



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	2
1.a Codes	2
1.b Output	4
2 Question	4
2.a Codes	5
2.b Output	7
3 Question	9
3.a Codes	9
3.b Output	11
4 Question	12
4.a Codes	12
4.b Output	12

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

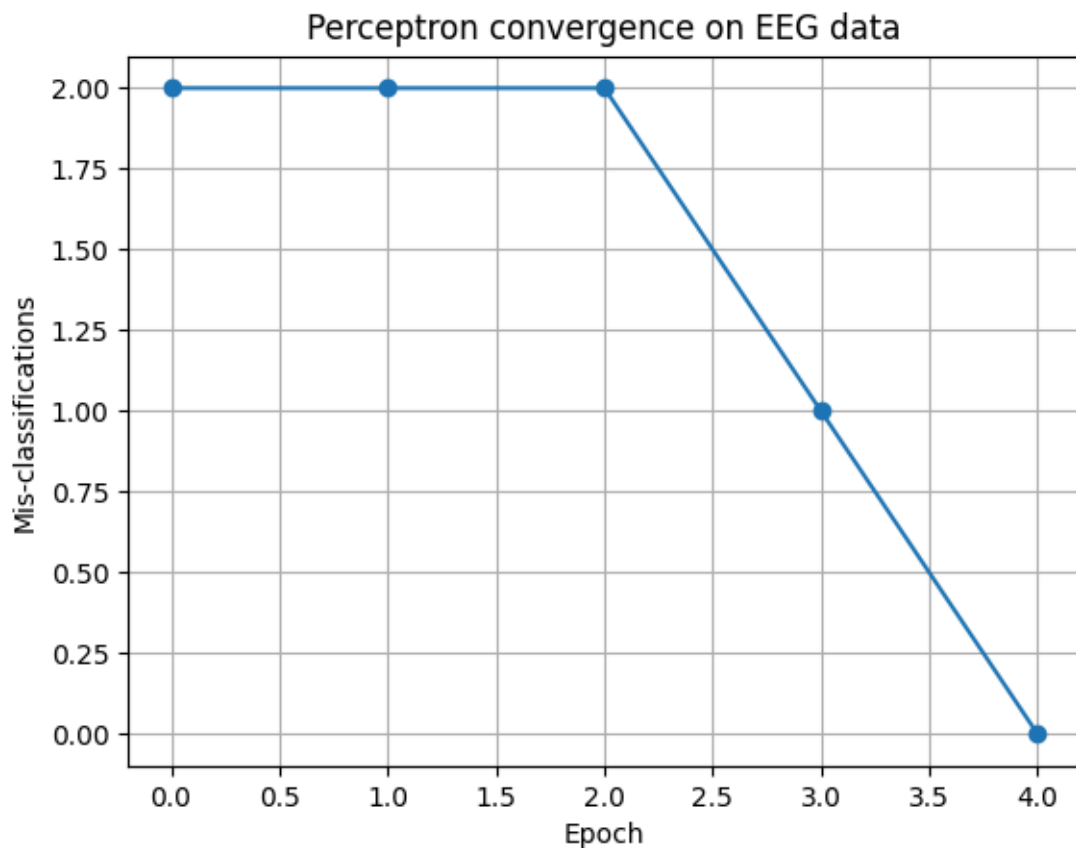
```
1 import random
2 import matplotlib.pyplot as plt
3
4 random.seed(42)
5
6 class Perceptron:
7     def __init__(self, lr=1.0):
8         self.w0 = random.uniform(-1, 1)
9         self.w1 = random.uniform(-1, 1)
10        self.w2 = random.uniform(-1, 1)
11        self.lr = lr
12
13    def predict(self, x):
14        z = self.w0 + self.w1 * x[0] + self.w2 * x[1]
15        return 1 if z ≥ 0 else -1
16
17    def update(self, x, target):
18        pred = self.predict(x)
19        if pred ≠ target:
20            self.w0 += self.lr * target * 1.0
21            self.w1 += self.lr * target * x[0]
22            self.w2 += self.lr * target * x[1]
23            return True
24        return False
25
26 x = [[2, 3], [1, 4], [2, 4], [4, 2], [5, 1], [4, 3]]
27 y = [+1, +1, +1, -1, -1, -1]
28
29 model = Perceptron(lr=1.0)
30
```

```

31 max_epochs = 100
32 epoch_errors = []
33
34 plt.ion()
35 fig, ax = plt.subplots()
36 ax.set_xlabel('Epoch')
37 ax.set_ylabel('Mis-classifications')
38 ax.set_title('Perceptron convergence on EEG data')
39 ax.grid(True)
40 line, = ax.plot([], [], marker='o')
41 plt.show()
42
43 for epoch in range(max_epochs):
44     miss = 0
45     for xi, yi in zip(x, y):
46         if model.update(xi, yi):
47             miss += 1
48     epoch_errors.append(miss)
49
50     line.set_data(range(len(epoch_errors)), epoch_errors)
51     ax.relim(); ax.autoscale_view()
52     plt.pause(0.2)
53
54     print(f"[epoch {epoch:02d}] miss={miss}")
55     if miss == 0:
56         print("Converged!")
57         break
58
59 plt.ioff()
60 plt.show()
61
62 print("Final weights:", model.w0, model.w1, model.w2)
63

