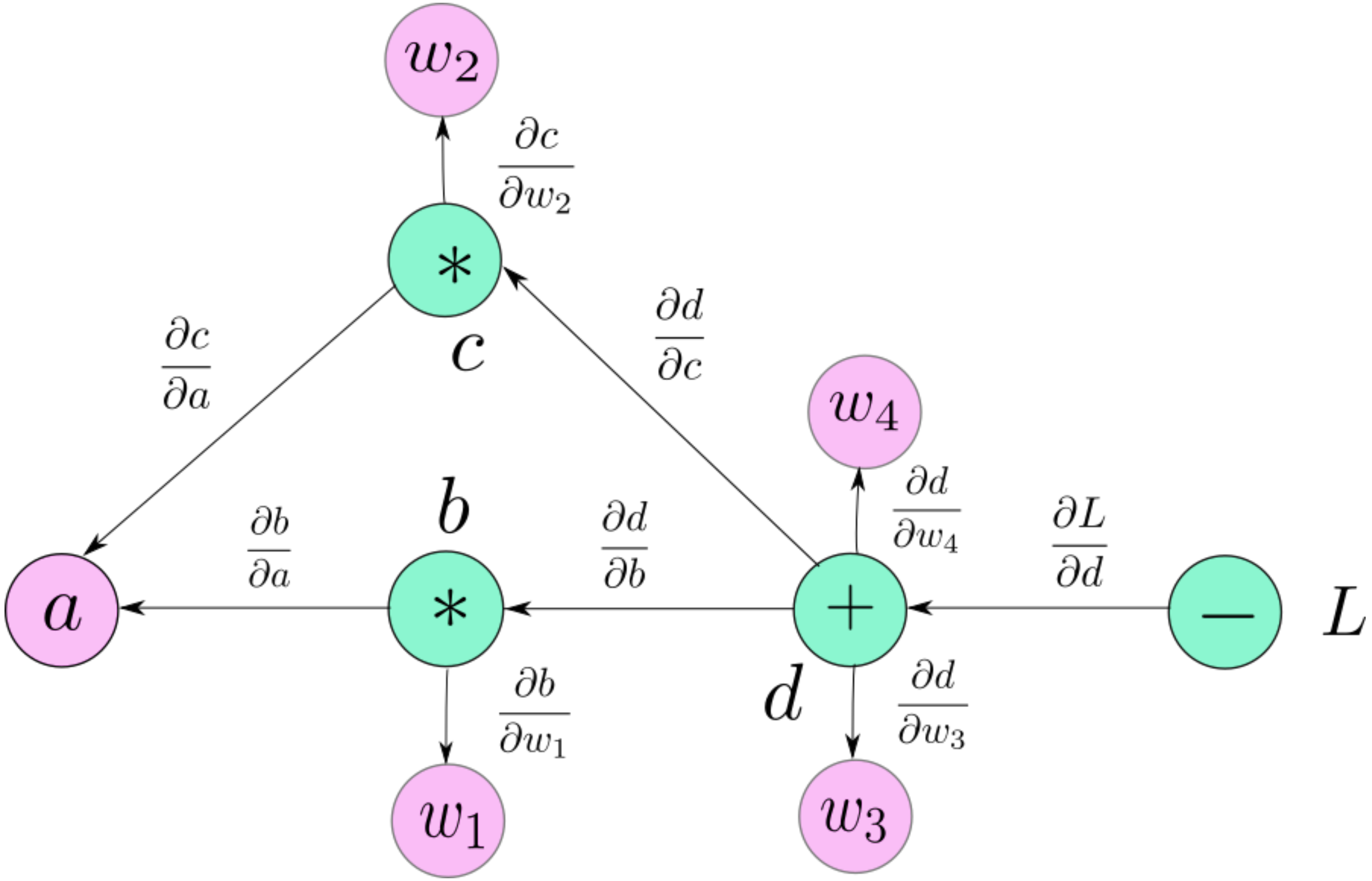
PyTorch Primer - v2

Fundamentals of GenAI

Autograd and Sequential

Data Trainers

PyTorch Autograd (autodiff) In a Nutshell



$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_4}$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_3}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial c} * \frac{\partial c}{\partial w_2}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial b} * \frac{\partial b}{\partial w_1}$$

