LAB 3:
FULLY CONNECTED DEEP NETWORKS (FCN)

University of Washington, Seattle

Spring 2022

OUTLINE

Part 1: Introduction to FCN (video link)

- Shallow vs Deep Networks
- Regression vs Classification
- Outputting Probabilities with Softmax Function

Part 2: Additional Data Prep Methods (video link)

- Training/Validation/Test sets
- One-hot encoding

Part 3: Stochastic Gradient Descent (video link)

- SGD, Mini-batch GD, Batch GD
- Implementing variants of Gradient Descent

Part 4: Additional Hyperparameters & Regularizers (video link)

- Activation functions
- Loss functions and Advanced Optimizers
- Regularizers
- Batch Normalization
- Weight initializations

Part 5: Training Fully Connected Deep Network (video link)

- Iris Classification Example

Lab Assignment (video link)

- MNIST Classification using FCN
- Tips for Training Your Model

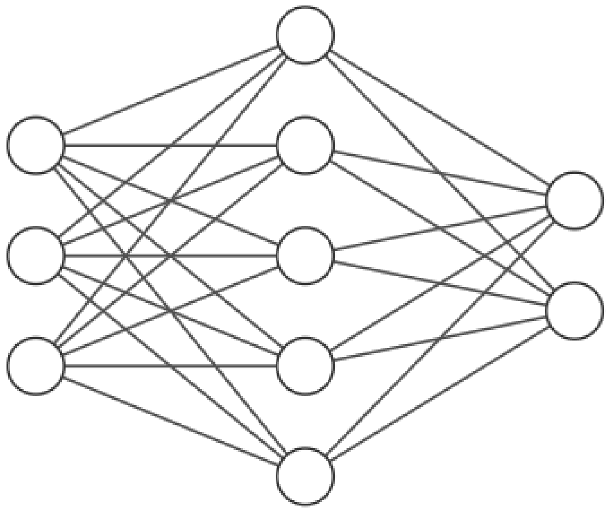
INTRODUCTION TO FCN

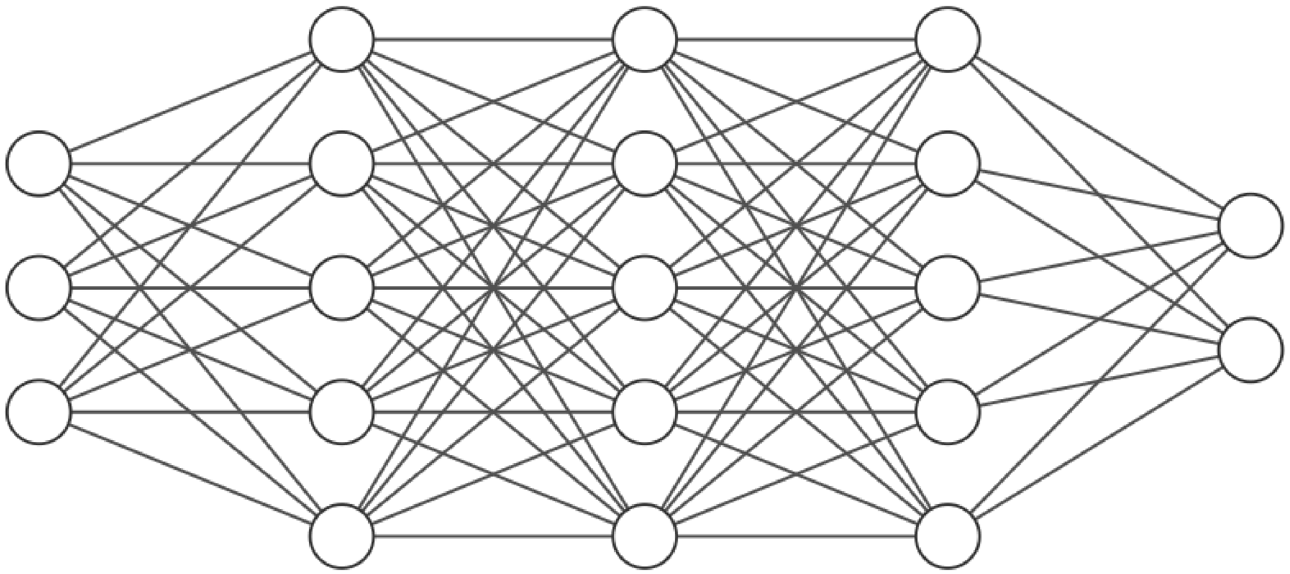
Shallow vs Deep Networks

Regression vs Classification

Outputting Probabilities with Softmax Function

Shallow vs Deep Networks





Shallow Networks

One hidden layer

Universal function approximator

(with enough hidden neurons)

