VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

B.Tech. Fall Semester 2025-26
School Of Computer Science and Engineering
(SCOPE)

Digital Assignment - I

Deep Learning Lab

Apurva Mishra: 22BCE2791

Date: 05 August, 2025

Contents

1 Question

You have recently joined a company as a machine learning intern. Your task is to develop a simple binary emotion classifier using EEG signals recorded from two electrodes, representing Feature 1 and Feature 2.

Write a Python program to:
- Implement the perceptron learning algorithm with step-by-step weight updates.
- Visualize the errors across epochs.
- Demonstrate convergence of weights when applied to a linearly separable dataset.

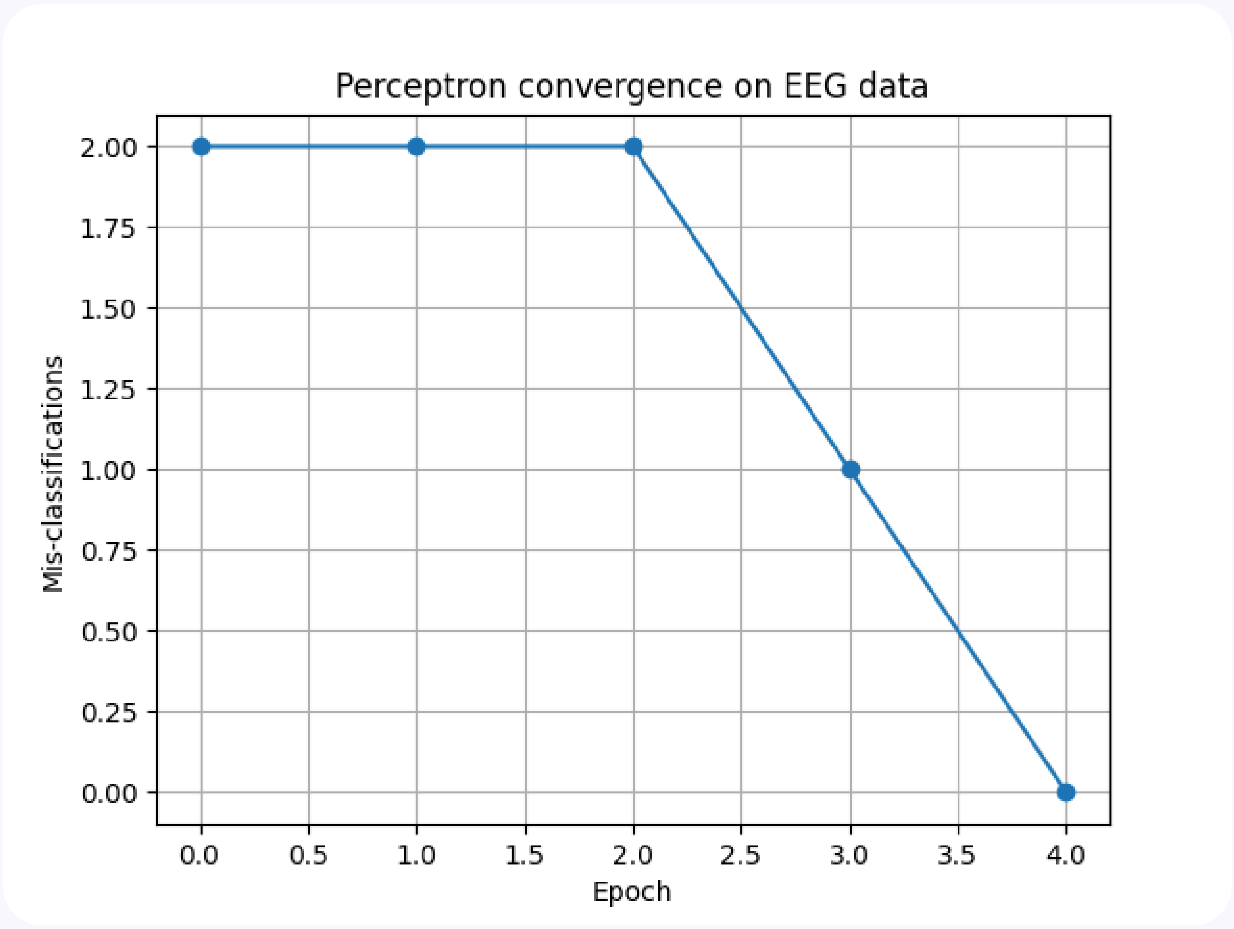
1.a Codes

main.py

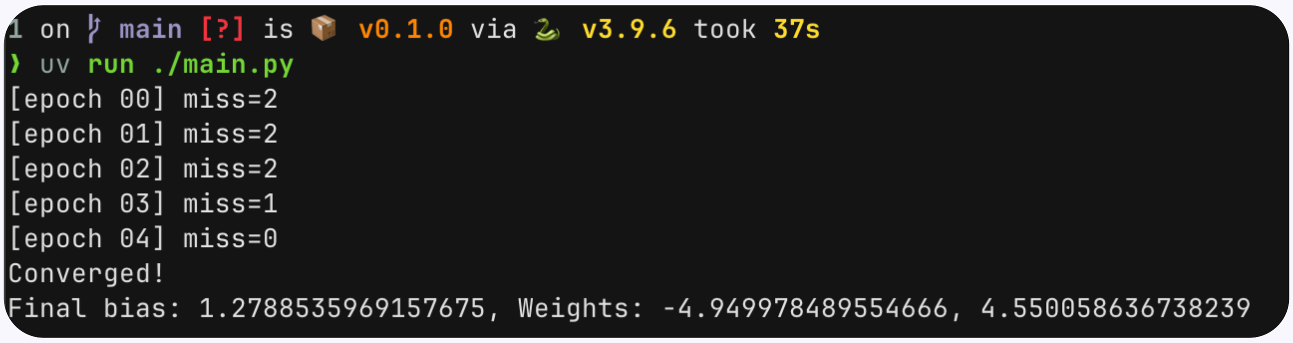
```python
import random
import matplotlib.pyplot as plt

random.seed(42)

class Perceptron:
    def __init__(self, lr=1.0):
        self.w0 = random.uniform(-1, 1)
        self.w1 = random.uniform(-1, 1)
        self.w2 = random.uniform(-1, 1)
        self.lr = lr

    def predict(self, x):
        z = self.w0 + self.w1 * x[0] + self.w2 * x[1]
        return 1 if z >= 0 else -1

    def update(self, x, target):
        pred = self.predict(x)
        if pred != target:
            self.w0 += self.lr * target * 1.0
            self.w1 += self.lr * target * x[0]
            self.w2 += self.lr * target * x[1]
            return True
        return False

x = [[2, 3], [1, 4], [2, 4], [4, 2], [5, 1], [4, 3]]