PyTorch tensor attributes support gradient calculations

Data Trainers

```
x = pt.tensor(...)

x.data

x.is_leaf = True

x.requires_grad =
False

x.grad = None

x.grad_fn = None
```

Tensor value(s), such that x == x.data

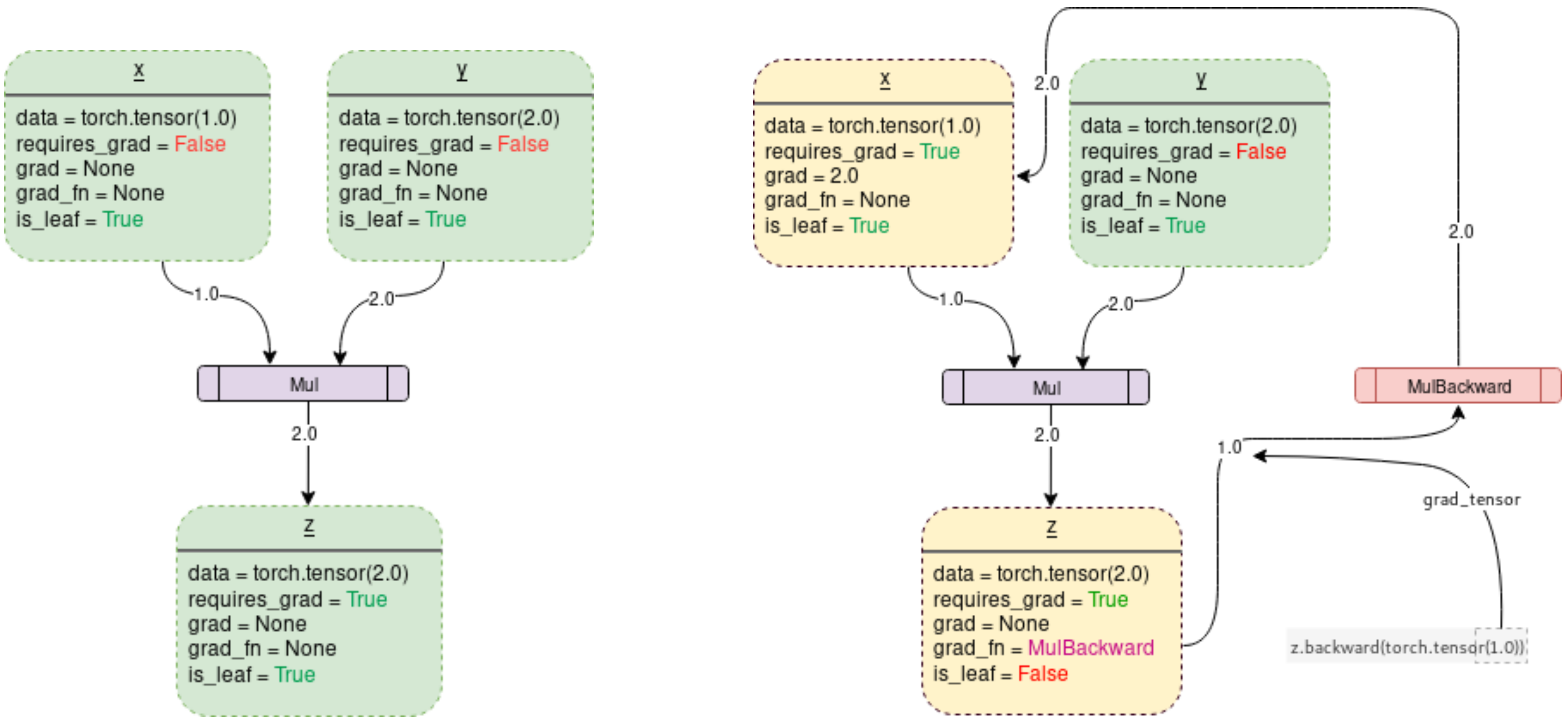
True unless created from a tensor with requires_grad=True

False when is_leaf = True

Gradient value if exists

Function that knows how to perform backward()