Easier to train

Fit for simple problems

Deep Networks

>1 hidden layer

Upper hidden neurons reuse lower-level features

Can approximate more complex & general functions

Harder to train

Fit for more complex problems

Regression vs Classification

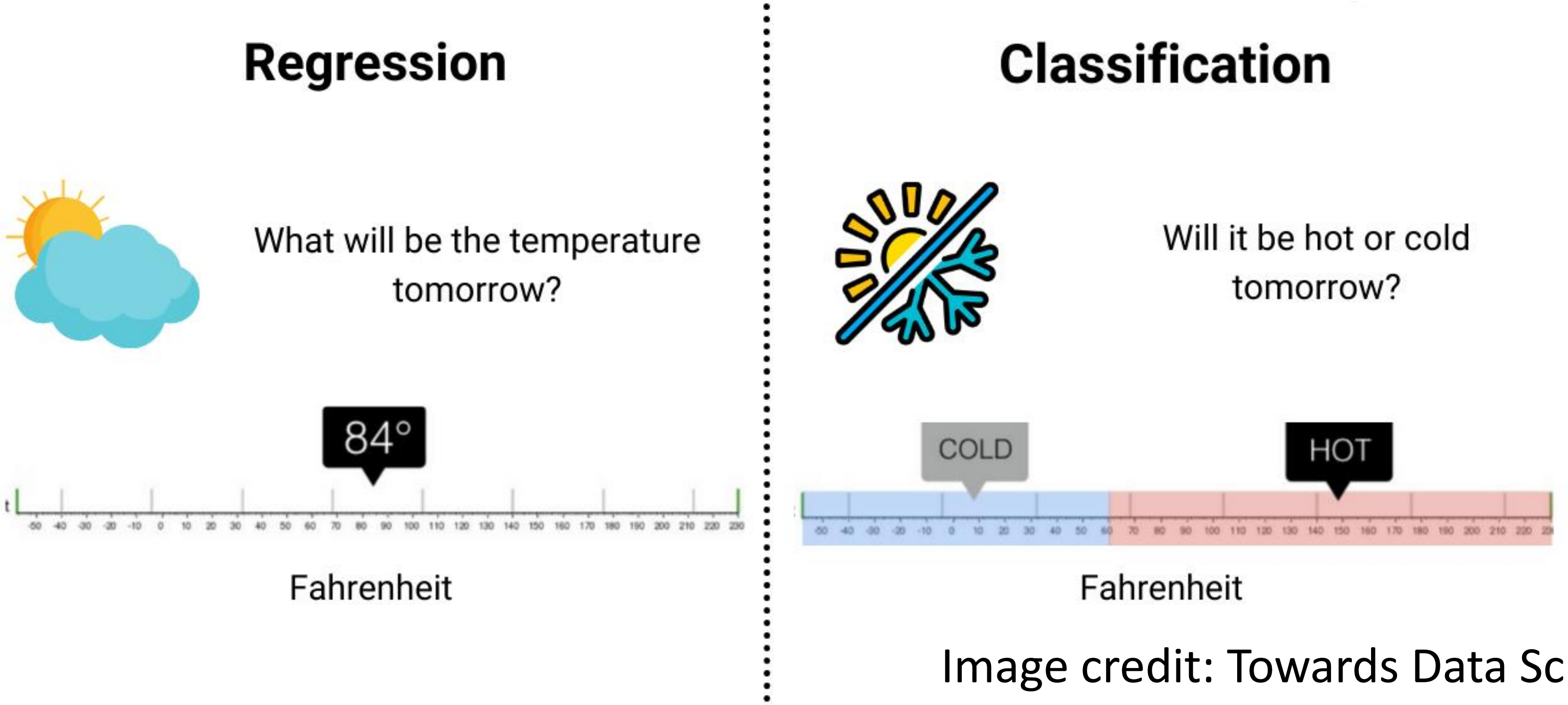
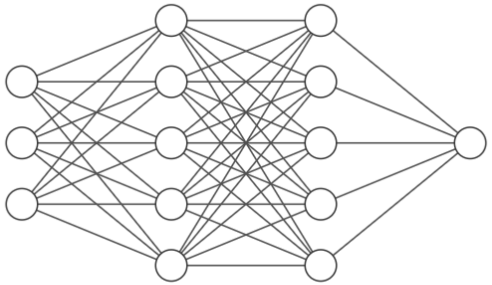
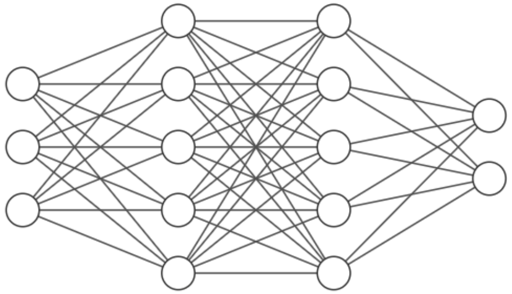


