

# Question 1: Supervised Learning for Bioprinting

In this question, we continue creating machine learning algorithms which can predict cell viability from bioprinting parameters, similar to Shuyu Tian et al. "Machine assisted experimentation of extrusion-based bioprinting systems". In: *Micromachines* 12.7 (2021), p. 780. However, this time we are using neural networks.

You will train in PyTorch:

- a classification neural network to predict acceptable/unacceptable cell viability,
- a regression neural network to predict real-valued cell viability.

Preprocess the dataset according to the provided code in this thesis.

<https://scholarscompass.vcu.edu/cgi/viewcontent.cgi?article=7979&context=etd>

For classification, do not use the "Acceptable Viability?" column. Instead, label a sample as acceptable if viability  $\geq 70\%$ , and unacceptable otherwise. Split both datasets 85/15 into training and test sets.

Validation protocol: On the training set, perform 5-fold cross validation. Report the mean performance across folds and the performance when training on the full training set.

Metrics: • Regression: MSE • Classification: Accuracy, Precision, Recall

Important: Only evaluate the default model and your final best model on the test set.

1. Default models. Train regression and classification networks in PyTorch using the default optimizer settings from Tutorial 2. Report training (5-fold CV + full training) and test performance.
2. Activation functions. Test ReLU and tanh, plus one mixed-activation configuration. State which performs best.
3. Learning rate and momentum. Try two learning rates from different orders of magnitude and train with/without momentum. Plot and discuss the effect on the 5 training loss curves.
4. Best model vs default. Combine your best hyperparameters, train on the full training set, and evaluate on the test set. Compare to the default model and discuss.

## Import data

In [223...

```
import pickle
import numpy as np
import torch
```

```

import torch.nn as nn

with open("A2Q1_data.pkl", "rb") as f:
    ds_classification = pickle.load(f)

with open("A2Q1_data_regression.pkl", "rb") as f:
    ds_regression = pickle.load(f)

def to_tensors(data):
    X_train = torch.tensor(data["X_train"].astype(np.float32).values, dtype=torch.float32)
    X_test = torch.tensor(data["X_test"].astype(np.float32).values, dtype=torch.float32)
    y_train = torch.tensor(data["y_train"].values, dtype=torch.float32).unsqueeze(1)
    y_test = torch.tensor(data["y_test"].values, dtype=torch.float32).unsqueeze(1)
    return X_train, X_test, y_train, y_test

X_train_c, X_test_c, y_train_c, y_test_c = to_tensors(ds_classification)
X_train_r, X_test_r, y_train_r, y_test_r = to_tensors(ds_regression)

INPUT_DIM = X_train_c.shape[1] # 45

```

## Clasification and Regression Neural Networks

In [224...

```

from sklearn.metrics import precision_score, accuracy_score, recall_score

class ClassificationNetwork(nn.Module):
    def __init__(self, act1=nn.ReLU(), act2=nn.Tanh()): # Can choose activation function
        super().__init__()
        self.lyrs = nn.ModuleList([
            nn.Linear(45, 64),
            act1,
            nn.Linear(64, 32),
            act2,
            nn.Linear(32, 1),
        ])
        pos_weight = torch.tensor([203/97])
        self.loss_func = nn.BCEWithLogitsLoss(pos_weight=pos_weight)

    def forward(self, x):
        for lyr in self.lyrs:
            x = lyr(x)
        return x

    def learn(self, X, y, epochs=1000, lr=0.001, momentum=0.0): # Can choose momentum
        losses = []
        velocity = [torch.zeros_like(p) for p in self.parameters()]
        for epoch in range(epochs):
            y_pred = self(X)
            loss = self.loss_func(y_pred.squeeze(), y.squeeze())
            self.zero_grad()
            loss.backward()
            with torch.no_grad():
                for v, p in zip(velocity, self.parameters()):
                    v.mul_(momentum).add_(p.grad)
                    p.sub_(lr * v)
            losses.append(loss.item())