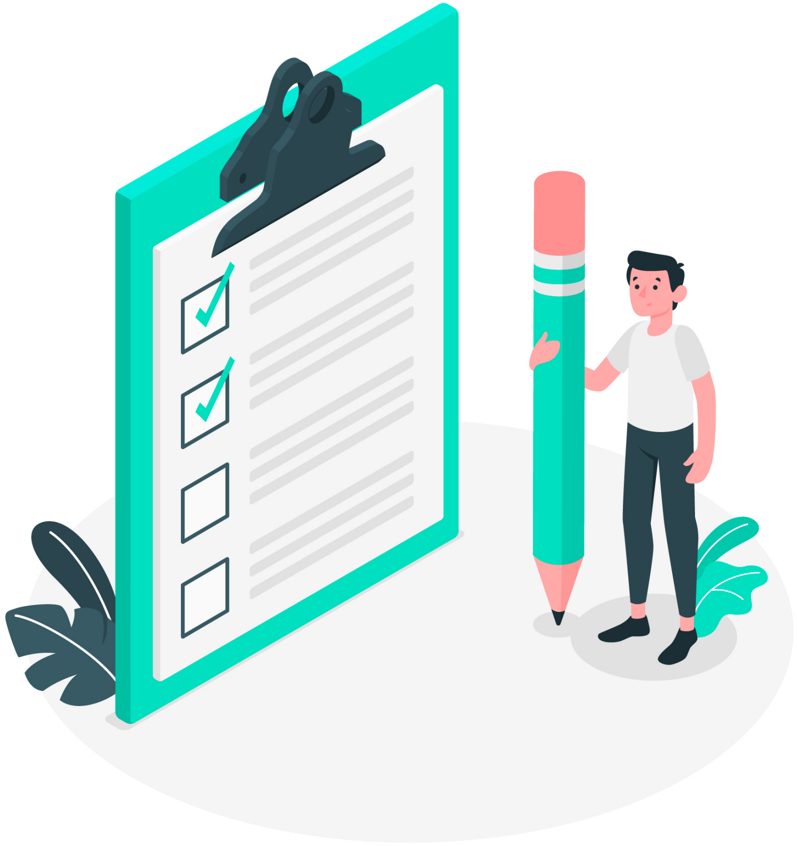
PyTorch disables tensor autodiff (autograd) by default