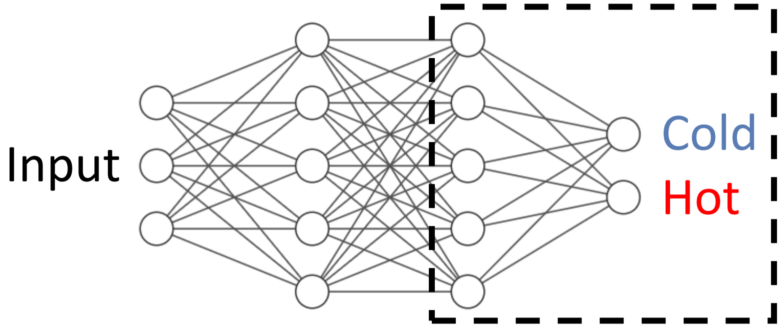
Image credit: Towards Data Science

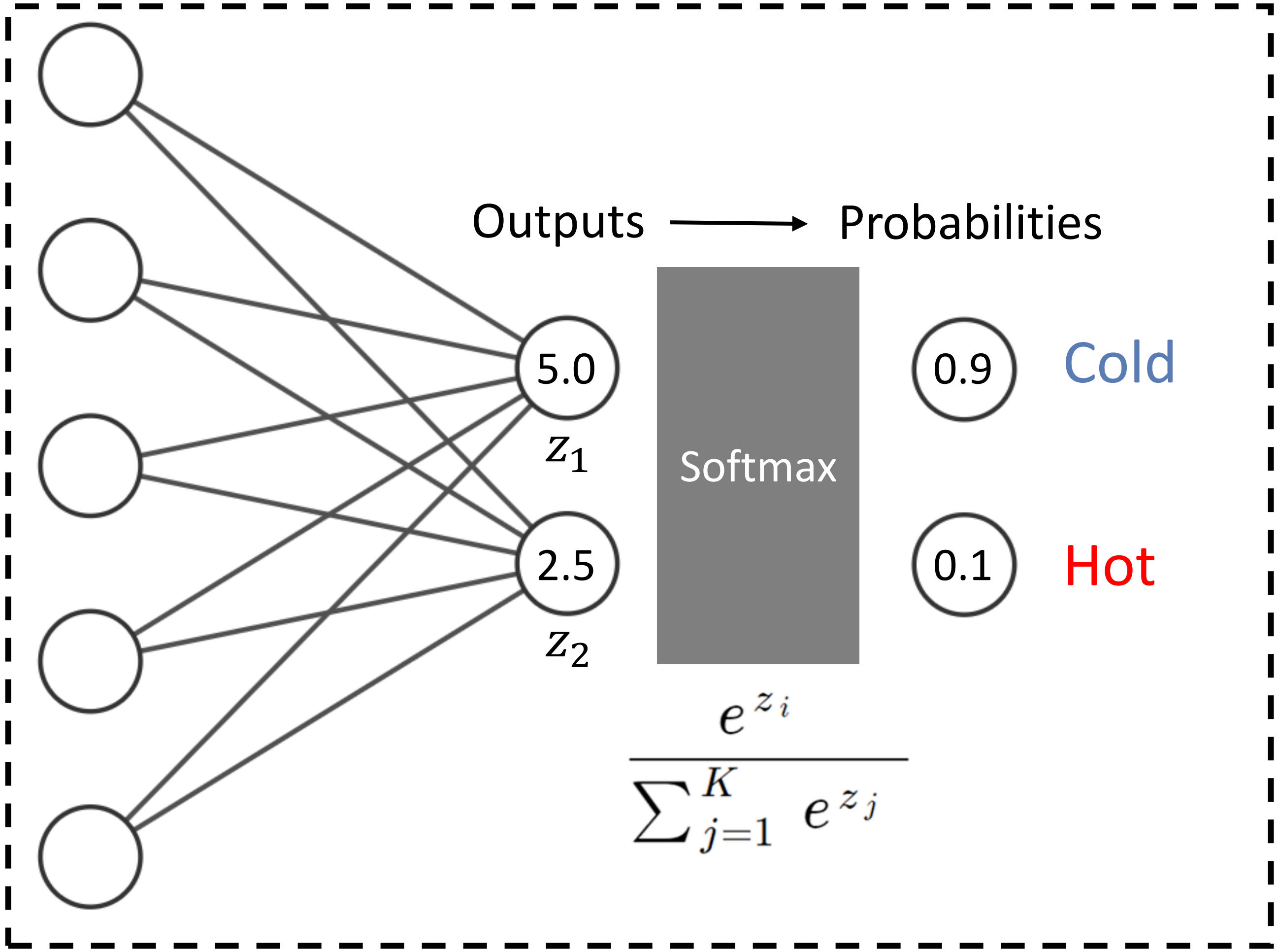
Input  84

Input 

Cold
Hot

Outputting Probabilities with Softmax Function





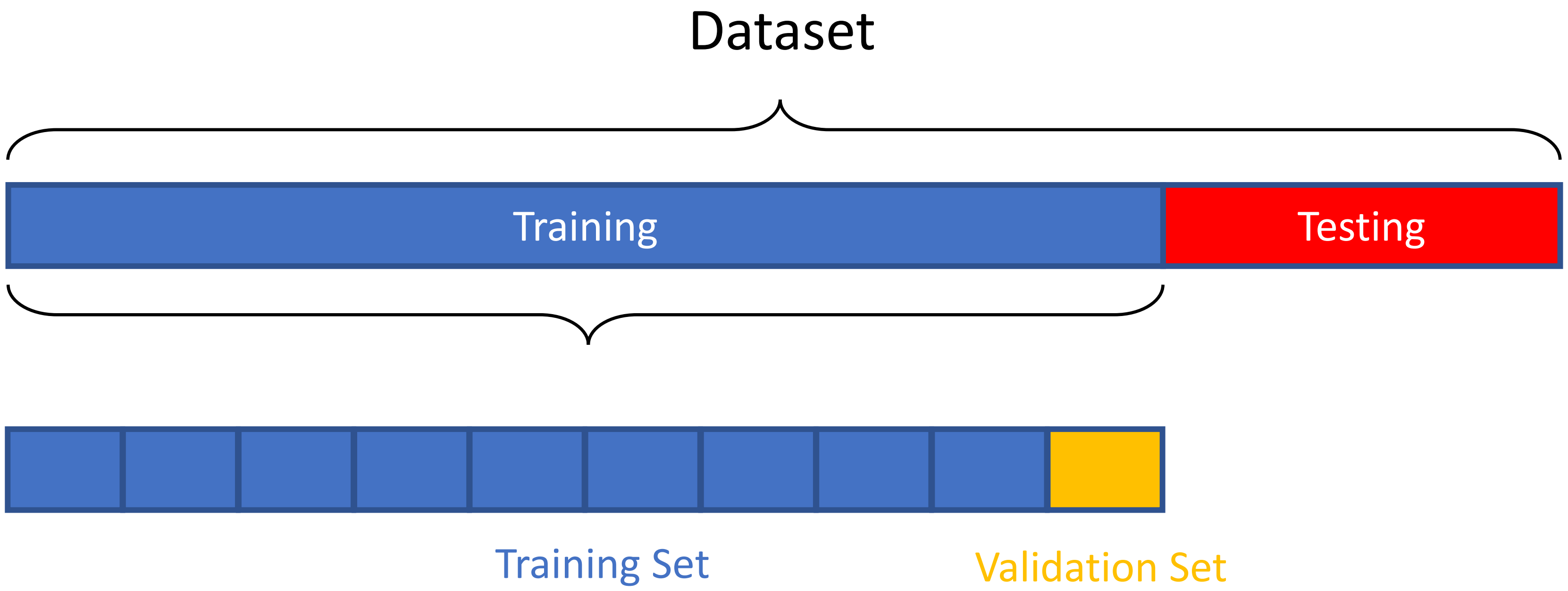
torch.nn.Softmax()

ADDITIONAL DATA PREP METHODS

Training/Validation/Test Sets

One-Hot Encoding

Train/Validation/Test Split



Common ratios

70, 15, 15

80, 10, 10

60, 20, 20

⋮

During training

Training loss ↓

Validation performance ↑

After training

Testing performance ↑

Train/Validation/Test Split

```python
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

iris = load_iris()

X = iris['data']
y = iris['target']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
                                                    test_size=0.2,
                                                    random_state=2)

X_validation = X_train[:int(len(X_test))]
y_validation = y_train[:int(len(X_test))]

X_train = X_train[int(len(X_test)):]
y_train = y_train[int(len(X_test)):]
```