y = [+1, +1, +1, -1, -1, -1]

model = Perceptron(lr=1.0)

```

```python
max_epochs = 100
epoch_errors = []

plt.ion()
fig, ax = plt.subplots()
ax.set_xlabel('Epoch')
ax.set_ylabel('Mis-classifications')
ax.set_title('Perceptron convergence on EEG data')
ax.grid(True)
line, = ax.plot([], [], marker='o')
plt.show()

for epoch in range(max_epochs):
    miss = 0
    for xi, yi in zip(x, y):
        if model.update(xi, yi):
            miss += 1
    epoch_errors.append(miss)

    line.set_data(range(len(epoch_errors)), epoch_errors)
    ax.relim(); ax.autoscale_view()
    plt.pause(0.2)

    print(f"[epoch {epoch:02d}] miss={miss}")
    if miss == 0:
        print("Converged!")
        break

plt.ioff()
plt.show()

print("Final weights:", model.w0, model.w1, model.w2)
```

1.b Output



Visualization of errors across epochs



```
1 on ᚹ main [?] is 📦 v0.1.0 via 🐍 v3.9.6 took 37s
❯ uv run ./main.py
[epoch 00] miss=2
[epoch 01] miss=2
[epoch 02] miss=2
[epoch 03] miss=1
[epoch 04] miss=0
Converged!
Final bias: 1.2788535969157675, Weights: -4.949978489554666, 4.550058636738239
```

Convergence of Weights

2 Question

Build an MLFFNN with:
- 2 input neurons (features)
- One hidden layer (with 3–5 neurons using ReLU or tanh)
- Output layer with a sigmoid neuron

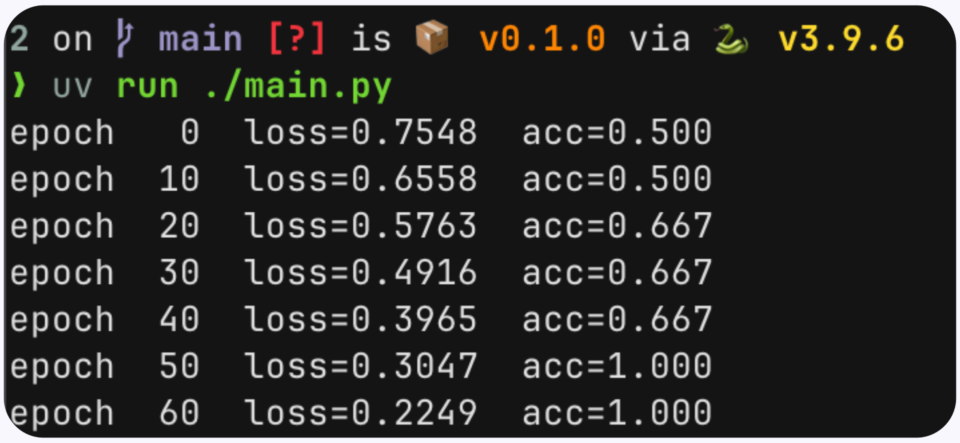
- Train using backpropagation and binary cross-entropy loss
- Track convergence metrics, prediction accuracy, and visualize the feature transformation.

