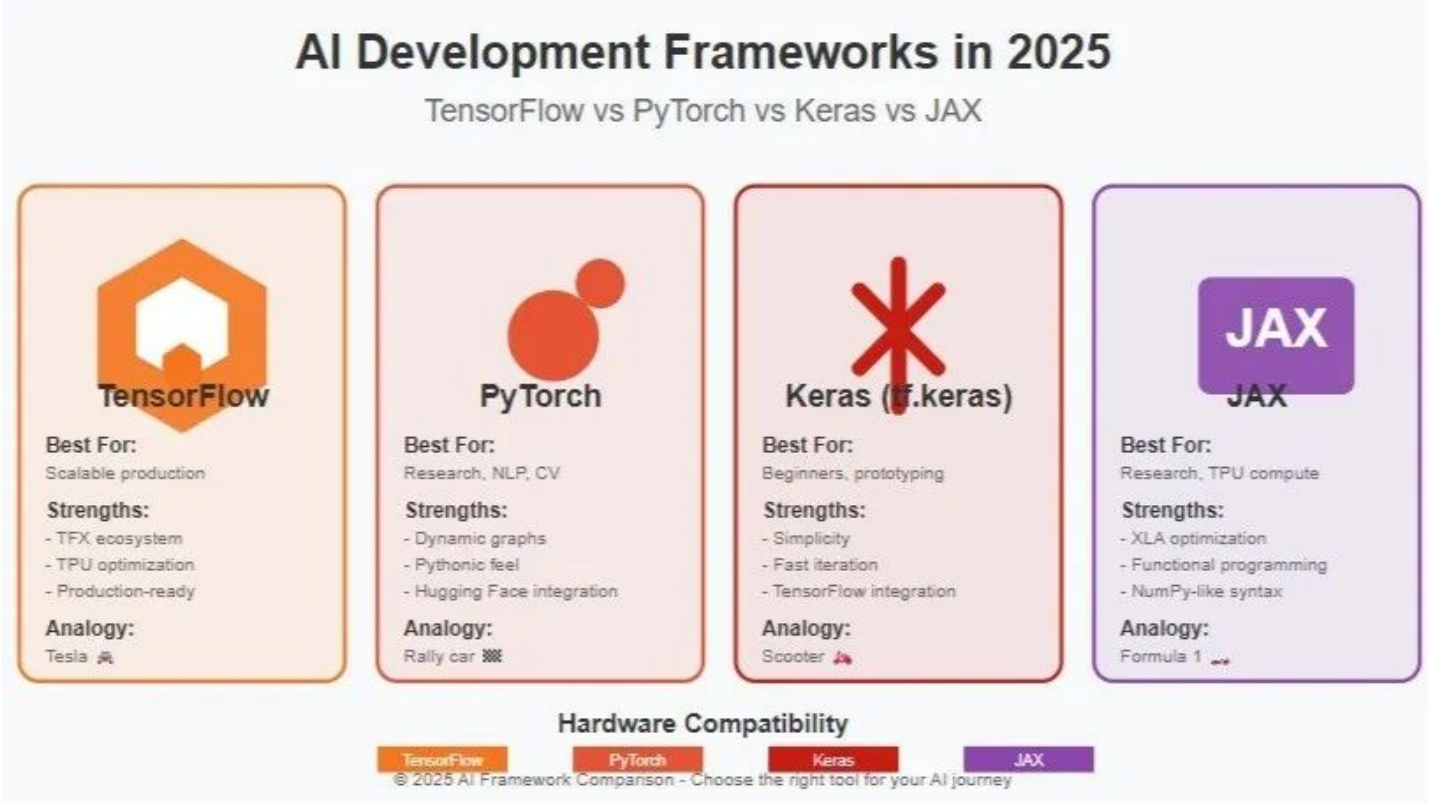
Machine learning 3a

pyTorch Intro + Ekman Chapter 5 (beginning)