```

```

        return losses

    def evaluate(self, X, y):
        with torch.no_grad():
            logits = self(X)
            labels = (torch.sigmoid(logits).squeeze() >= 0.5).float().numpy()
            y_np = y.squeeze().numpy()
            acc = accuracy_score(y_np, labels)
            prec = precision_score(y_np, labels, zero_division=0)
            rec = recall_score(y_np, labels, zero_division=0)
            return acc, prec, rec

class RegressionNetwork(nn.Module):
    def __init__(self, act1=nn.ReLU(), act2=nn.ReLU()): # Can choose activation fun
        super().__init__()
        self.lyrs = nn.ModuleList([
            nn.Linear(45, 64),
            act1,
            nn.Linear(64, 32),
            act2,
            nn.Linear(32, 1),
        ])
        self.loss_func = nn.MSELoss()

    def forward(self, x):
        for lyr in self.lyrs:
            x = lyr(x)
        return x

    def learn(self, X, y, epochs=1000, lr=0.001, momentum=0.0): # Can choose moment
        losses = []
        velocity = [torch.zeros_like(p) for p in self.parameters()]
        for epoch in range(epochs):
            y_pred = self(X)
            loss = self.loss_func(y_pred.squeeze(), y.squeeze())
            self.zero_grad()
            loss.backward()
            with torch.no_grad():
                for v, p in zip(velocity, self.parameters()):
                    v.mul_(momentum).add_(p.grad)
                    p.sub_(lr * v)
            losses.append(loss.item())
        return losses

    def evaluate(self, X, y):
        with torch.no_grad():
            preds = self(X)
            mse = self.loss_func(preds.squeeze(), y.squeeze())
            return mse.item()

```

## CV Function

In [225...

```

from sklearn.model_selection import KFold

def cv_function(X, y, network_class, task="classification", act1=nn.ReLU(), act2=nn

```

```

        epochs=1000, lr=0.001, momentum=0.0, n_splits=5):

    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    fold_metrics = []
    fold_losses = []

    for fold, (tr_idx, val_idx) in enumerate(kf.split(X)):
        X_tr, y_tr = X[tr_idx], y[tr_idx]
        X_val, y_val = X[val_idx], y[val_idx]

        model = network_class(act1=act1, act2=act2)
        losses = model.learn(X_tr, y_tr, epochs=epochs, lr=lr, momentum=momentum)

        fold_metrics.append(model.evaluate(X_val, y_val))
        fold_losses.append(losses)

    return fold_metrics, fold_losses

```

```

In [226... def cv_results(fold_metrics, task="classification"):
    if task == "classification":
        accs = [m[0] for m in fold_metrics]
        precs = [m[1] for m in fold_metrics]
        recs = [m[2] for m in fold_metrics]
        print(f"Mean Accuracy: {np.mean(accs):.4f} ± {np.std(accs):.4f}")
        print(f"Mean Precision: {np.mean(precs):.4f} ± {np.std(precs):.4f}")
        print(f"Mean Recall: {np.mean(recs):.4f} ± {np.std(recs):.4f}")
    else:
        print(f"Mean MSE: {np.mean(fold_metrics):.4f} ± {np.std(fold_metrics):.4f}")

```

## Default Models

Classification NN 5-fold CV:

```

In [227... metrics, losses = cv_function(X_train_c, y_train_c, ClassificationNetwork)

cv_results(metrics, task="classification")

```

```

Mean Accuracy:  0.4800 ± 0.1533
Mean Precision: 0.2805 ± 0.1614
Mean Recall:    0.6191 ± 0.3498

```

Regression NN 5-fold CV:

```

In [228... metrics, losses = cv_function(X_train_r, y_train_r, RegressionNetwork)

cv_results(metrics, task="regression")

```

```

Mean MSE: 281.7732 ± 48.2961

```

Train on full training data:

```

In [256... # Classification
clf_default = ClassificationNetwork()
clf_default.learn(X_train_c, y_train_c, epochs=1000, lr=0.001)

```

```
default_acc, default_prec, default_rec = clf_default.evaluate(X_test_c, y_test_c)
print(f"Test Accuracy: {default_acc:.4f}, Precision: {default_prec:.4f}, Recall: {d
```

Test Accuracy: 0.6667, Precision: 0.0000, Recall: 0.0000

```
In [257... # Regression
reg_default = RegressionNetwork()
reg_default.learn(X_train_r, y_train_r, epochs=1000, lr=0.001)
default_mse = reg_default.evaluate(X_test_r, y_test_r)
print(f"Test MSE: {default_mse:.4f}")
```

Test MSE: 264.1830

In the default model, we see that precision and recall is 0, signalling the model is predicting everything as class 0. This motivates why we need model tuning.