2.a Codes

main.py

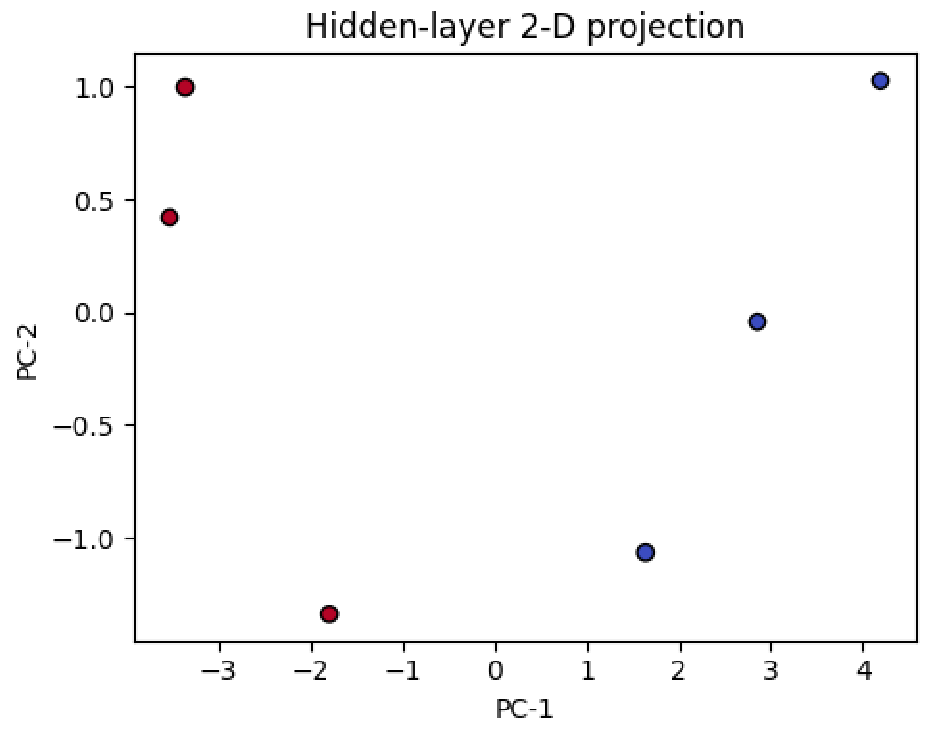
```python
import jax
import jax.numpy as jnp
import optax
import flax.linen as nn
from flax.training.train_state import TrainState
import matplotlib.pyplot as plt
import numpy as onp

X = jnp.array([[98.6, 0.2],
               [99.5, 0.9],
               [98.7, 0.8],
               [99.0, 0.3],
               [100.2, 0.7],
               [98.4, 0.6]], dtype=jnp.float32)

y = jnp.array([[0],
               [1],
               [1],
               [0],
               [1],
               [0]], dtype=jnp.float32)

mu, sigma = X.mean(0), X.std(0)
Xn = (X - mu) / sigma

class MLP(nn.Module):
    hidden: int = 4
    @nn.compact
    def __call__(self, x):
        x = nn.Dense(self.hidden)(x)
        x = nn.relu(x)
        return nn.Dense(1)(x)

model = MLP()
rng = jax.random.key(0)
params = model.init(rng, jnp.ones((1, 2)))["params"]

@jax.jit
def loss_fn(params, x, y):
    logits = model.apply({"params": params}, x)
    return optax.sigmoid_binary_cross_entropy(logits, y).mean()

@jax.jit
def accuracy(params, x, y):
    logits = model.apply({"params": params}, x)
    return ((logits > 0.5) == y).mean()

optimizer = optax.adam(1e-2)
state = TrainState.create(
    apply_fn=model.apply,
    params=params,
    tx=optimizer
)

@jax.jit
def train_step(state, x, y):
    grads = jax.grad(loss_fn)(state.params, x, y)
    return state.apply_gradients(grads=grads)

epochs = 500
loss_hist, acc_hist = [], []

for epoch in range(epochs):
    state = train_step(state, Xn, y)
    if epoch % 10 == 0 or epoch == epochs - 1:
        loss = loss_fn(state.params, Xn, y)
        acc  = accuracy(state.params, Xn, y)
        loss_hist.append(float(loss))
        acc_hist.append(float(acc))
        print(f"epoch {epoch:3d}  loss={loss:.4f}  acc={acc:.3f}")

plt.figure(figsize=(6,4))
h = 200
xx, yy = onp.meshgrid(onp.linspace(X[:,0].min()-1, X[:,0].max()+1, h),
                      onp.linspace(X[:,1].min()-1, X[:,1].max()+1, h))
grid = jnp.c_[xx.ravel(), yy.ravel()]
grid_n = (grid - mu) / sigma
zz = nn.sigmoid(model.apply({"params": state.params}, grid_n)).reshape(xx.shape)

plt.contourf(xx, yy, zz, levels=[0, 0.5, 1], cmap="coolwarm", alpha=.3)
plt.scatter(X[:,0], X[:,1], c=y.ravel(), cmap="coolwarm",
edgecolor="k")
plt.xlabel("Temperature (°F)")
plt.ylabel("Inflammation")
plt.title("Learned decision boundary")
plt.colorbar(label="Flu prob")
plt.tight_layout()

@jax.jit
def hidden_rep(params, x):
    x = nn.Dense(4).apply({"params": params["Dense_0"]}, x)
    return nn.relu(x)

H = hidden_rep(state.params, Xn)          # (6, 4)

H = H - H.mean(0)
cov = jnp.cov(H.T)
w, V = jnp.linalg.eigh(cov)
idx = jnp.argsort(w)[::-1][:2]
proj = H @ V[:, idx]

plt.figure(figsize=(5,4))
plt.scatter(proj[:,0], proj[:,1], c=y.ravel(), cmap="coolwarm",
edgecolor="k")
plt.title("Hidden-layer 2-D projection")
plt.xlabel("PC-1"); plt.ylabel("PC-2")
plt.tight_layout()
plt.show()
```