Demo

PyTorch AutoGrad

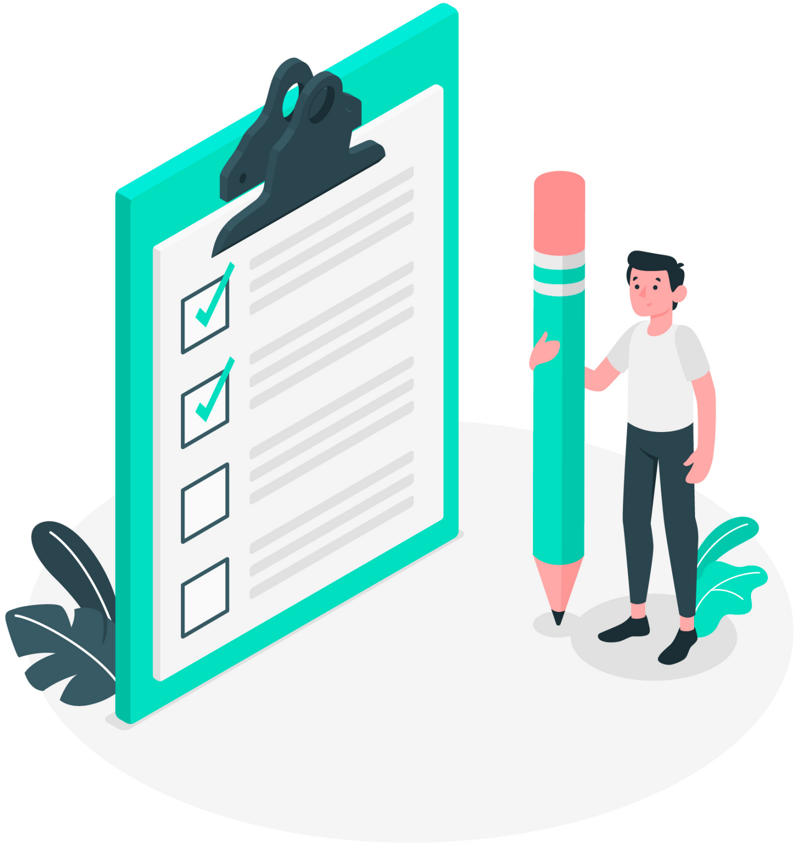
- Experiment with autograd



Demo

PyTorch AutoGrad

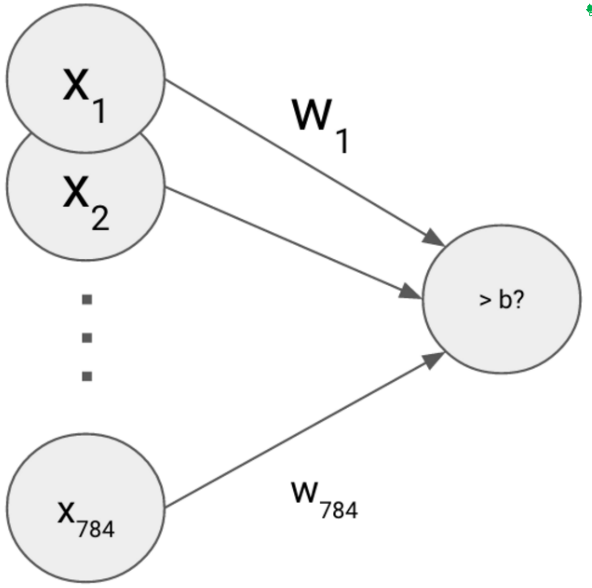
- Experiment with autograd



Layers

- **Available Layers:**
  - **Linear**
  - Convolution
  - Padding
  - Pooling
  - Normalization
  - Dropout
  - Recurrent
  - Embedding

Many-to-one, as well as fully-connected many-to-many



$$\sum_{i}^{n} w_i x_i > b$$

nn.Linear() is a fully connected layer, every input to every output

```
from torch import nn
model = nn.Linear(#_inputs,
#_outputs,
                        bias = True)

model.parameters()
model.weight, model.bias

model.zero_grad()

model.weight.grad, model.bias.grad
```

```
model(X)
```

Bias by default

Weights & bias

Zero out gradients

Access gradient values

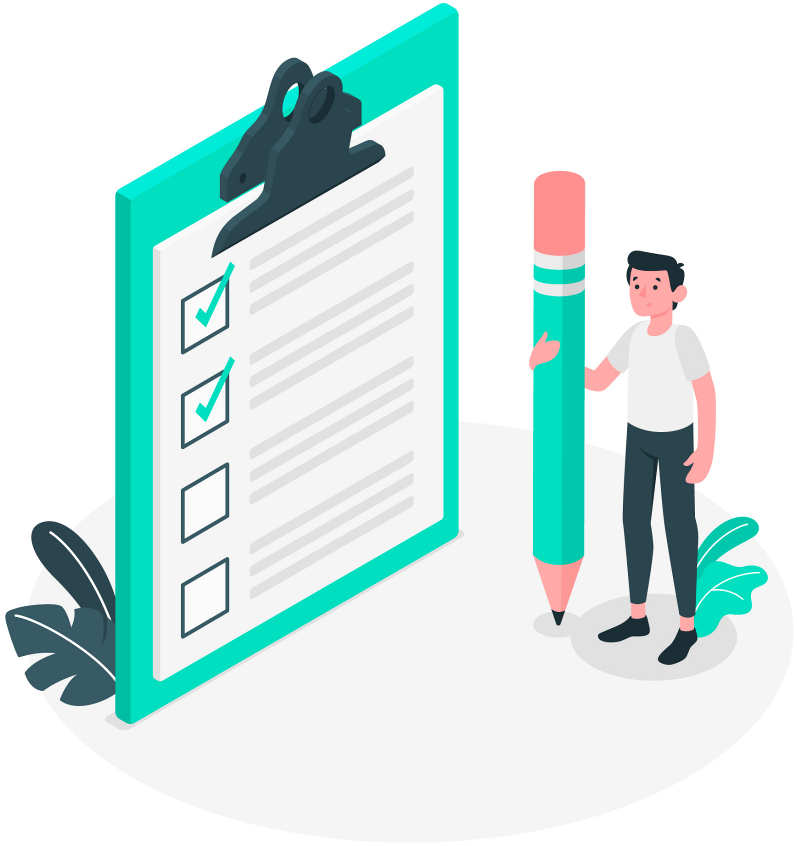
forward()

Exercise

Data Trainers
Text Similarity

Binary Classifier with nn.Linear

- Use the nn package



To read sharded CSV files, use Pandas **concat** and then decode the CSV into features and targets for the training **TensorDataset**

```
from torch.utils.data import TensorDataset

df = pd.concat(
    pd.read_csv(file) for file in Path('data/').glob('part-
*.csv')
)
...


X = pt.tensor(train_df[FEATURES].values)

y = pt.tensor(train_df[TARGET].values)

train_ds = TensorDataset(y, X)
```

