


    # Print stats
    epoch_loss /= len(train_loader)
    accuracy = correct / total
    accuracy_test = correct_test / total_test
    print(f"Epoch {epoch+1}/{EPOCHS}, loss: {epoch_loss:.4f}, acc: {accuracy:.4f}, test acc: {accura
    epochs.append(epoch)
    losses.append(epoch_loss)
    accuracies.append(accuracy)
    accuracies_test.append(accuracy_test)
```

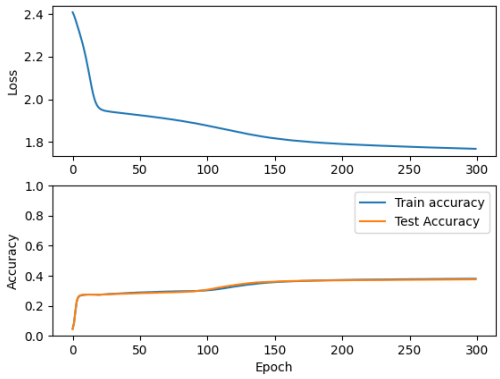
```
    # decay learning rate
    if LR_DECAY:
        scheduler.step()


# Plot loss and accuracy
fig, (ax1, ax2) = plt.subplots(2)
ax1.plot(epochs, losses, label='Loss')
ax1.set_ylabel('Loss')
ax2.plot(epochs, accuracies, label='Train accuracy')
ax2.plot(epochs, accuracies_test, label='Test Accuracy')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy')
ax2.set_ylim([0,1])
ax2.legend()
try:
    fig.suptitle(f'{NAME} train {int(TRAIN_PROP * 100)}%')
    plt.savefig(f'p_{NAME}_{TRAIN_PROP}.png')
except:
    plt.savefig(f'p_{NAME}_{TRAIN_PROP}.png')


# compute average precision and accuracy across all classes for trained model
# on test data
# o = model(X_test)
# y = y_test.squeeze()
# yhat = torch.argmax(o, dim=1)
# correct = (yhat == y).sum().item()
# total = y.size(0)
# accuracy = correct / total
# print(f"Test accuracy: {accuracy:.4f}")
# # confusion matrix
# confusion = torch.zeros(num_classes, num_classes)
# for i in range(y.size(0)):
#     confusion[y[i], yhat[i]] += 1
# print(confusion)
# # precision
# precision = torch.zeros(num_classes)
# for i in range(num_classes):
#     precision[i] = confusion[i,i] / confusion[:,i].sum()
# print(precision)
# # recall
# recall = torch.zeros(num_classes)
# for i in range(num_classes):
#     recall[i] = confusion[i,i] / confusion[i,:].sum()
# print(recall)
```