2.b Output

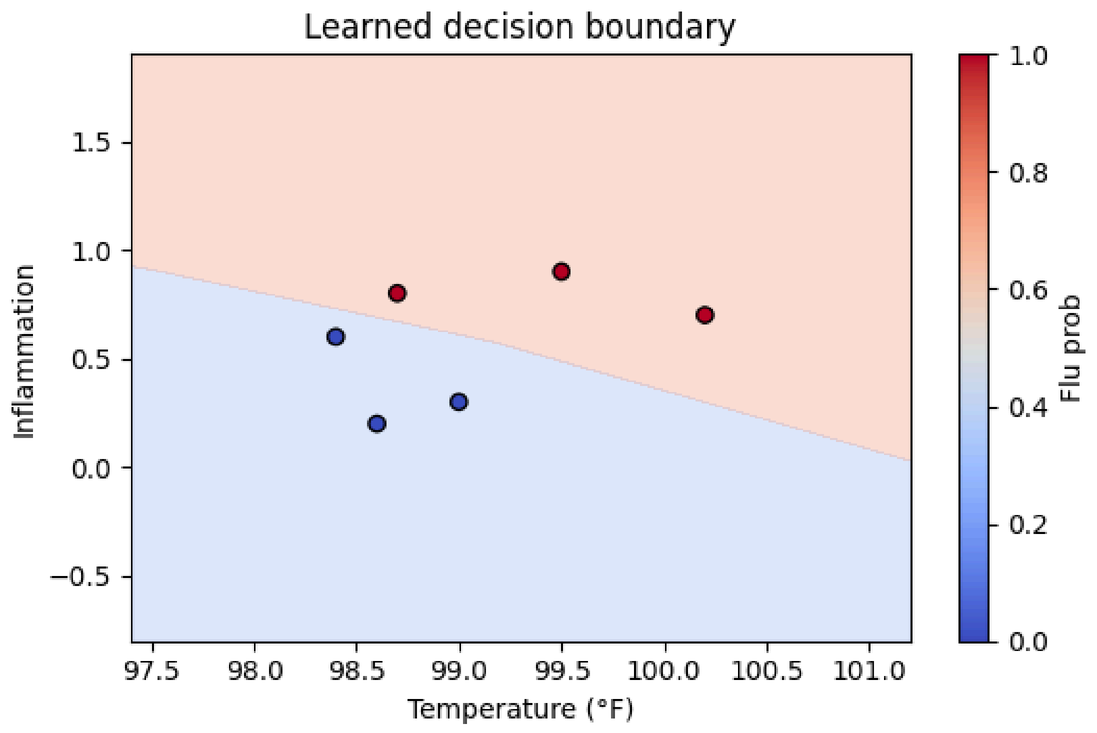


```
2 on ᚴ main [?] is 📦 v0.1.0 via 🐍 v3.9.6
❯ uv run ./main.py
epoch    0  loss=0.7548  acc=0.500
epoch   10  loss=0.6558  acc=0.500
epoch   20  loss=0.5763  acc=0.667
epoch   30  loss=0.4916  acc=0.667
epoch   40  loss=0.3965  acc=0.667
epoch   50  loss=0.3047  acc=1.000
epoch   60  loss=0.2249  acc=1.000
```

Convergence of Weights



2-D Projections



Learned Decision Boundary

3 Question

You'll train a basic feedforward neural network to approximate the function: $y = 3x^2 + 2x + 1$