```

1.b Output



Visualization of errors across epochs

```
1 on j 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

```

1  import jax
2  import jax.numpy as jnp
3  import optax
4  import flax.linen as nn
5  from flax.training.train_state import TrainState
6  import matplotlib.pyplot as plt
7  import numpy as onp
8
9  X = jnp.array([[98.6, 0.2],
10                [99.5, 0.9],
11                [98.7, 0.8],
12                [99.0, 0.3],
13                [100.2, 0.7],
14                [98.4, 0.6]], dtype=jnp.float32)
15
16  y = jnp.array([[0],
17                [1],
18                [1],
19                [0],
20                [1],
21                [0]], dtype=jnp.float32)
22
23  mu, sigma = X.mean(0), X.std(0)
24  Xn = (X - mu) / sigma
25
26  class MLP(nn.Module):
27      hidden: int = 4
28      @nn.compact
29      def __call__(self, x):
30          x = nn.Dense(self.hidden)(x)
31          x = nn.relu(x)
32          return nn.Dense(1)(x)
33
34  model = MLP()
35  rng = jax.random.key(0)
36  params = model.init(rng, jnp.ones((1, 2)))["params"]
37
38  @jax.jit
39  def loss_fn(params, x, y):

```

```

40     logits = model.apply({"params": params}, x)
41     return optax.sigmoid_binary_cross_entropy(logits, y).mean()
42
43 @jax.jit
44 def accuracy(params, x, y):
45     logits = model.apply({"params": params}, x)
46     return ((logits > 0.5) == y).mean()
47
48 optimizer = optax.adam(1e-2)
49 state = TrainState.create(
50     apply_fn=model.apply,
51     params=params,
52     tx=optimizer
53 )
54
55 @jax.jit
56 def train_step(state, x, y):
57     grads = jax.grad(loss_fn)(state.params, x, y)
58     return state.apply_gradients(grads=grads)
59
60 epochs = 500
61 loss_hist, acc_hist = [], []
62
63 for epoch in range(epochs):
64     state = train_step(state, Xn, y)
65     if epoch % 10 == 0 or epoch == epochs - 1:
66         loss = loss_fn(state.params, Xn, y)
67         acc = accuracy(state.params, Xn, y)
68         loss_hist.append(float(loss))
69         acc_hist.append(float(acc))
70         print(f"epoch {epoch:3d}  loss={loss:.4f}  acc={acc:.3f}")
71
72 plt.figure(figsize=(6,4))
73 h = 200
74 xx, yy = onp.meshgrid(onp.linspace(X[:,0].min()-1, X[:,0].max()+1,
75                                onp.linspace(X[:,1].min()-1, X[:,1].max()+1,
76                                h)),
77                                h))
78 grid = jnp.c_[xx.ravel(), yy.ravel()]
79 grid_n = (grid - mu) / sigma
80 zz = nn.sigmoid(model.apply({"params": state.params},
81 grid_n)).reshape(xx.shape)
82 plt.contourf(xx, yy, zz, levels=[0, 0.5, 1], cmap="coolwarm",
83 alpha=.3)
84 plt.scatter(X[:,0], X[:,1], c=y.ravel(), cmap="coolwarm",

```