```
# # f1
# f1 = 2 * (precision * recall) / (precision + recall)
# print(f1)
# # macro
# macro = f1.mean()
# print(f"Macro: {macro:.4f}")
```

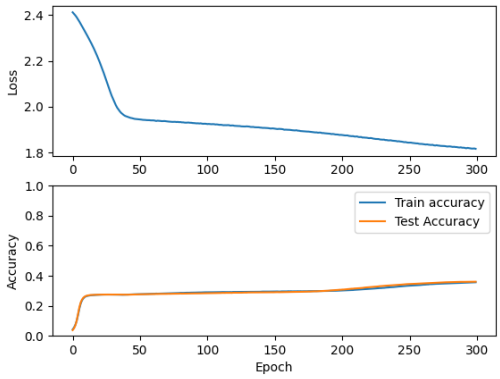
3 Appendix



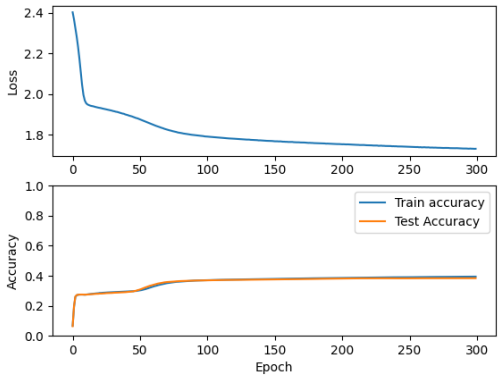
baseline train 50%



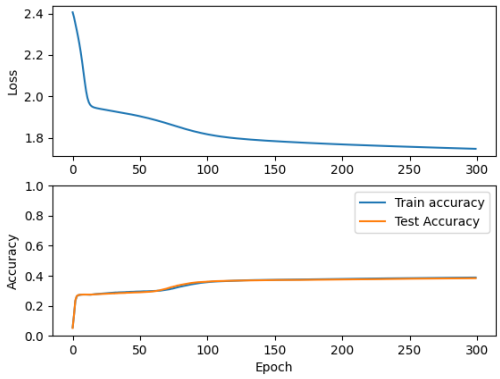
baseline train 25%



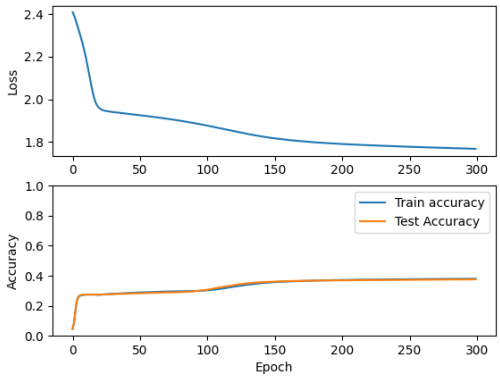
dropout train 100%



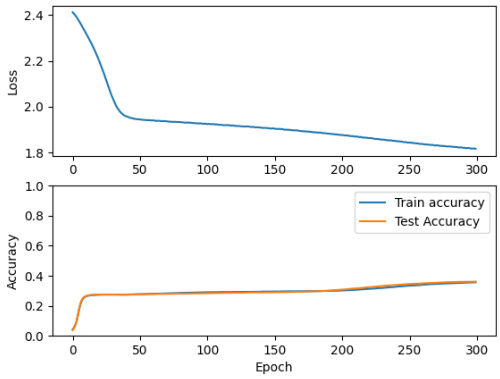
dropout train 75%



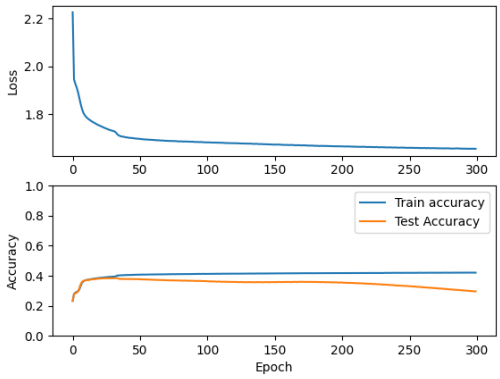
dropout train 50%



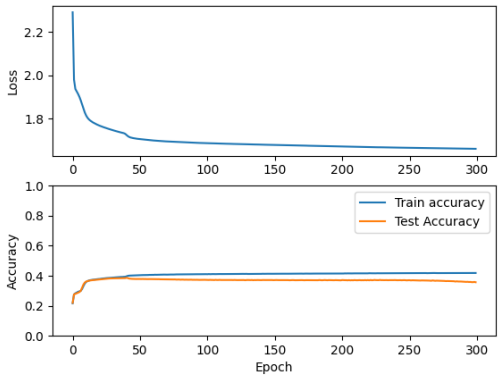
dropout train 25%



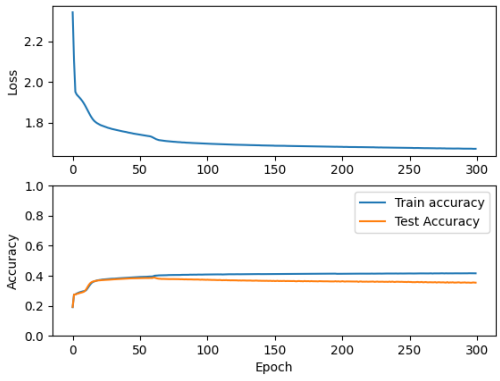
momentum train 100%



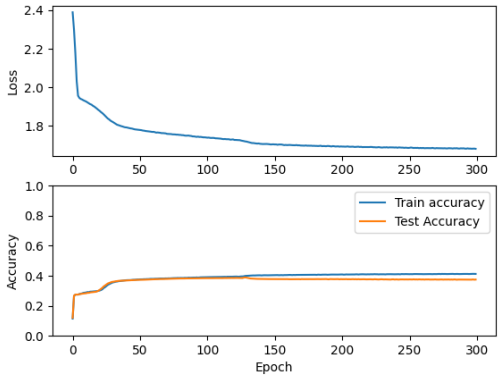
momentum train 75%



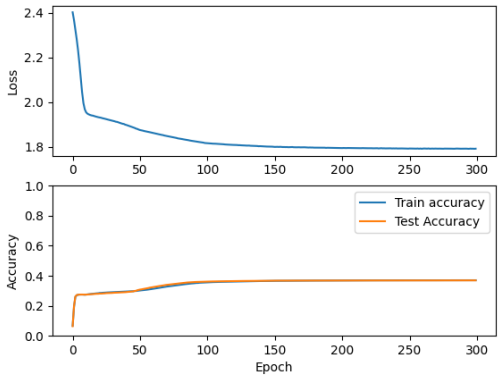
momentum train 50%



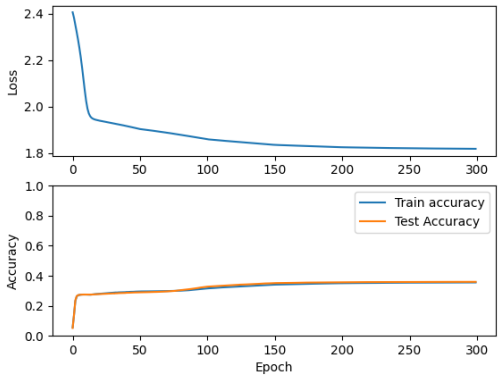
momentum train 25%



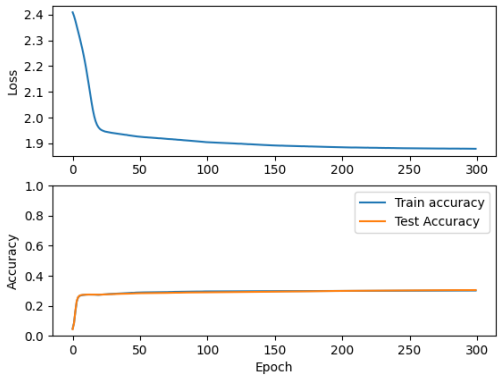
lr_decay train 100%



lr_decay train 75%



lr_decay train 50%



lr_decay train 25%