Testing data ratio (0.2 = 20%)

Random seed to use for splitting

Load Iris dataset

Extract features (X) and target labels (y)

Scale features using scikit-learn provided standard scaler

Split the dataset into Training (80%) and Testing (20%)

Assign subset of the training dataset as validation data (same size as testing)

Use the remaining dataset as training

Final split ratios = Training: 60%, Testing: 20%, Validation: 20%

One-hot Encoding

Human Readable		Machine Readable				
Dog	→	1	0	0	0	0
Cat	→	0	1	0	0	0
Turtle	→	0	0	1	0	0
Fish	→	0	0	0	1	0
Snake	→	0	0	0	0	1
Categorical Data		One-hot vectors				

One-hot Encoding

```
labels = torch.tensor([0,1,2])
print(labels)
```

```
tensor([0, 1, 2])
```

Original labels are 0, 1 and 2

```
labels_one_hot = torch.nn.functional.one_hot(labels)
print(labels_one_hot)
```

```
tensor([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Labels after one-hot encoding (torch.nn.functional.one_hot())

Official function documentation:
https://pytorch.org/docs/stable/generated/torch.nn.functional.one_hot.html

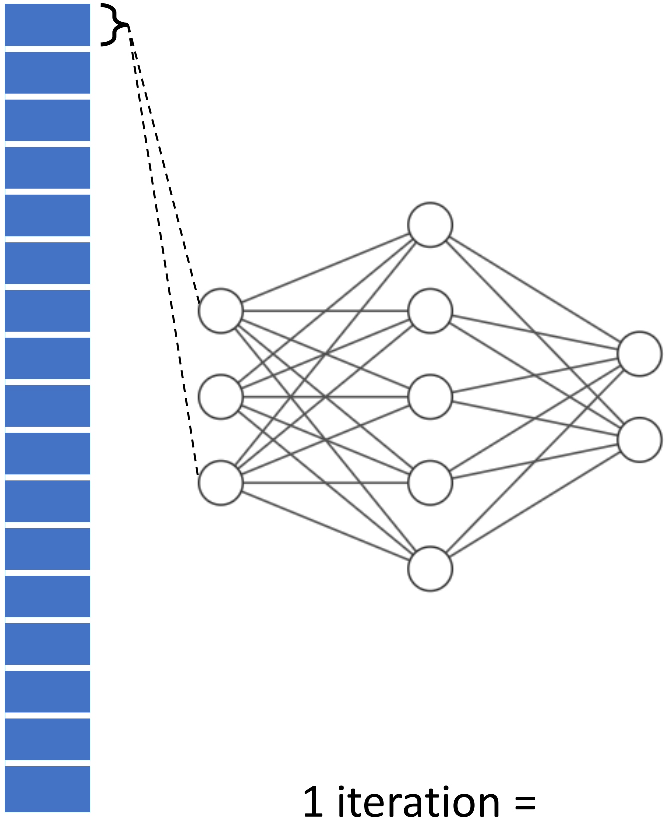
STOCHASTIC GRADIENT DESCENT

SGD, Mini-batch GD, Batch GD

Variants of Gradient Descent

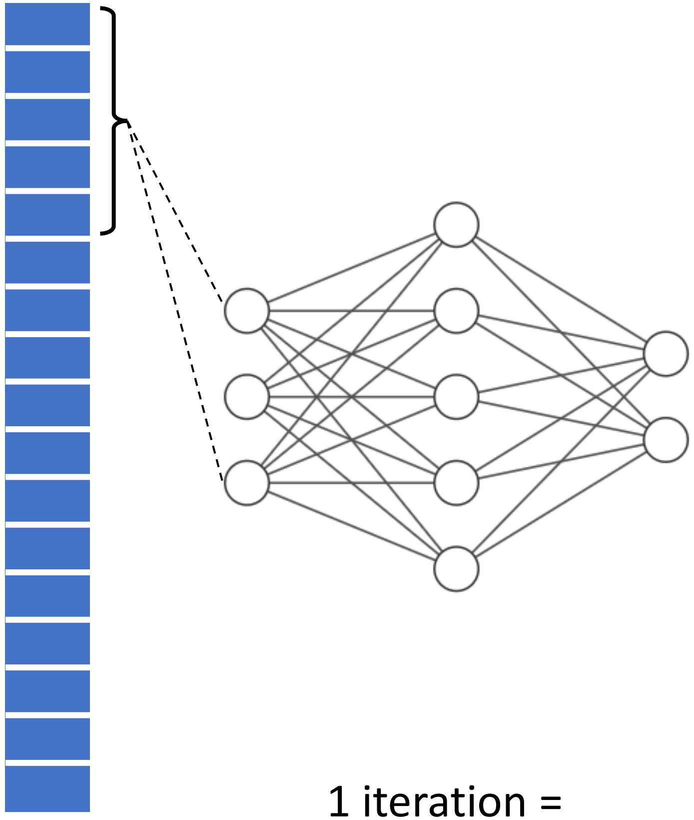