```

edgecolor="k")
82 plt.xlabel("Temperature (°F)")
83 plt.ylabel("Inflammation")
84 plt.title("Learned decision boundary")
85 plt.colorbar(label="Flu prob")
86 plt.tight_layout()
87
88 @jax.jit
89 def hidden_rep(params, x):
90     x = nn.Dense(4).apply({"params": params["Dense_0"]}, x)
91     return nn.relu(x)
92
93 H = hidden_rep(state.params, Xn)          # (6, 4)
94
95 H = H - H.mean(0)
96 cov = jnp.cov(H.T)
97 w, V = jnp.linalg.eigh(cov)
98 idx = jnp.argsort(w)[::-1][:2]
99 proj = H @ V[:, idx]
100
101 plt.figure(figsize=(5,4))
102 plt.scatter(proj[:,0], proj[:,1], c=y.ravel(), cmap="coolwarm",
103             edgecolor="k")
104 plt.title("Hidden-layer 2-D projection")
105 plt.xlabel("PC-1"); plt.ylabel("PC-2")
106 plt.tight_layout()
107 plt.show()

```

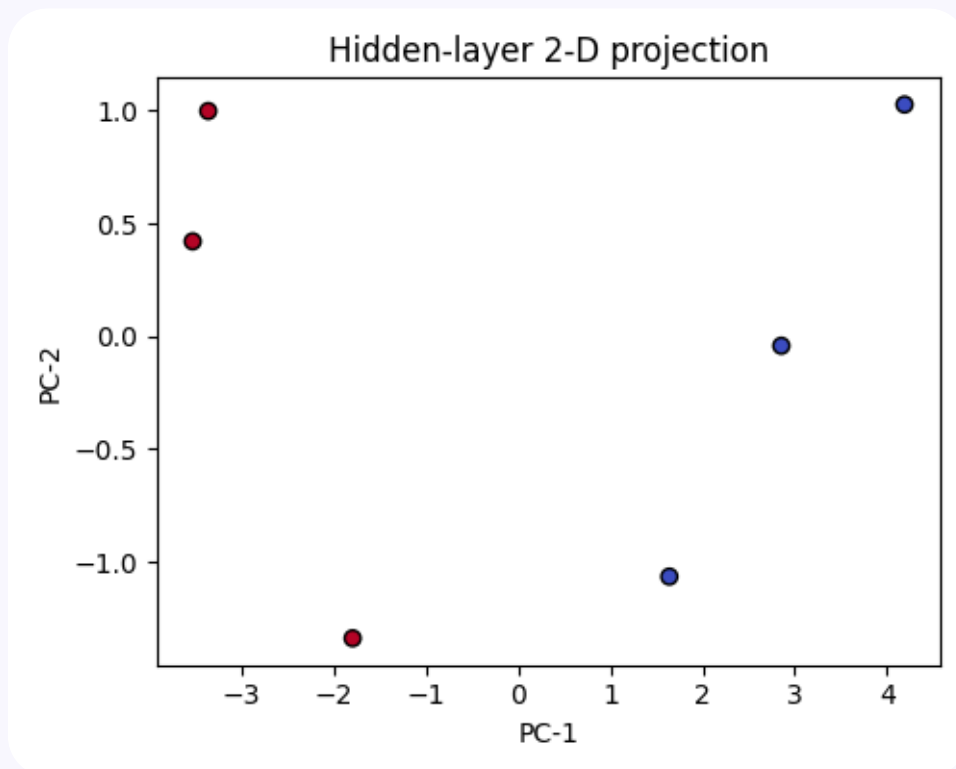
2.b Output