## Activation Functions

Test ReLU:

```
In [231... metrics, losses = cv_function(X_train_c, y_train_c, ClassificationNetwork, act1=nn.
cv_results(metrics, task="classification")
```

Mean Accuracy: 0.6100 ± 0.1611

Mean Precision: 0.3259 ± 0.3096

Mean Recall: 0.3220 ± 0.3767

```
In [232... metrics, losses = cv_function(X_train_r, y_train_r, RegressionNetwork, act1=nn.ReLU
cv_results(metrics, task="regression")
```

Mean MSE: 298.6478 ± 48.1751

Test Tanh:

```
In [233... metrics, losses = cv_function(X_train_c, y_train_c, ClassificationNetwork, act1=nn.
cv_results(metrics, task="classification")
```

Mean Accuracy: 0.5500 ± 0.0723

Mean Precision: 0.3651 ± 0.0722

Mean Recall: 0.4389 ± 0.2067

```
In [234... metrics, losses = cv_function(X_train_r, y_train_r, RegressionNetwork, act1=nn.Tanh
cv_results(metrics, task="regression")
```

Mean MSE: 283.2494 ± 55.0205

Test mixed-activation configuration:

```
In [254... metrics, losses = cv_function(X_train_c, y_train_c, ClassificationNetwork, act1=nn.
cv_results(metrics, task="classification")
```

Mean Accuracy: 0.4533 ± 0.1118  
Mean Precision: 0.2398 ± 0.1264  
Mean Recall: 0.5346 ± 0.3101

```
In [236... metrics, losses = cv_function(X_train_r, y_train_r, RegressionNetwork, act1=nn.ReLU  
cv_results(metrics, task="regression")
```

Mean MSE: 284.4179 ± 51.3679

For the context of the experiment, we are most interested in recall to avoid missing acceptable samples. Thus, the mixed-activation configuration performed the best, with the highest precision and recall (MSE is negligible between Tanh and mixed-activation).

## Learning Rate and Momentum

```
In [237... import matplotlib.pyplot as plt  
  
lr_values = [0.1, 0.001]  
mom_values = [0.0, 0.9]
```

Classification:

```
In [238... fig, axes = plt.subplots(2, 2, figsize=(12, 8))  
fig.suptitle("Training Loss Curves: Classification")  
  
for i, lr in enumerate(lr_values):  
    for j, mom in enumerate(mom_values):  
  
        metrics, fold_losses = cv_function(X_train_c, y_train_c, ClassificationNetw  
  
        ax = axes[i][j]  
        for k, losses in enumerate(fold_losses):  
            ax.plot(losses, alpha=0.7, label=f"Fold {k+1}")  
        ax.set_title(f"lr={lr}, momentum={mom}")  
        ax.set_xlabel("Epoch")  
        ax.set_ylabel("Loss")  
        ax.legend(fontsize=7)  
  
        cv_results(metrics, task="classification")  
        print(f" lr={lr}, momentum={mom}\n")  
  
plt.tight_layout()  
plt.show()
```

