



OVERVIEW & SETUP

```
BentoML AI/ML framework
pip install bentoml
# Verify installation
python -c "import bentoml; print('OK')"
```

GPU support check

```
import torch;
print(torch.cuda.is_available()),
```

Use virtual environment: `python -m venv venv`

DATA PREPARATION

```
import pandas as pd
df = pd.read_csv('data.csv')
X = df.drop('target', axis=1).values
y = df['target'].values
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2)
```

Normalize/standardize features for best results

MODEL DEFINITION

```
# Simple neural network:
import torch.nn as nn
class Model(nn.Module):
def __init__(self):
super().__init__()
self.layers = nn.Sequential(
nn.Linear(input_size, 128),
nn.ReLU(),
nn.Linear(128, num_classes)
)
def forward(self, x): return
self.layers(x)
```

TRAINING LOOP

```
optimizer =
torch.optim.Adam(model.parameters(),
lr=1e-3)
criterion = nn.CrossEntropyLoss()
for epoch in range(num_epochs):
for batch_X, batch_y in dataloader:
optimizer.zero_grad()
output = model(batch_X)
loss = criterion(output, batch_y)
loss.backward()
optimizer.step()
```

EVALUATION

```
from sklearn.metrics import
accuracy_score, classification_report
model.eval()
with torch.no_grad():
preds = model(X_test)
accuracy = accuracy_score(y_test,
preds.argmax(1))
print(classification_report(y_test,
preds.argmax(1)))
```

Track precision, recall, F1; use confusion matrix

INFERENCE & SERVING

```
# Save model
torch.save(model.state_dict(),
'model.pt')
# Load model
model.load_state_dict(torch.load('model.pt'))
model.eval()
# Inference
with torch.no_grad():
prediction = model(input_tensor)
```

HYPERPARAMETERS

Learning rate: start with 1e-3, tune with scheduler

Batch size: 32-256 depending on GPU memory

Epochs: use early stopping to prevent overfitting

```
scheduler =
torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer)
scheduler.step(val_loss)
```

Grid search or Optuna for systematic tuning

DATASETS & LOADERS

```
from torch.utils.data import Dataset,
DataLoader
class MyDataset(Dataset):
def __len__(self): return len(self.data)
def __getitem__(self, idx): return
self.data[idx]
```

loader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4)

Pin memory for faster GPU data transfer

OPTIMIZATION TECHNIQUES

Mixed precision:

```
torch.cuda.amp.autocast()",
```

Gradient clipping:

```
clip_grad_norm_(model.parameters(),
1.0)",
```

Batch normalization:

```
nn.BatchNorm1d()",
```

Dropout: `nn.Dropout(0.5)` for regularization",

Learning rate warmup

```
from transformers import
get_linear_schedule_with_warmup
```

FRAMEWORKS & TOOLS

TensorFlow / Keras	production ML
PyTorch	research & NLP
Scikit-learn	classical ML
Hugging Face	pretrained
transformers	
MLflow / W&B	experiment
tracking	
ONNX	model
export/portability	

DEPLOYMENT

```
# Export to ONNX
torch.onnx.export(model, dummy_input,
'model.onnx')
```

FastAPI serving example

```
@app.post('/predict')
def predict(data: InputData):
tensor = preprocess(data)
return model(tensor).tolist()
```

Use TorchServe, BentoML, or Triton for production

COMMON GOTCHAS

Gradient explosion: use gradient clipping",

Memory leak: always `zero_grad()` before `backward()`",

Train/eval mode: `model.train()` vs `model.eval()`",

GPU-CPU mismatch: ensure all tensors on same device",

Data leakage: split before normalization, not after",