Training Dataset

SGD



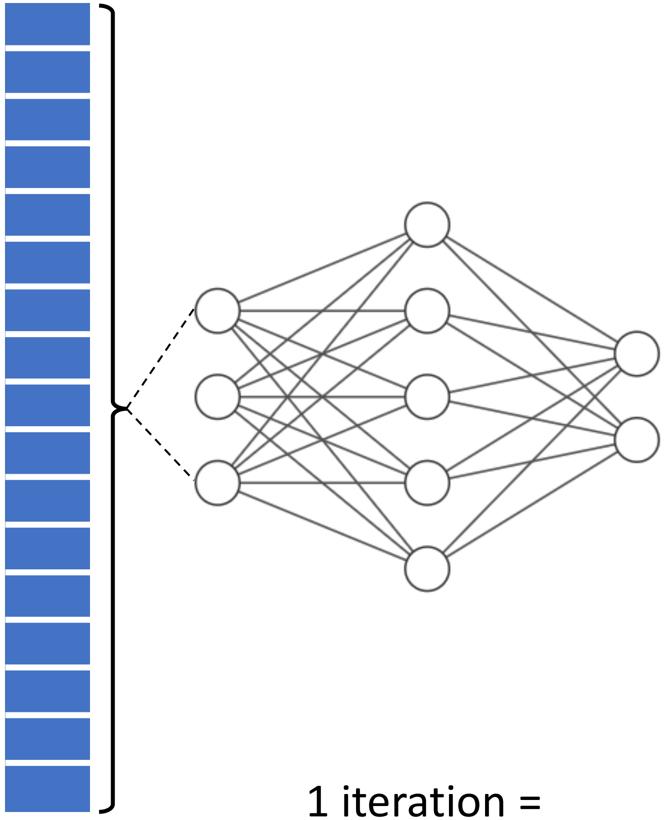
1 iteration =
Fwd/bwd pass 1 training sample

Mini-batch



1 iteration =
Fwd/bwd pass n-training samples
(n < total # of training samples)

Batch GD



1 iteration =
Fwd/bwd pass full training samples

Implementing Variants of Gradient Descent

SGD	Mini-batch	Batch GD
For epoch in range(epochs)	For epoch in range(epochs)	For epoch in range(epochs)
For sample in train	For mini-batch in train	Training Loop
Training loop • zero_grad • fwd pass input • compute loss • backpropagation • update weights/biases	Training Loop	

Total # of iterations	Epochs × m (m =total # of samples in training)	Epochs × R (R = m/mini-batch size)	Epochs

ADDITIONAL HYPER-PARAMETERS & REGULARIZERS

Activation Functions

Loss Functions and Advanced Optimizers

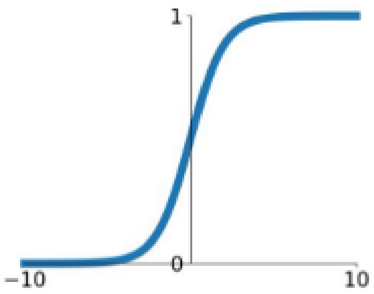
Regularizers

Batch Normalization

Network Initialization

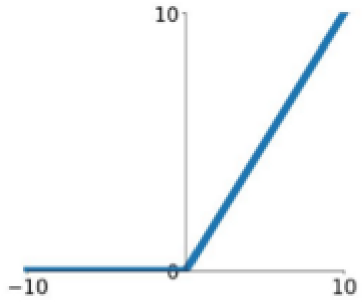
Activation Functions

Sigmoid



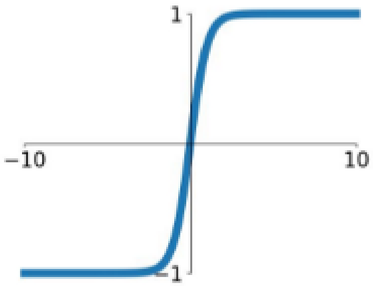
torch.nn.functional.sigmoid()

ReLU



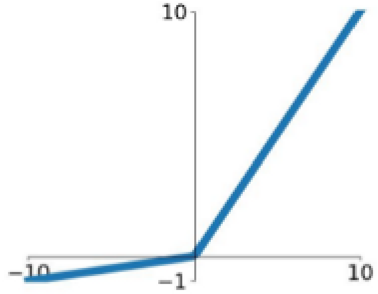
torch.nn.functional.relu()

Tanh



torch.nn.functional.tanh()

Leaky ReLU



torch.nn.functional.leaky_relu()

Activation Functions