Deep Learning Libraries

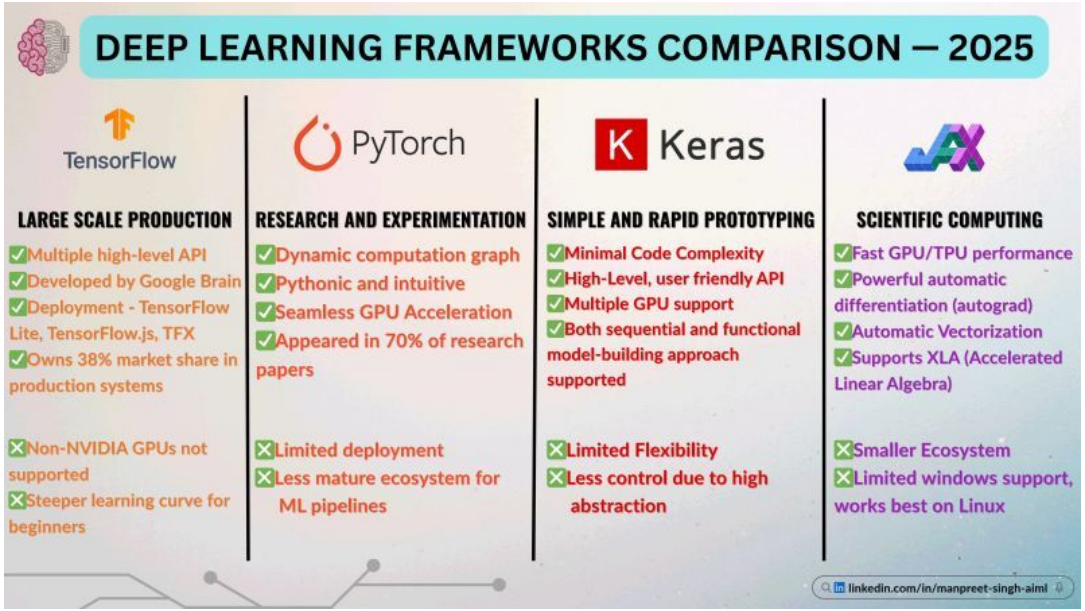


Why PyTorch

Used in 70% of research papers

Support for non-NVIDIA GPUs

Runs (very slowly) on CPUs



Availability

pyTorch and Tensorflow are to popular frameworks both are popular and flexible.

The TensorFlow versions of the code examples are interspersed throughout the book, and the PyTorch versions are available online on the book Web site here: https://github.com/NVDLI/LDL

The examples in canvas are rewritten versions using PyTorch with some additions for more biological exemples

Coding - Keras and PyTorch

If you are not so familiar with Python, it is worth pointing out that functions can be defined with optional arguments, and to avoid having to pass the arguments in a specific order, optional arguments can be passed by first naming which argument we are trying to set. An example is the **num_classes** argument in the **to_categorical** function.

Code Snippet 5-3 Create the Network

```
# Object used to initialize weights.
initializer = keras.initializers.RandomUniform(
    minval=-0.1, maxval=0.1)

# Create a Sequential model.
# 784 inputs.
# Two Dense (fully connected) layers with 25 and 10 neurons.
# tanh as activation function for hidden layer.
# Logistic (sigmoid) as activation function for output layer.
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(25, activation='tanh',
                       kernel_initializer=initializer,
                       bias_initializer='zeros'),
    keras.layers.Dense(10, activation='sigmoid',
                       kernel_initializer=initializer,
                       bias_initializer='zeros')])
```

7) TensorFlow → PyTorch mapping (quick)

- `tf.data.Dataset` → `DataLoader`
- `tf.keras.Model` → `nn.Module`
- `GradientTape()` → `loss.backward()`
- `optimizer.apply_gradients` → `optimizer.step()`
- `SparseCategoricalCrossentropy(from_logits=True)` → `CrossEntropyLoss()`