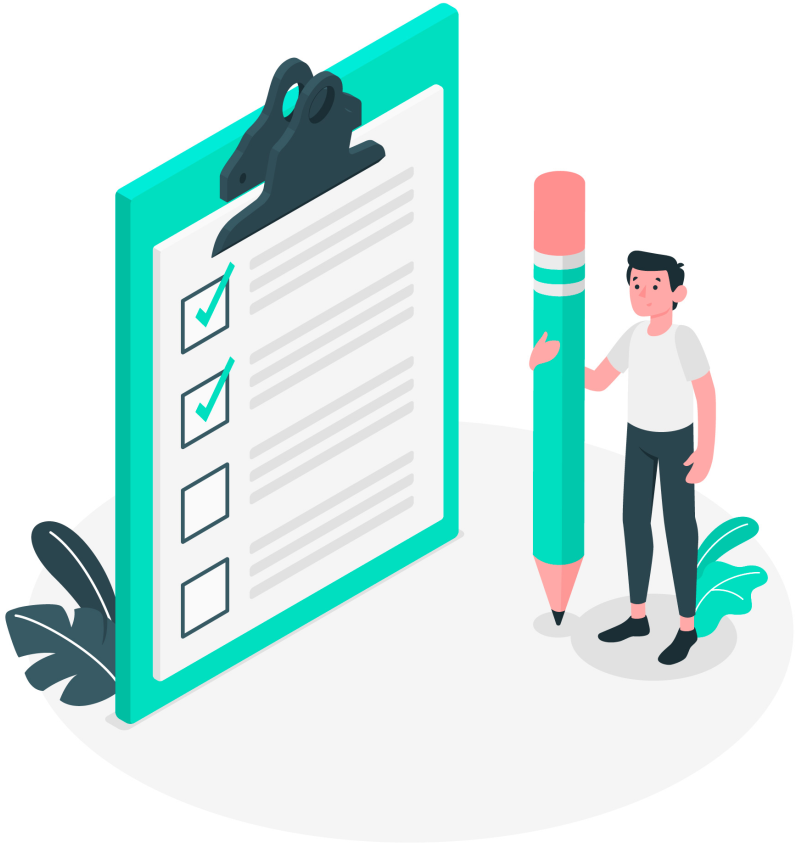
TensorDataset is used to instantiate an enumerable DataLoader, which supports data batching and indexing

```
from torch.utils.data import DataLoader

train_ds = TensorDataset(y, X)

train_dl = DataLoader(train_ds, batch_size=BATCH_SIZE)

for epoch in range(EPOCHS):

      for batch_idx, batch in enumerate(train_dl):

            y, X = batch
```