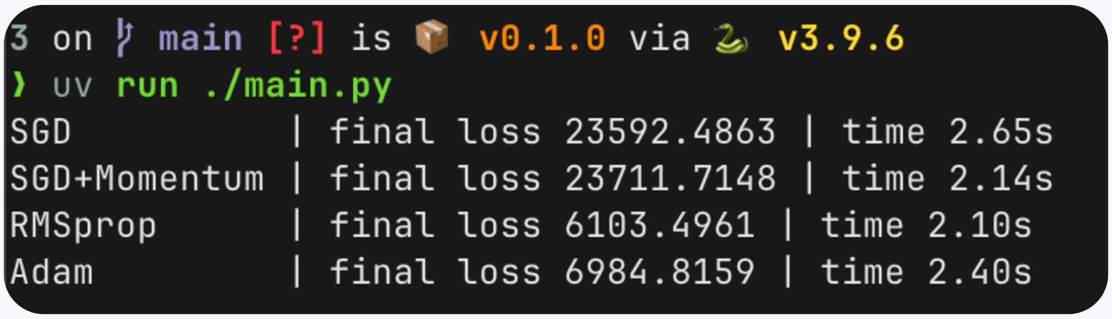
3.a Codes

main.py

```python
import time
import jax
import jax.numpy as jnp
import optax
import flax.linen as nn
from flax.training.train_state import TrainState
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
X_np = np.linspace(-10, 10, 1000).reshape(-1,
1).astype(np.float32)
y_np = (3*X_np**2 + 2*X_np + 1 +
        np.random.normal(0, 10, X_np.shape)).astype(np.float32)

X, y = jnp.array(X_np), jnp.array(y_np)

split = int(0.8*len(X))
X_train, y_train = X[:split], y[:split]
X_test,  y_test  = X[split:], y[split:]

class MLP(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Dense(16)(x); x = nn.relu(x)
        x = nn.Dense(8)(x);  x = nn.relu(x)
        return nn.Dense(1)(x)          # linear output

model = MLP()
rng   = jax.random.key(0)
init  = lambda: model.init(rng, jnp.ones((1,1)))["params"]

opts = {
    "SGD"            : optax.sgd(1e-3),
    "SGD+Momentum"   : optax.sgd(1e-3, momentum=0.9),
    "RMSprop"        : optax.rmsprop(1e-3),
    "Adam"           : optax.adam(1e-3),
}

```

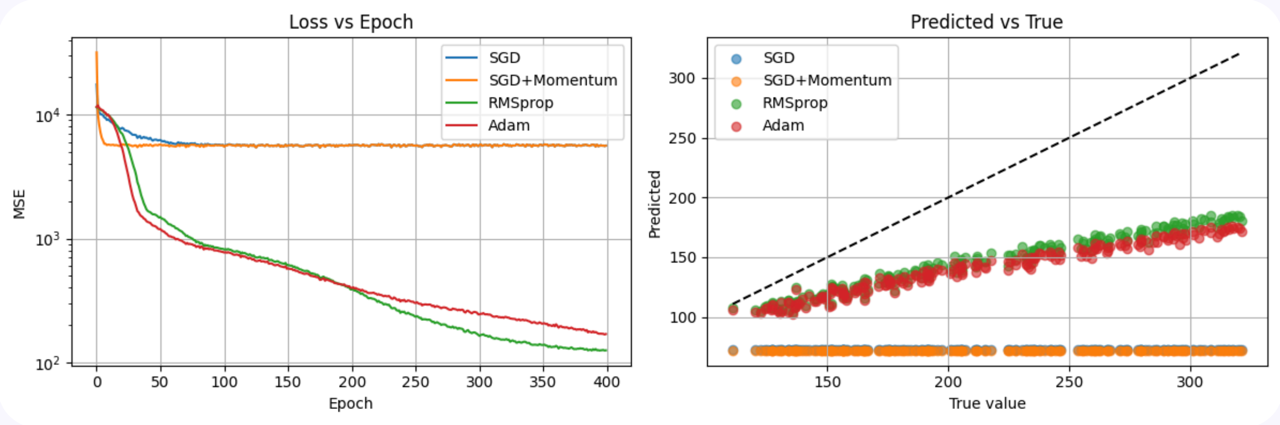
```python
EPOCHS     = 400
BATCH_SIZE = 64
STEPS      = len(X_train) // BATCH_SIZE

@jax.jit
def mse(params, x, y):
    return ((model.apply({"params": params}, x) - y) ** 2).mean()

@jax.jit
def train_step(state, xb, yb):
    loss, grads = jax.value_and_grad(mse)(state.params, xb, yb)
    return state.apply_gradients(grads=grads), loss

def run(name, opt):
    params = init()
    state  = TrainState.create(apply_fn=model.apply,
                               params=params,
                               tx=opt)

    losses = []
    start  = time.perf_counter()
    for epoch in range(EPOCHS):
        perm = jax.random.permutation(jax.random.key(epoch), len(X_train))
        Xb, yb = X_train[perm], y_train[perm]
        epoch_loss = 0.0
        for step in range(STEPS):
            b = slice(step*BATCH_SIZE, (step+1)*BATCH_SIZE)
            state, loss = train_step(state, Xb[b], yb[b])
            epoch_loss += loss
        losses.append(float(epoch_loss/STEPS))

    wall = time.perf_counter() - start
    final = mse(state.params, X_test, y_test)
    print(f"{name:<12} | final loss {final:.4f} | time {wall:.2f}s")
    return state.params, losses

hist, preds = {}, {}
for name, opt in opts.items():
    params, losses = run(name, opt)
    hist[name]  = losses
    preds[name] = model.apply({"params": params}, X_test)

plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
for name, ls in hist.items():
    plt.plot(ls, label=name)
plt.xlabel("Epoch"); plt.ylabel("MSE"); plt.title("Loss vs Epoch")
plt.yscale("log"); plt.legend(); plt.grid()

plt.subplot(1,2,2)
for name, yhat in preds.items():
    plt.scatter(y_test, yhat, alpha=0.6, label=name)
lims = [y_test.min(), y_test.max()]
plt.plot(lims, lims, 'k--')
plt.xlabel("True value"); plt.ylabel("Predicted"); plt.title("Predicted vs True")
plt.legend(); plt.grid()

plt.tight_layout()
plt.show()
```