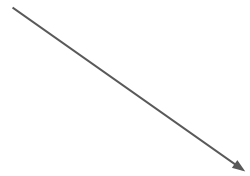
```
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, padding=2)
        self.pool = nn.MaxPool2d(2,2)
        self.fc1 = nn.Linear(32*7*7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        return self.fc2(x)  # logits

model = SimpleCNN().to(device)
model
```

Why PyTorch

TorchVision
A package for computer vision tasks, offering datasets, models, and transformations.
TorchText
A library for handling text data and NLP tasks.
PyTorch Lightning
A high-level interface for PyTorch that helps organize complex codebases and reduce boilerplate.
ONNX (Open Neural Network Exchange)
PyTorch can export models to ONNX format, which allows for interoperability between frameworks and deployment in other systems.

What is PyTorch

Check !



PyTorch for Biology: A Gentle, Code-First Intro

This notebook is for students with a biological background who want a fast, practical start with PyTorch. We'll resembling gene expression and promoter detection so everything runs quickly on CPU.

```
# Section 1: Setup and installation checks
import torch, numpy as np, random, pandas as pd
import matplotlib.pyplot as plt

# Reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

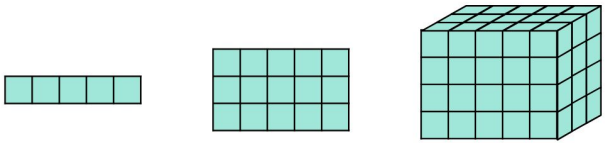
[1] ✓ 2.1s

```
PyTorch version: 2.5.1
CUDA available: True
Using device: cuda
```

PyTorch

Tensors

- High-dimensional matrices (arrays)



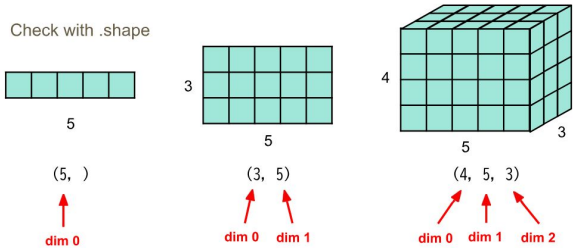
1-D tensor
e.g. audio

2-D tensor
e.g. black&white images

3-D tensor
e.g. RGB images

Tensors – Shape of Tensors

- Check with .shape



Note: `dim` in PyTorch == `axis` in NumPy

2. Tensors: creation, shapes, and dtypes

We'll start with DNA-encoded toy data (A,T,G,C mapped to 0–3) to show tensor creation, shapes, and dtypes.

```
# Small DNA example: 4 samples, 8 bases each encoded as ints 0-3
bases
    [ Generate docs (⌘ + .)
    [3,3,2,2,1,1,0,0],
    [1,0,1,0,2,2,3,3],
    [2,2,1,1,0,0,3,3],
], dtype=np.int64)

tensor_from_np = torch.tensor(bases)                # infers dtype/int64
float_tensor = torch.tensor(bases, dtype=torch.float32)
zeros = torch.zeros((2,3), dtype=torch.float32)
ones  = torch.ones((2,3), dtype=torch.float32)
rand  = torch.rand((2,3), dtype=torch.float32)

print("tensor_from_np:", tensor_from_np.shape, tensor_from_np.dtype)
print("float_tensor :", float_tensor.shape, float_tensor.dtype)
print("zeros shape   :", zeros.shape, "ones shape:", ones.shape, "rand shape:", rand.shape)
```

✓ 0.0s

```
tensor_from_np: torch.Size([4, 8]) torch.int64
float_tensor : torch.Size([4, 8]) torch.float32
zeros shape   : torch.Size([2, 3]) ones shape: torch.Size([2, 3]) rand shape: torch.Size([2, 3])
```

Tensors – Creating Tensors

- Directly from data (list or numpy.ndarray)