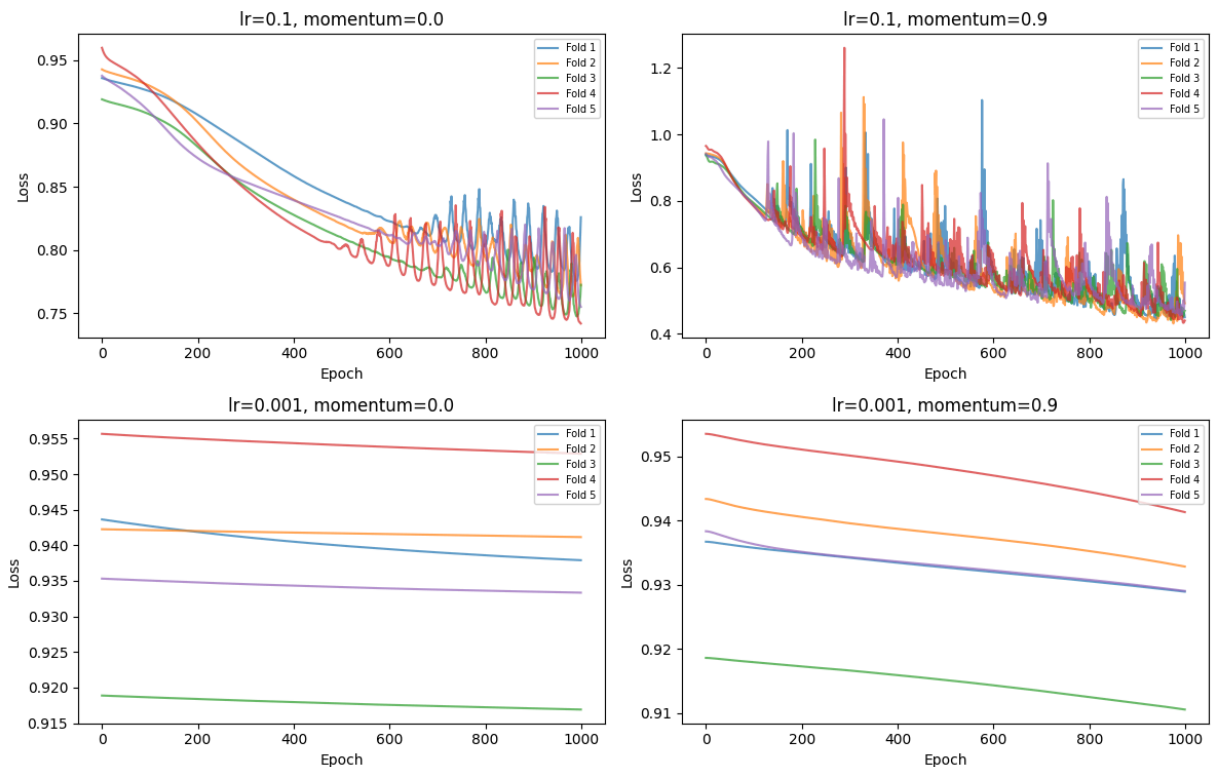
Mean Accuracy:  $0.5833 \pm 0.0279$   
Mean Precision:  $0.4172 \pm 0.1155$   
Mean Recall:  $0.5762 \pm 0.1888$   
lr=0.1, momentum=0.0

Mean Accuracy:  $0.5667 \pm 0.0211$   
Mean Precision:  $0.3542 \pm 0.0808$   
Mean Recall:  $0.4164 \pm 0.0605$   
lr=0.1, momentum=0.9

Mean Accuracy:  $0.4633 \pm 0.1536$   
Mean Precision:  $0.1680 \pm 0.1396$   
Mean Recall:  $0.4611 \pm 0.4020$   
lr=0.001, momentum=0.0

Mean Accuracy:  $0.5133 \pm 0.1185$   
Mean Precision:  $0.2536 \pm 0.1408$   
Mean Recall:  $0.4148 \pm 0.3071$   
lr=0.001, momentum=0.9

Training Loss Curves: Classification



Regression:

In [239...

```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
fig.suptitle("Training Loss Curves: Regression")

for i, lr in enumerate(lr_values):
    for j, mom in enumerate(mom_values):

        metrics, fold_losses = cv_function(X_train_r, y_train_r, RegressionNetwork,

        ax = axes[i][j]
```

```

for k, losses in enumerate(fold_losses):
    ax.plot(losses, alpha=0.7, label=f"Fold {k+1}")
ax.set_title(f"lr={lr}, momentum={mom}")
ax.set_xlabel("Epoch")
ax.set_ylabel("Loss")
ax.legend(fontsize=7)

cv_results(metrics, task="regression")
print(f" lr={lr}, momentum={mom}\n")

plt.tight_layout()
plt.show()