3.b Output



```
3 on  main [?] is 📦 v0.1.0 via 🐍 v3.9.6
❯ uv run ./main.py
SGD          | final loss 23592.4863 | time 2.65s
SGD+Momentum | final loss 23711.7148 | time 2.14s
RMSprop      | final loss 6103.4961 | time 2.10s
Adam         | final loss 6984.8159 | time 2.40s
```

Training Data



Data Plots

4 Question

You're working with a financial tech company building an automated check-processing system. One key task is to read handwritten numeric account codes from scanned checks. The system must be both highly accurate and computationally efficient. To validate different neural network strategies, you're required to implement a digit classification system using the MNIST dataset (digits 0–9, 10 classes) using two approaches:

- A deep learning framework (e.g., PyTorch, TensorFlow) and use CrossEntropyLoss and

Adam optimizer

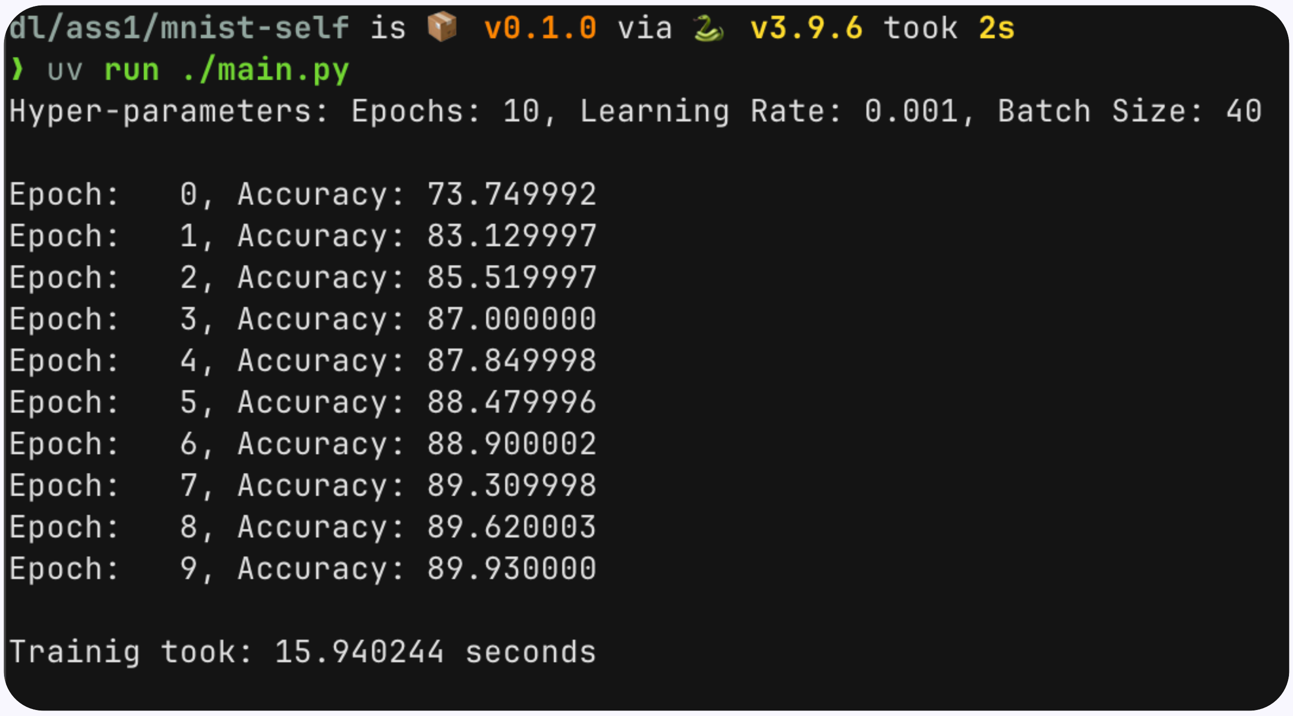
- Manual implementation using NumPy without any deep learning framework

Train a simple feedforward ANN on the MNIST dataset (10 classes) using both approaches and compare their behavior.