```
x = torch.tensor([[1, -1], [-1, 1]])
x = torch.from_numpy(np.array([[1, -1], [-1, 1]]))
```

```
tensor([[1., -1.],
        [-1., 1.]])
```

- Tensor of constant zeros & ones

```
x = torch.zeros([2, 2])
x = torch.ones([1, 2, 5])
```

shape

```
tensor([[0., 0.],
        [0., 0.]])
```

```
tensor([[[1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.]]])
```

Tensors – Data Type

- Using different data types for model and data will cause errors.

Data type	dtype	tensor
32-bit floating point	`torch.float`	`torch.FloatTensor`
64-bit integer (signed)	`torch.long`	`torch.LongTensor`

PyTorch

Tensors – Common Operations

Common arithmetic functions are supported, such as:

- Addition

```
z = x + y
```

- Summation

```
y = x.sum()
```

- Subtraction

```
z = x - y
```

- Mean

```
y = x.mean()
```

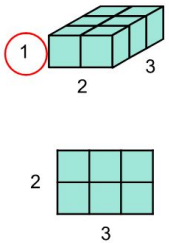
- Power

```
y = x.pow(2)
```

Tensors – Common Operations

- **Squeeze**: remove the specified dimension with length = 1

```
>>> x = torch.zeros([1, 2, 3])
>>> x.shape
torch.Size([1, 2, 3])
>>> x = x.squeeze(0)
(dim = 0)
>>> x.shape
torch.Size([2, 3])
```



3. Basic tensor operations (indexing, slicing, reshaping)

Think of rows as samples and columns as genes/features. We'll slice, reshape, and concatenate.

```
data = float_tensor  # from earlier cell

first_two_samples = data[:2, :]          # slicing
first_four_features = data[:, :4]
reshaped = data.view(2, 2, 8)             # reshape into (batch, chunk, features)
concat = torch.cat([data, data], dim=0)   # stack samples

print("First two samples:\n", first_two_samples)
print("First four features of all samples:\n", first_four_features)
print("Reshaped (2,2,8):", reshaped.shape)
print("Concatenated shape:", concat.shape)

# Elementwise operations
mean_per_feature = data.mean(dim=0)
std_per_feature  = data.std(dim=0)
print("Feature means:", mean_per_feature)
print("Feature stds :", std_per_feature)
```

✓ 0.0s

```
First two samples:
 tensor([[0., 1., 2., 3., 0., 1., 2., 3.],
        [3., 3., 2., 2., 1., 1., 0., 0.]])
First four features of all samples:
 tensor([[0., 1., 2., 3.],
        [3., 3., 2., 2.],
        [1., 0., 1., 0.],
        [2., 2., 1., 1.]])
Reshaped (2,2,8): torch.Size([2, 2, 8])
Concatenated shape: torch.Size([8, 8])
Feature means: tensor([1.5000, 1.5000, 1.5000, 1.5000, 0.7500, 1.0000, 2.0000, 2.2500])
Feature stds : tensor([1.2910, 1.2910, 0.5774, 1.2910, 0.9574, 0.8165, 1.4142, 1.5000])
```

pyTorch - simple functions

4. Autograd: gradients on simple functions

PyTorch tracks operations when `requires_grad=True`. Let's compute a simple mean-square and get gradients.