```python
class Model(torch.nn.Module):

    def __init__(self, input_dim, output_dim):

        super(Model, self).__init__()

        self.layer1 = torch.nn.Linear(input_dim, 5)
        self.layer2 = torch.nn.Linear(5, 5)
        self.layer3 = torch.nn.Linear(5, output_dim)

    def forward(self, x):

        out1 = torch.nn.functional.relu(self.layer1(x))
        out2 = torch.nn.functional.relu(self.layer2(out1))
        output = torch.nn.functional.softmax(self.layer3(out2), dim=1)

        return output
```

Apply ReLU activation function to outputs of layer1 and 2

Loss Functions

Regression

Mean Squared Error $$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$ torch.nn.MSELoss()

Classification

Cross Entropy $$L_{\mathrm{CE}} = -\sum_{i=1}^{n} t_i \log(p_i)$$ torch.nn.CrossEntropyLoss()

NOTE: **torch.nn.CrossEntropyLoss()** automatically implements **one-hot encoding** and **softmax** when providing integer labels as targets.

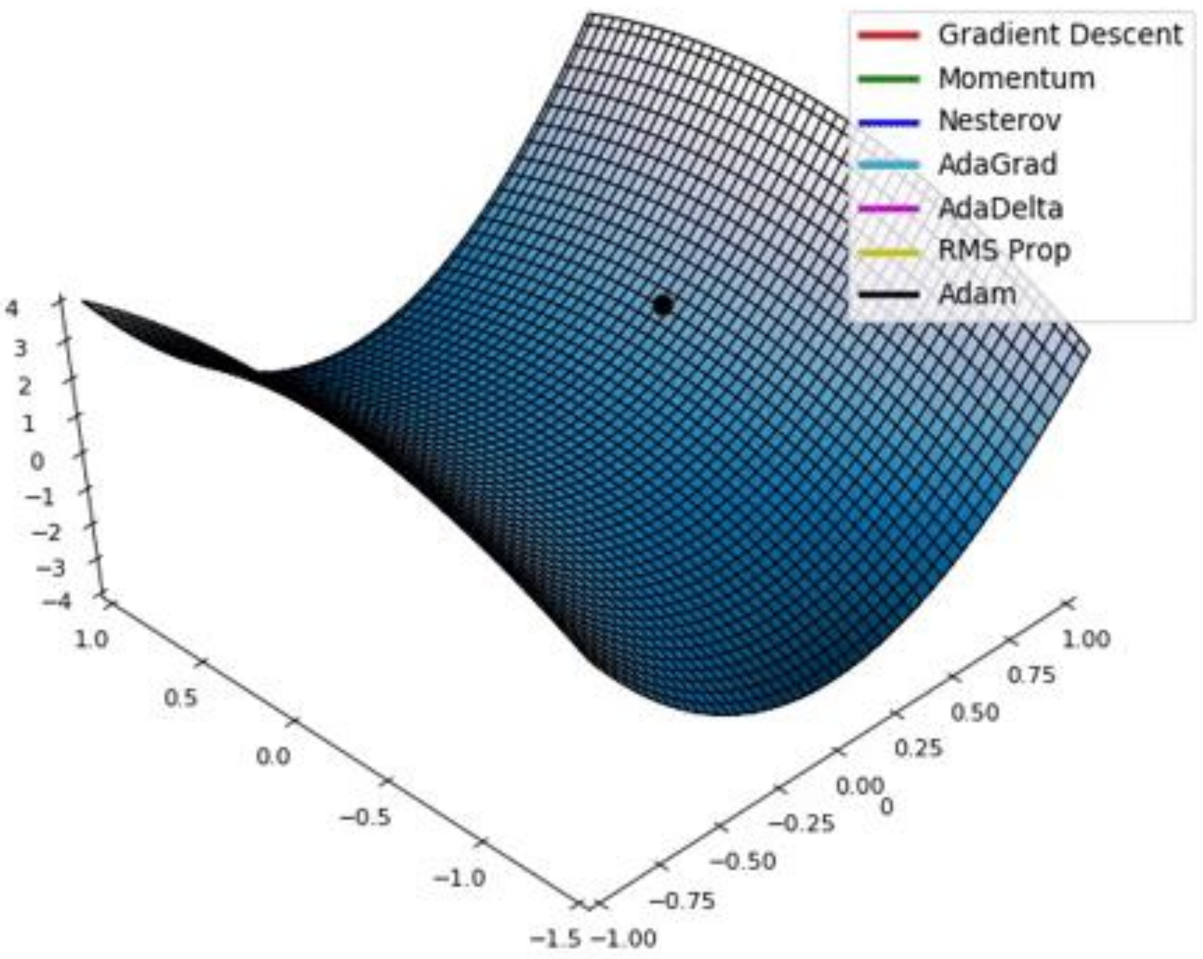
Advanced Optimizers

Gradient Descent
torch.optim.SGD()

Adam