4.a Codes

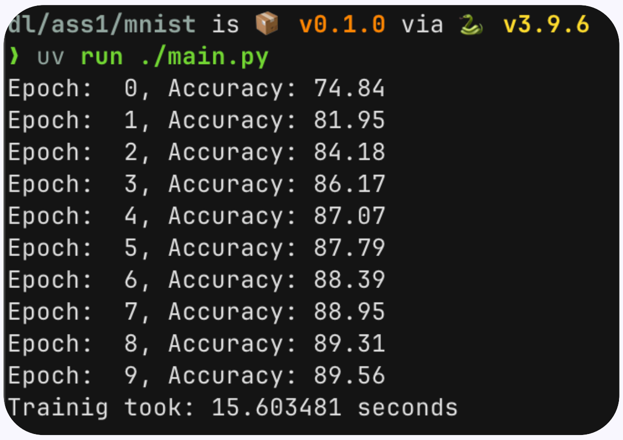
Github: Mnist hand coded without dl framework
Github: Mnist coded with jax framework

4.b Output



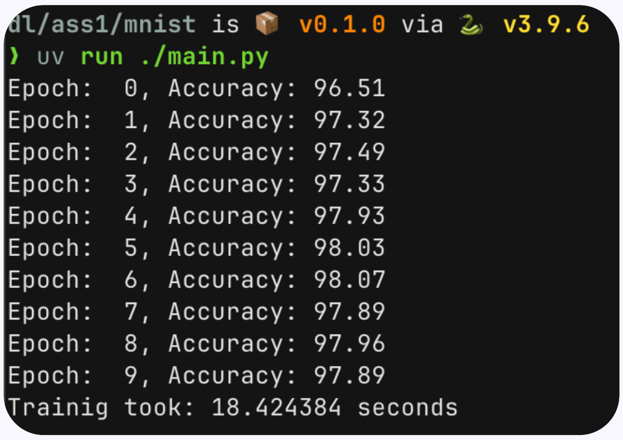
```
dl/ass1/mnist-self is 📦 v0.1.0 via 🐍 v3.9.6 took 2s
❯ uv run ./main.py
Hyper-parameters: Epochs: 10, Learning Rate: 0.001, Batch Size: 40

Epoch:   0, Accuracy: 73.749992
Epoch:   1, Accuracy: 83.129997
Epoch:   2, Accuracy: 85.519997
Epoch:   3, Accuracy: 87.000000
Epoch:   4, Accuracy: 87.849998
Epoch:   5, Accuracy: 88.479996
Epoch:   6, Accuracy: 88.900002
Epoch:   7, Accuracy: 89.309998
Epoch:   8, Accuracy: 89.620003
Epoch:   9, Accuracy: 89.930000

Trainig took: 15.940244 seconds
```

Mnist trained without any framework



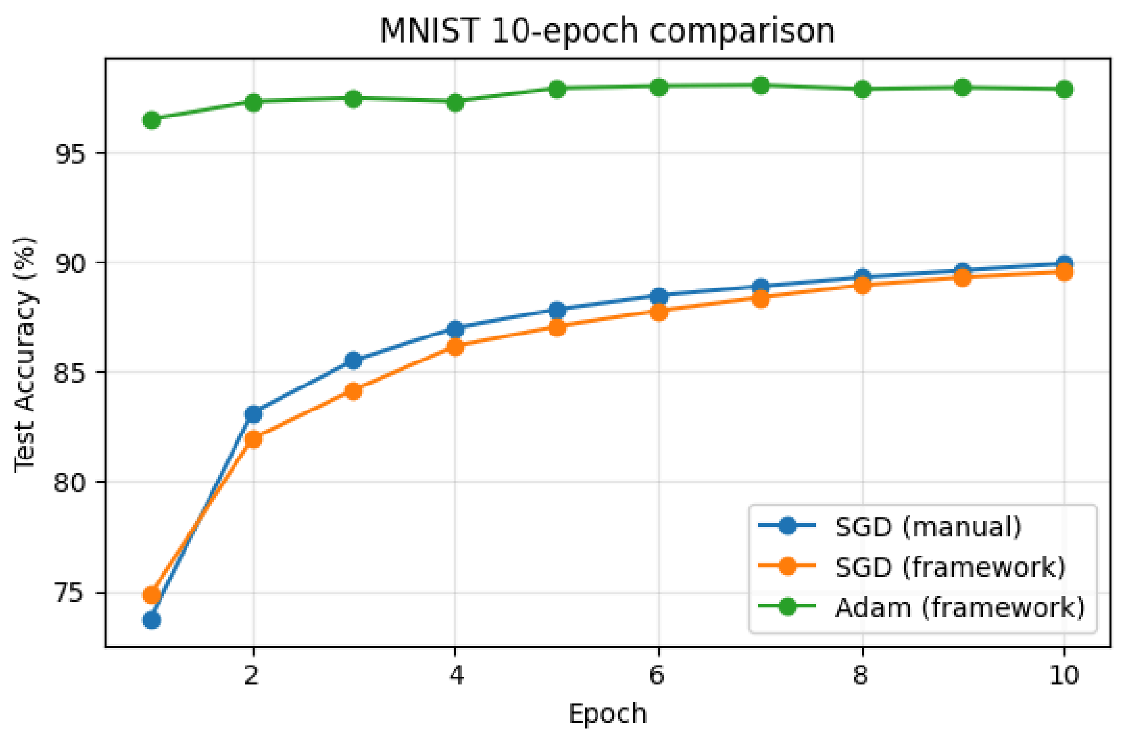
```
dl/ass1/mnist is 📦 v0.1.0 via 🐍 v3.9.6
❯ uv run ./main.py
Epoch:  0, Accuracy: 74.84
Epoch:  1, Accuracy: 81.95
Epoch:  2, Accuracy: 84.18
Epoch:  3, Accuracy: 86.17
Epoch:  4, Accuracy: 87.07
Epoch:  5, Accuracy: 87.79
Epoch:  6, Accuracy: 88.39
Epoch:  7, Accuracy: 88.95
Epoch:  8, Accuracy: 89.31
Epoch:  9, Accuracy: 89.56
Trainig took: 15.603481 seconds
```