```
x = torch.linspace(-2, 2, steps=5, requires_grad=True)
y = (x**2).mean()  # simple function
print("y:", y.item())

y.backward()       # compute dy/dx
print("dy/dx:", x.grad)
```

0.0s

```
y: 2.0
dy/dx: tensor([-0.8000, -0.4000,  0.0000,  0.4000,  0.8000])
```

Tensors – Gradient Calculation

1. `>>> x = torch.tensor([[1., 0.], [-1., 1.]], requires_grad=True)`
2. `>>> z = x.pow(2).sum()`
3. `>>> z.backward()`
4. `>>> x.grad`

```
tensor([[ 2.,  0.],
        [-2.,  2.]])
```

1. $x = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$
2. $z = \sum_i \sum_j x_{i,j}^2$
3. $\frac{\partial z}{\partial x_{i,j}} = 2x_{i,j}$
4. $\frac{\partial z}{\partial x} = \begin{bmatrix} 2 & 0 \\ -2 & 2 \end{bmatrix}$

See here to learn about gradient calculation.

pyTorch - data handling

- DataLoader

5. Data handling: loading tabular biological data

We'll simulate a small gene expression table (samples × genes) and load it with pandas, then feed it to a DataLoader.

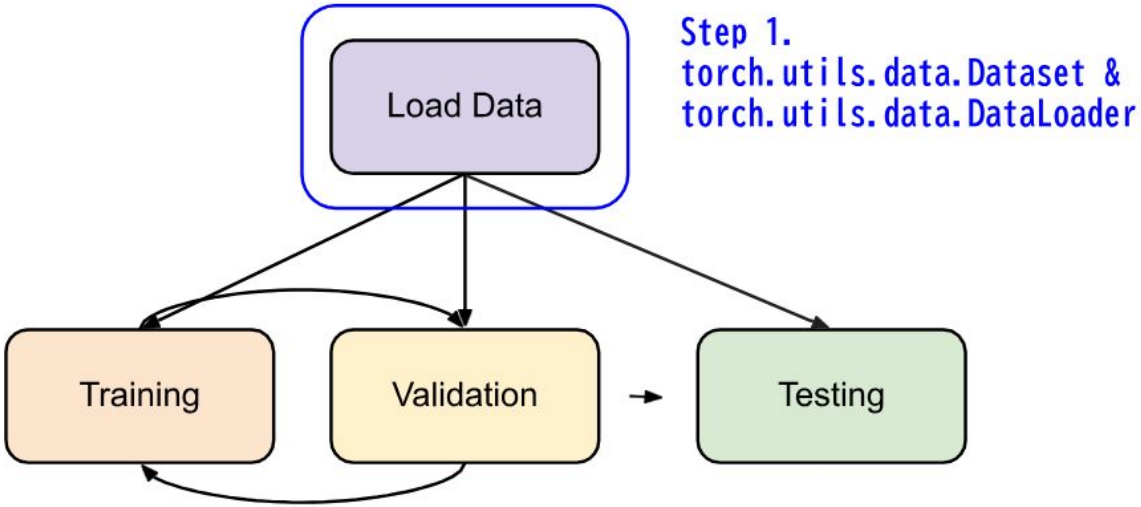
```
# Simulate a small gene expression dataset
n_samples = 200
n_genes = 20
# Healthy vs disease: disease has slightly higher expression on first 5 genes
labels = np.random.randint(0, 2, size=n_samples)
expression = np.random.normal(loc=0.0, scale=1.0, size=(n_samples, n_genes))
expression[labels == 1, :5] += 1.0  # shift disease class

# Build pandas DataFrame (for illustration of CSV-like loading)
gene_names = [f"Gene{i+1}" for i in range(n_genes)]
df = pd.DataFrame(expression, columns=gene_names)
df["label"] = labels
print(df.head())

# Convert to tensors
X = torch.tensor(df[gene_names].values, dtype=torch.float32)
y = torch.tensor(df["label"].values, dtype=torch.long)

from torch.utils.data import TensorDataset, DataLoader

dataset = TensorDataset(X, y)
train_loader = DataLoader(dataset, batch_size=32, shuffle=True)
val_loader   = DataLoader(dataset, batch_size=64, shuffle=False)
```

✓ 0.0s