```

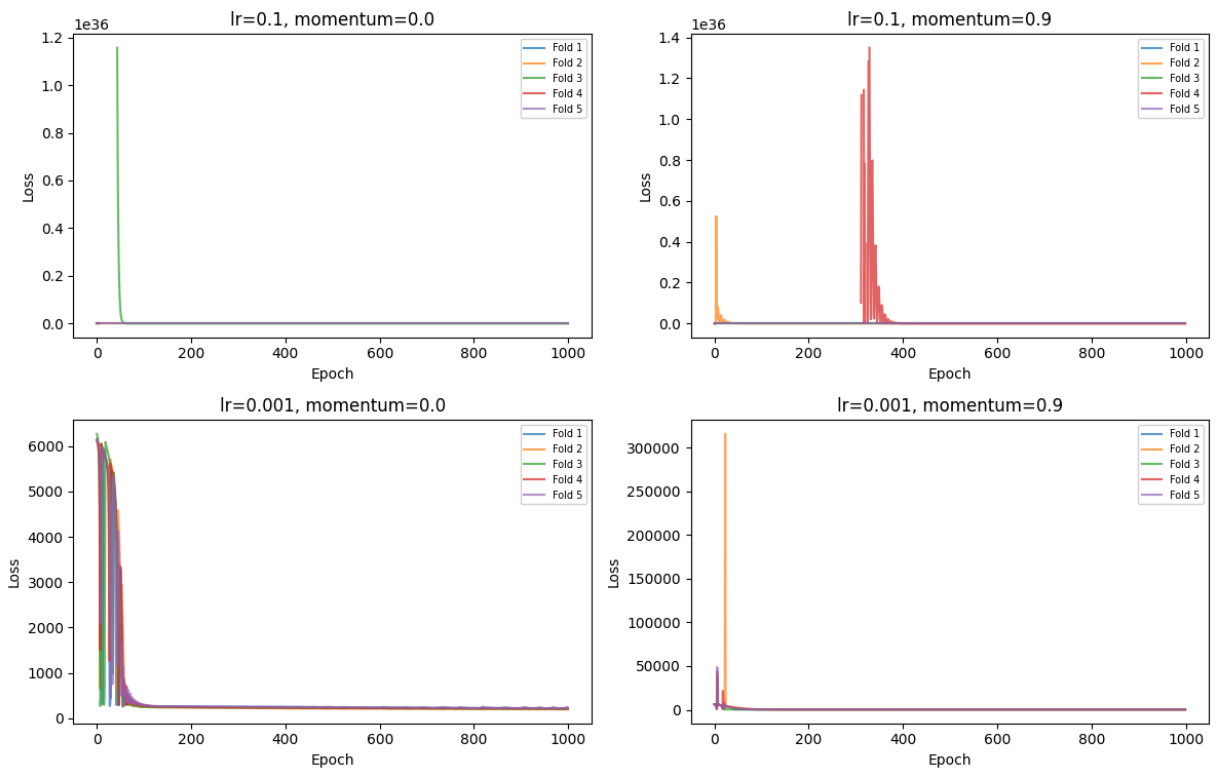
Mean MSE: nan  $\pm$  nan  
 lr=0.1, momentum=0.0

Mean MSE: 7354.5119  $\pm$  14154.5815  
 lr=0.1, momentum=0.9

Mean MSE: 293.3278  $\pm$  54.9582  
 lr=0.001, momentum=0.0

Mean MSE: 272.7555  $\pm$  51.8384  
 lr=0.001, momentum=0.9

Training Loss Curves: Regression



For classification, lr=0.1 without momentum achieved the best recall. For regression, lr=0.1 without momentum caused gradient explosion (NaN), suggesting it is too aggressive. lr=0.001 with momentum=0.9 gave the most stable and competitive results across both tasks.

## Best Model vs. Default

Best classification model:

```
In [253... clf_best = ClassificationNetwork(act1=nn.ReLU(), act2=nn.Tanh())
clf_best.learn(X_train_c, y_train_c, epochs=1000, lr=0.1, momentum=0)
acc, prec, rec = clf_best.evaluate(X_test_c, y_test_c)
print(f"Accuracy: {acc:.4f}, Precision: {prec:.4f}, Recall: {rec:.4f}")
```

Accuracy: 0.4444, Precision: 0.3235, Recall: 0.6111

Best regression model:

```
In [241... reg_best = RegressionNetwork(act1=nn.ReLU(), act2=nn.Tanh())
reg_best.learn(X_train_r, y_train_r, epochs=1000, lr=0.001, momentum=0.9)
mse = reg_best.evaluate(X_test_r, y_test_r)
print(f"MSE: {mse:.4f}")
```

MSE: 226.0764

Compare with the default model:

```
In [258... print(f"Acc: {default_acc:.4f}, Prec: {default_prec:.4f}, Rec: {default_rec:.4f}")
print(f"MSE: {default_mse:.4f}")
```

Acc: 0.6667, Prec: 0.0000, Rec: 0.0000

MSE: 264.1830

The default model predicted entirely class 0, making it useless for prediction. The best model achieves recall = 0.61 and precision = 0.32, representing a meaningful improvement despite lower raw accuracy (0.44 vs 0.67). The accuracy drop is expected since the default model was "accurate" only by predicting the majority class. For regression, there is an improvement in MSE (226.08 vs 264.18), suggesting hyperparameter tuning contributed to better regression performance.