Exercise

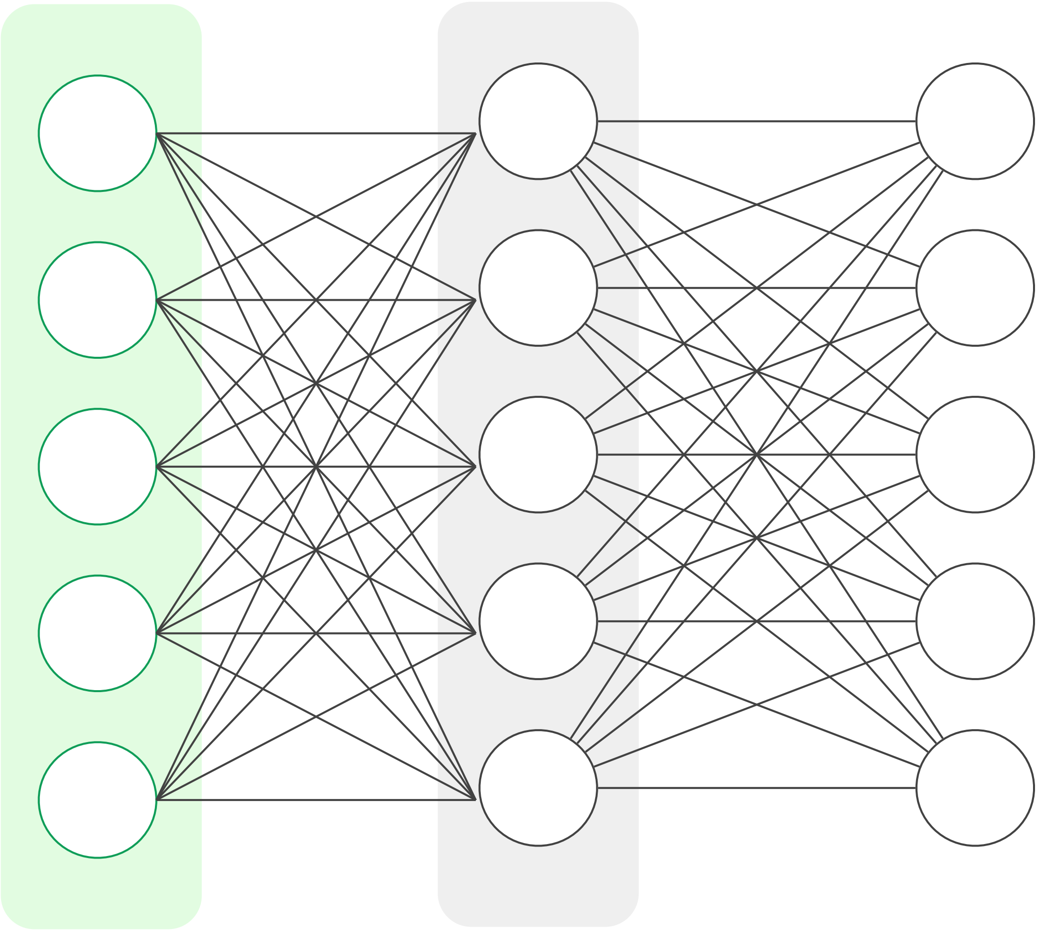
Dataset and DataLoaders

- Use the data package



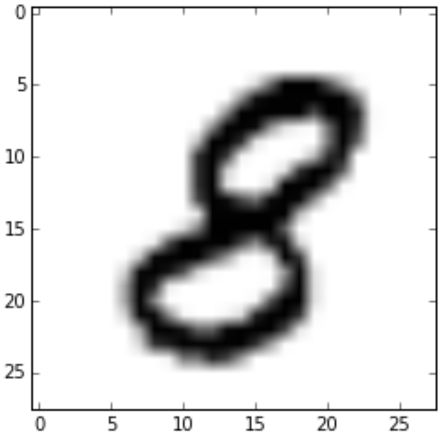
torch.nn.Sequential helps organize neural nets with many layers

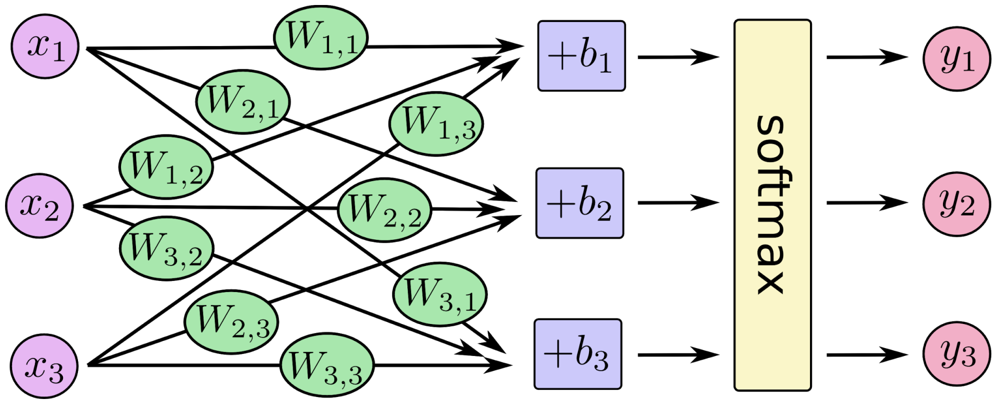
```
model = nn.Sequential(
    nn.Linear(5, 5),
    nn.ReLU(),
    nn.Linear(5, 5),
    nn.ReLU(),
    nn.Linear(5, 5)
)
model(X)
```