```
      Gene1     Gene2     Gene3     Gene4     Gene5     Gene6     Gene7  \
0  0.087047 -0.299007  0.091761 -1.987569 -0.219672  0.357113  1.477894
1  1.296120  1.261055  1.005113  0.765413 -0.415371 -0.420645 -0.342715
2 -0.034712 -1.168678  1.142823  0.751933  0.791032 -0.909387  1.402794
3 -0.783253 -0.322062  0.813517 -1.230864  0.227460  1.307143 -1.607483
4  1.865775  0.473833 -1.191303  0.656554 -0.974682  0.787085  1.158596

      Gene8     Gene9    Gene10  ...    Gene12    Gene13    Gene14    Gene15  \
0 -0.518270 -0.808494 -0.501757  ...  0.328751 -0.529760  0.513267  0.097078
1 -0.802277 -0.161286  0.404051  ...  0.174578  0.257550 -0.074446 -1.918771
2 -1.401851  0.586857  2.190456  ... -0.566298  0.099651 -0.503476 -1.550663
3  0.184634  0.259883  0.781823  ... -1.320457  0.521942  0.296985  0.250493
4 -0.820682  0.963376  0.412781  ...  1.896793 -0.245388 -0.753736 -0.889514

     Gene16    Gene17    Gene18    Gene19    Gene20  label
0  0.968645 -0.702053 -0.327662 -0.392108 -1.463515      0
1 -0.026514  0.060230  2.463242 -0.192361  0.301547      1
2  0.068563 -1.062304  0.473592 -0.919424  1.549934      0
3  0.346448 -0.680025  0.232254  0.293072 -0.714351      0
4 -0.815810 -0.077102  0.341152  0.276691  0.827183      0

[5 rows x 21 columns]
```

Dataloaders etc

Training & Testing Neural Networks - in Pytorch



Dataset and DataLoader

Dataset & Dataloader

- Dataset: stores data samples and expected values
- Dataloader: groups data in batches, enables multiprocessing

- `dataset = MyDataset(file)`
- `dataloader = DataLoader(dataset, batch_size, shuffle=True)`

Training: True
Testing: False

How the data should look like

Dataset & Dataloader

```
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):
    def init (self, file):
        self.data = ...

    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return len(self.data)
```

Read data & preprocess

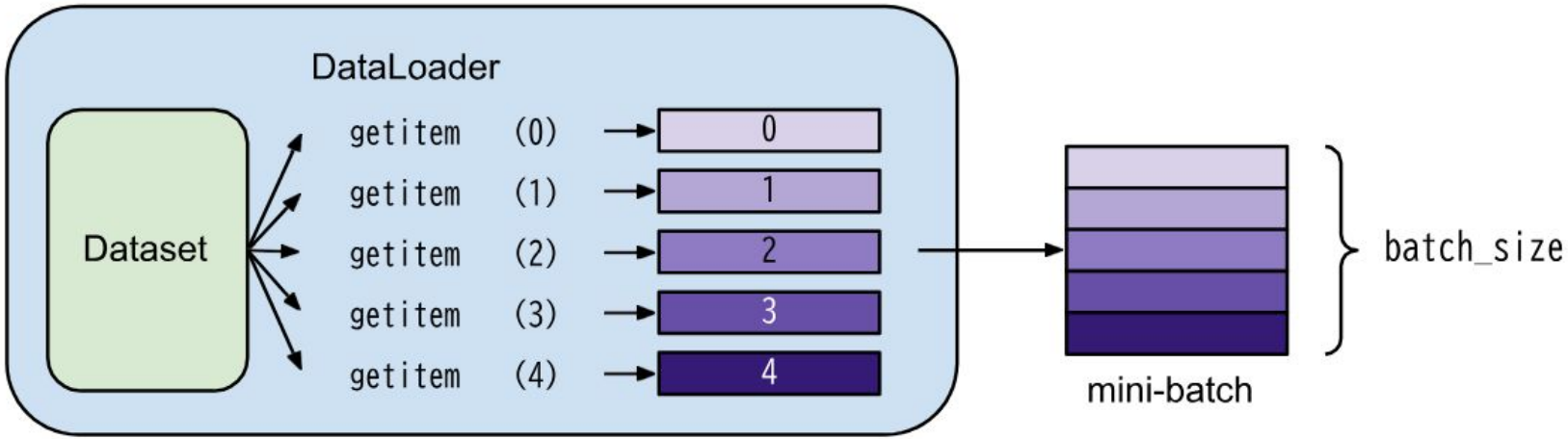
Returns one sample at a time

Returns the size of the dataset

Summary of dataloading

Dataset & Dataloader