```

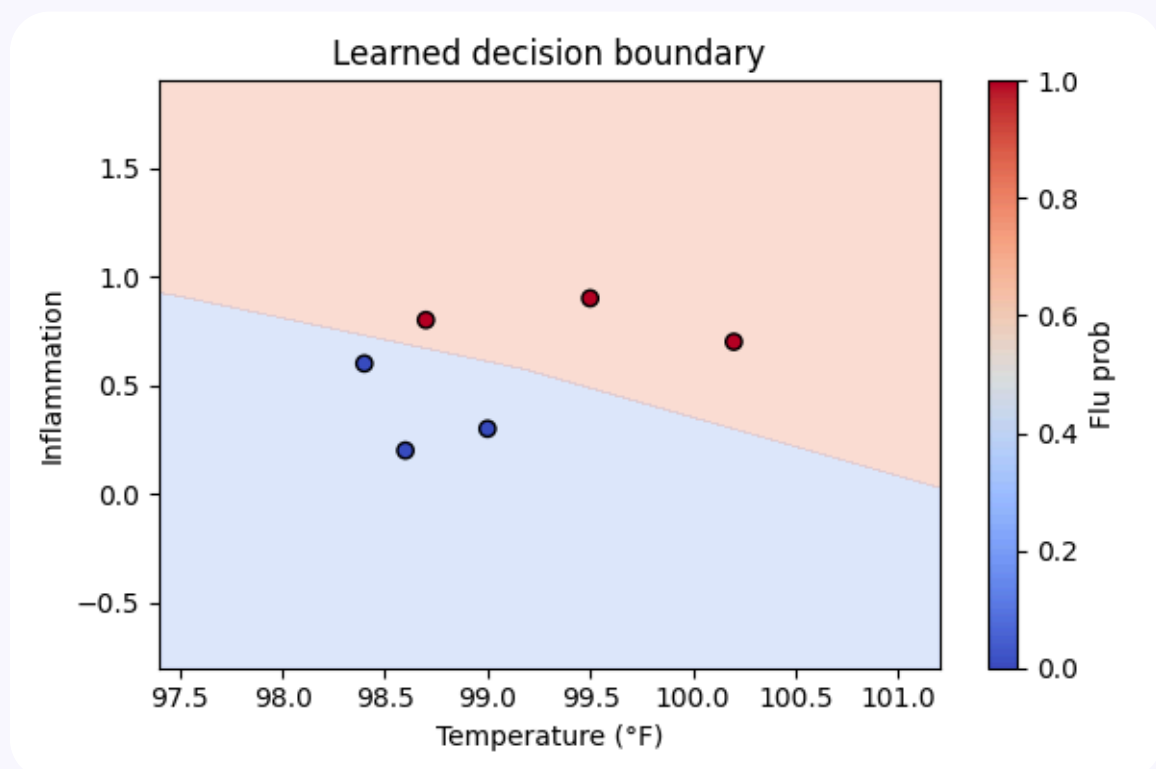
2 on jupyter main [?] is v0.1.0 via v3.9.6
> uv run ./main.py
epoch    0  loss=0.7548  acc=0.500
epoch   10  loss=0.6558  acc=0.500
epoch   20  loss=0.5763  acc=0.667
epoch   30  loss=0.4916  acc=0.667
epoch   40  loss=0.3965  acc=0.667
epoch   50  loss=0.3047  acc=1.000
epoch   60  loss=0.2249  acc=1.000

```

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

```
1 import time
2 import jax
3 import jax.numpy as jnp
4 import optax
5 import flax.linen as nn
6 from flax.training.train_state import TrainState
7 import numpy as np
8 import matplotlib.pyplot as plt
9
10 np.random.seed(0)
11 X_np = np.linspace(-10, 10, 1000).reshape(-1,
12 1).astype(np.float32)
13 y_np = (3*X_np**2 + 2*X_np + 1 +
14         np.random.normal(0, 10, X_np.shape)).astype(np.float32)
15 X, y = jnp.array(X_np), jnp.array(y_np)
16
17 split = int(0.8*len(X))
18 X_train, y_train = X[:split], y[:split]
19 X_test, y_test = X[split:], y[split:]
20
21 class MLP(nn.Module):
22     @nn.compact
23     def __call__(self, x):
24         x = nn.Dense(16)(x); x = nn.relu(x)
25         x = nn.Dense(8)(x); x = nn.relu(x)
26         return nn.Dense(1)(x) # linear output
27
28 model = MLP()
29 rng = jax.random.key(0)
30 init = lambda: model.init(rng, jnp.ones((1,1)))["params"]
31
32 opts = {
33     "SGD" : optax.sgd(1e-3),
34     "SGD+Momentum" : optax.sgd(1e-3, momentum=0.9),
35     "RMSprop" : optax.rmsprop(1e-3),
36     "Adam" : optax.adam(1e-3),
37 }
38
```

```

39 EPOCHS      = 400
40 BATCH_SIZE  = 64
41 STEPS       = len(X_train) // BATCH_SIZE
42
43 @jax.jit
44 def mse(params, x, y):
45     return ((model.apply({"params": params}, x) - y) ** 2).mean()
46
47 @jax.jit
48 def train_step(state, xb, yb):
49     loss, grads = jax.value_and_grad(mse)(state.params, xb, yb)
50     return state.apply_gradients(grads=grads), loss
51
52 def run(name, opt):
53     params = init()
54     state = TrainState.create(apply_fn=model.apply,
55                               params=params,
56                               tx=opt)
57
58     losses = []
59     start = time.perf_counter()
60     for epoch in range(EPOCHS):
61         perm = jax.random.permutation(jax.random.key(epoch),
len(X_train))
62         Xb, yb = X_train[perm], y_train[perm]
63         epoch_loss = 0.0
64         for step in range(STEPS):
65             b = slice(step*BATCH_SIZE, (step+1)*BATCH_SIZE)
66             state, loss = train_step(state, Xb[b], yb[b])
67             epoch_loss += loss
68         losses.append(float(epoch_loss/STEPS))
69
70     wall = time.perf_counter() - start
71     final = mse(state.params, X_test, y_test)
72     print(f"{name:<12} | final loss {final:.4f} | time {wall:.2f}
s")
73     return state.params, losses
74
75 hist, preds = {}, {}
76 for name, opt in opts.items():
77     params, losses = run(name, opt)
78     hist[name] = losses
79     preds[name] = model.apply({"params": params}, X_test)
80
81 plt.figure(figsize=(12,4))
82

```