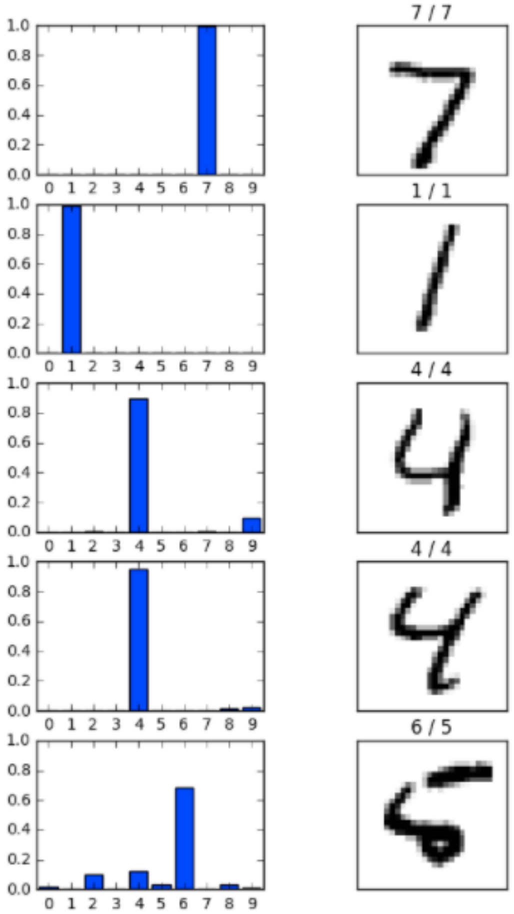
Softmax helps deal with non-binary (multivariate) target values

input vector
(pixel data)





output vector
(probability
estimate)



$$p(y=j|\mathbf{x}) = \frac{exp(\mathbf{w}_j^T\mathbf{x}+b_j)}{\sum_{k\in K} exp(\mathbf{w}_k^T\mathbf{x}+b_k)}$$

Negative log likelihood is the loss for **softmax** outputs

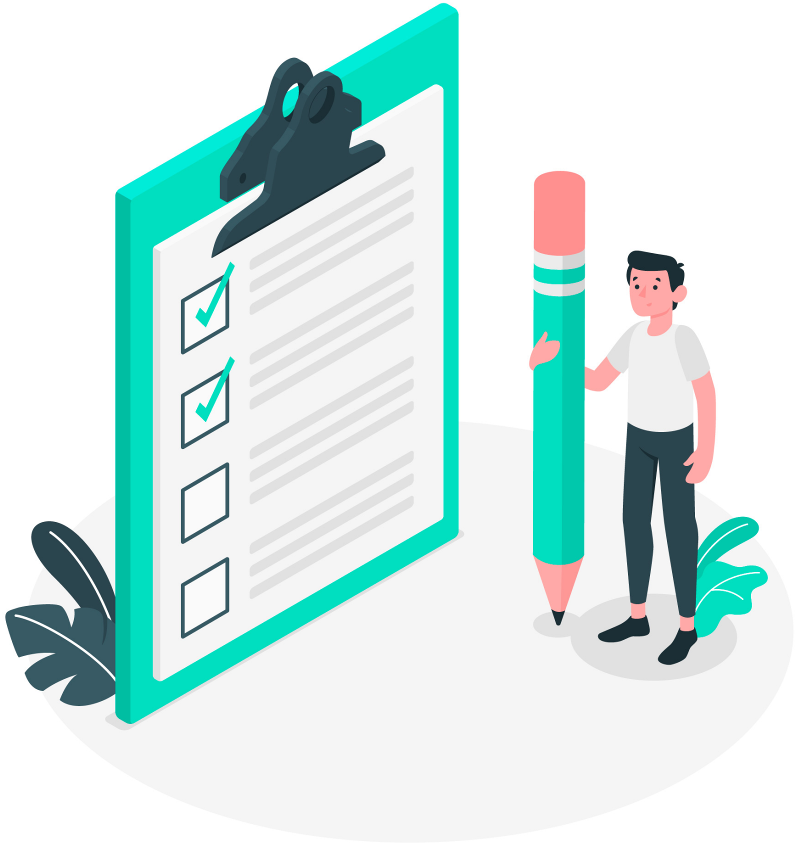
$$LogLoss = \sum_{(\mathbf{x},y)\in D} -y\log(y') - (1-y)\log(1-y')$$

torch.nn.functional.**log_softmax**() + **nll_loss**() = **cross_entropy**()

Demo

Sequential Networks

- Use nn.Sequential



Lab

Sequential Networks

- Use nn.Sequential