```
dataset = MyDataset(file)

dataloader = DataLoader(dataset, batch_size=5, shuffle=False)
```



Build a network

torch.nn – Build your own neural network

```
import torch.nn as nn
```

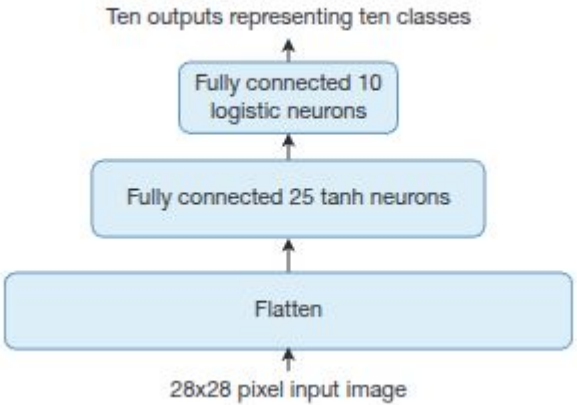
```
class MyModel(nn.Module):
    def init(self):
        super(MyModel, self).init ()
        self.net = nn.Sequential(
            nn.Linear(10, 32),
            nn.Sigmoid(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.net(x)
```

=

```
import torch.nn as nn
```

```
class MyModel(nn.Module):
    def _init_(self):
        super(MyModel, self).__init__()
        self.layer1 = nn.Linear(10, 32)
        self.layer2 = nn.Sigmoid()
        self.layer3 = nn.Linear(32,1)

    def  forward(self,  x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        return out
```

PyTorch a simple network



6. Building a simple feedforward network

We use a tiny multilayer perceptron: input → ReLU → hidden → ReLU → output (2 classes).

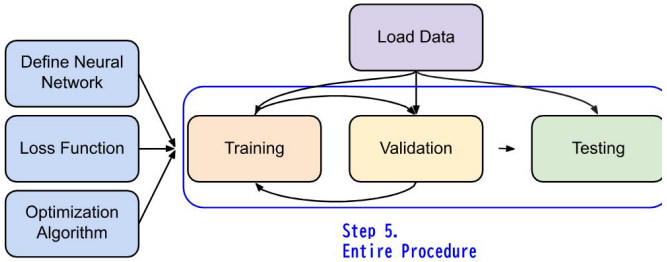
```
import torch.nn as nn

class ExpressionNet(nn.Module):
    def __init__(self, n_features, hidden=32):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_features, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, 2)
        )
    def forward(self, x):
        return self.net(x)

model = ExpressionNet(n_genes).to(device)
print(model)
print("Trainable parameters:", sum(p.numel() for p in model.parameters()))
```

[6] ✓ 0.3s