Mnist trained in jax with sgd



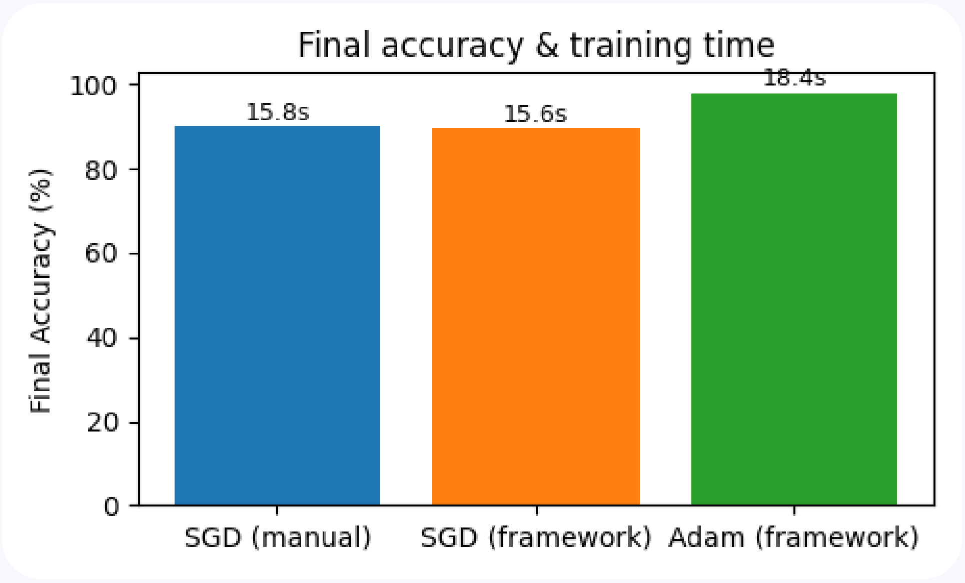
```
dl/ass1/mnist is 📦 v0.1.0 via 🐍 v3.9.6
❯ uv run ./main.py
Epoch:  0, Accuracy: 96.51
Epoch:  1, Accuracy: 97.32
Epoch:  2, Accuracy: 97.49
Epoch:  3, Accuracy: 97.33
Epoch:  4, Accuracy: 97.93
Epoch:  5, Accuracy: 98.03
Epoch:  6, Accuracy: 98.07
Epoch:  7, Accuracy: 97.89
Epoch:  8, Accuracy: 97.96
Epoch:  9, Accuracy: 97.89
Trainig took: 18.424384 seconds
```

Mnist trained in jax with adam



Accuracy

,



Trainnig Time

,

Observations

1. I trained 3 models
   A. Mnist without any framework only using jax.numpy which is a python package for working with nd-arrays using **sgd** optimizer.
   B. Mnist trained using jax dl framework using **sgd** optimizer
   C. Mnist trained using jax dl framework using **adam** optimizer
2. The adam version performed significantly better, quickly converging at 97.5% accuracy.
3. However, using adam cost around 20% extra training time.
4. As control I also trained the same code with sgd to compare against my hand coded implementation without using framework. Both of then showed similar convergence to 89.5% accuracy.
5. The sgd implemented using the dl framework was marginally faster to train however it performed marginally worse than the hand coded version.