```

83 plt.subplot(1,2,1)
84 for name, ls in hist.items():
85     plt.plot(ls, label=name)
86 plt.xlabel("Epoch"); plt.ylabel("MSE"); plt.title("Loss vs Epoch")
87 plt.yscale("log"); plt.legend(); plt.grid()
88
89 plt.subplot(1,2,2)
90 for name, yhat in preds.items():
91     plt.scatter(y_test, yhat, alpha=0.6, label=name)
92 lims = [y_test.min(), y_test.max()]
93 plt.plot(lims, lims, 'k--')
94 plt.xlabel("True value"); plt.ylabel("Predicted");
plt.title("Predicted vs True")
95 plt.legend(); plt.grid()
96
97 plt.tight_layout()
98 plt.show()

```

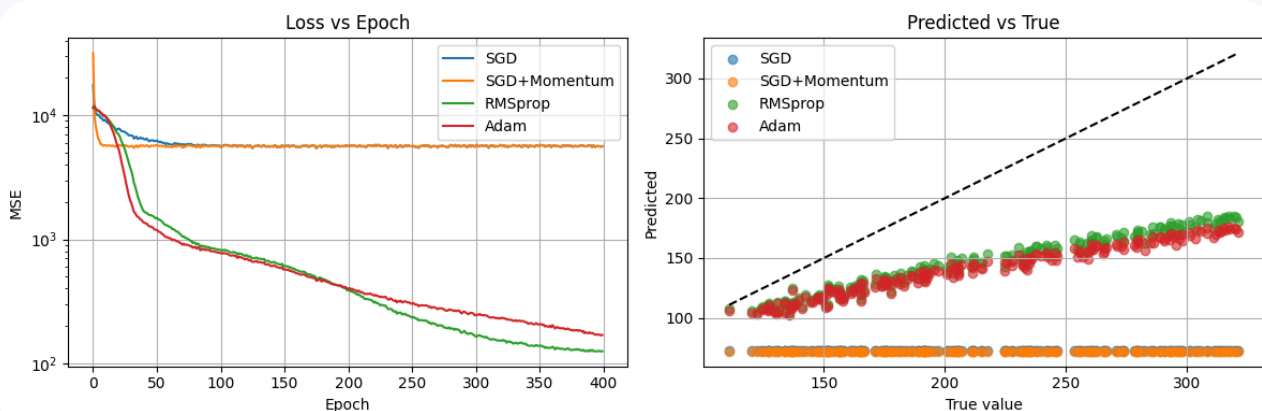
3.b Output