```
ExpressionNet(
  (net): Sequential(
    (0): Linear(in_features=20, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=32, bias=True)
    (3): ReLU()
    (4): Linear(in_features=32, out_features=2, bias=True)
  )
)
Trainable parameters: 1794
```

pyTorch - Training etc

Training & Testing Neural Networks – in Pytorch



7. Training loop: loss, optimizer, metrics

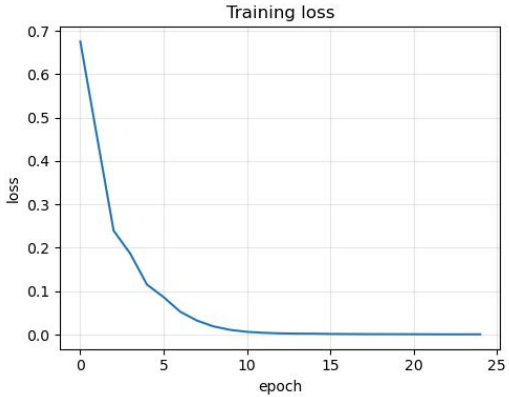
We'll train with cross-entropy loss and Adam, tracking loss and accuracy per epoch.

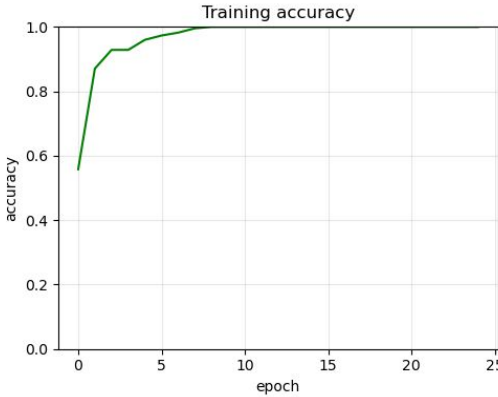
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)

def accuracy(logits, y_true):
    preds = logits.argmax(dim=1)
    return (preds == y_true).float().mean().item()

n_epochs = 25
history = {"train_loss": [], "train_acc": []}

for epoch in range(1, n_epochs+1):
    model.train()
    total_loss, total_acc, n_batches = 0.0, 0.0, 0
    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)
        optimizer.zero_grad()
        logits = model(xb)
        loss = criterion(logits, yb)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        total_acc += accuracy(logits, yb)
        n_batches += 1
    history["train_loss"].append(total_loss / n_batches)
    history["train_acc"].append(total_acc / n_batches)
    if epoch % 5 == 0:
        print(f"Epoch {epoch:02d} | loss {history['train_loss'][-1]:.4f} | acc {history['train_acc'][-1]:.3f}")
```

✓ 1.3s

```
Epoch 05 | loss 0.1153 | acc 0.960
Epoch 10 | loss 0.0108 | acc 1.000
Epoch 15 | loss 0.0021 | acc 1.000
Epoch 20 | loss 0.0009 | acc 1.000
Epoch 25 | loss 0.0006 | acc 1.000
```

Toy Biological problem (gene expression)





8. Toy biological task: gene-expression classification

We simulate healthy vs disease expression. Let's visualize training curves.

```
# Plot loss and accuracy
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(history["train_loss"], label="train loss")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.title("Training loss")
plt.grid(True, alpha=0.3)

plt.subplot(1,2,2)
plt.plot(history["train_acc"], label="train acc", color="green")
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.title("Training accuracy")
plt.ylim(0,1)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

(function) alpha: float

Saving and loading models

9. Saving and loading models

Persist model weights with `state_dict`; reload to continue training or run inference later.

```
ckpt_path = "expression_net.pt"
torch.save(model.state_dict(), ckpt_path)
print(f"Saved to {ckpt_path}")

# Load into a fresh model
loaded_model = ExpressionNet(n_genes).to(device)
loaded_model.load_state_dict(torch.load(ckpt_path, map_location=device))
loaded_model.eval()

# Verify a quick prediction matches shape
with torch.no_grad():
    test_logits = loaded_model(X[:2].to(device))
    print("Loaded model logits shape:", test_logits.shape)
```

[11] ✓ 0.0s

```
Saved to expression_net.pt
Loaded model logits shape: torch.Size([2, 2])
/tmp/ipykernel_2782383/237955992.py:7: FutureWarning: You are using `torch.load` with `weights_onl
  loaded_model.load_state_dict(torch.load(ckpt_path, map_location=device))
```