```

3 on 3 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

```

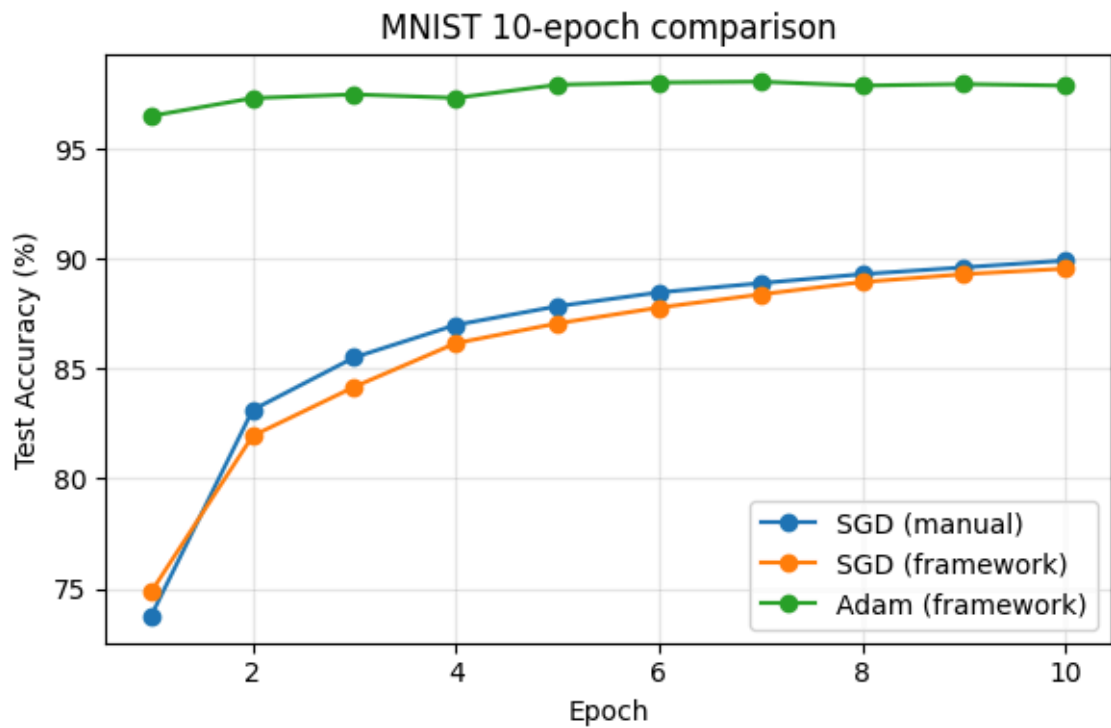
Mnist trained in jax with sgd

```

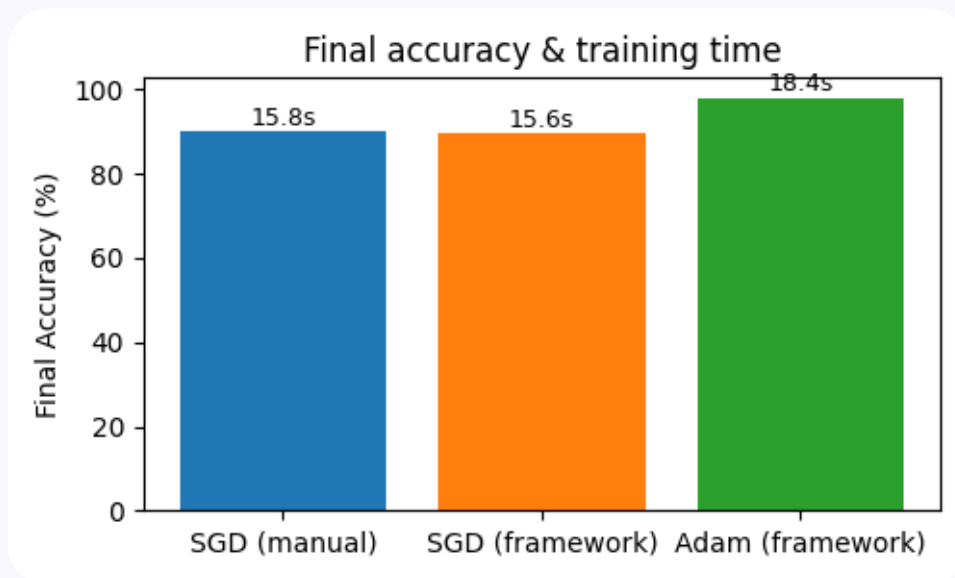
dl/ass1/mnist is v0.1.0 via v3.9.6
> uv run ./main.py
Epoch: 0, Accuracy: 96.51
Epoch: 1, Accuracy: 97.32
Epoch: 2, Accuracy: 97.49
Epoch: 3, Accuracy: 97.33
Epoch: 4, Accuracy: 97.93
Epoch: 5, Accuracy: 98.03
Epoch: 6, Accuracy: 98.07
Epoch: 7, Accuracy: 97.89
Epoch: 8, Accuracy: 97.96
Epoch: 9, Accuracy: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

```

Mnist trained in jax with adam



Accuracy



Traininig Time

Observations

- I trained 3 models
 - Mnist without any framework only using `jax.numpy` which is a python package for working with nd-arrays using `sgd` optimizer.
 - Mnist trained using `jax` dl framework using `sgd` optimizer
 - Mnist trained using `jax` dl framework using `adam` optimizer
- The adam version performed significantly better, quickly converging at 97.5% accuracy.
- However, using adam cost around 20% extra training time.
- 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.
- The `sgd` implemented using the dl framework was marginally faster to train however it performed marginally worse than the hand coded version.