torch.optim.Adam()

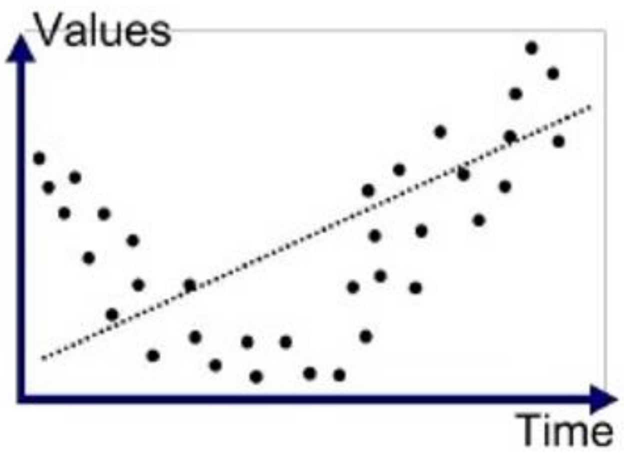
RMS Prop
torch.optim.RMSprop()

Ada Delta
torch.optim.Adadelta()



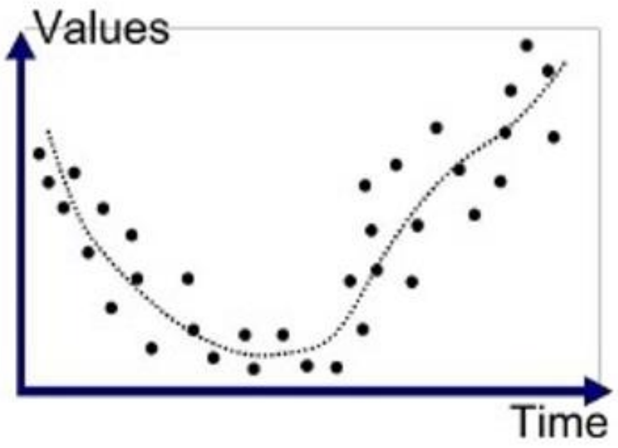
More Optimizers: https://pytorch.org/docs/stable/optim.html

Avoiding Overfitting with Regularization



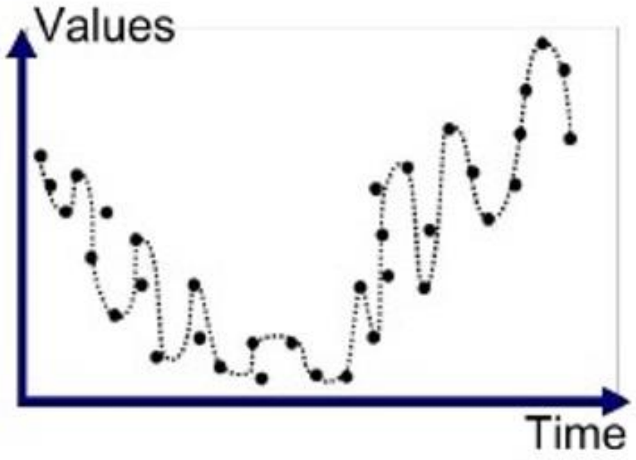
Underfitted

Bad training accuracy
Bad testing accuracy



Good Fit/Robust

Good training accuracy
Good testing accuracy



Overfitted

Great training accuracy
Bad testing accuracy

L1, L2 Regularizations in PyTorch

L1 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Penalizes sum of absolute values of weights

Results in a sparse model

Not suitable for learning complex patterns

Robust to outliers

L2 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Penalizes sum of squared values of weights

Results in a dense model

Learns complex patterns

Sensitive to outliers

L1, L2 Regularizations in PyTorch

L1

```python
l1_penalty = torch.nn.L1Loss(size_average=False)
reg_loss = 0

for param in model.parameters():

    reg_loss += l1_penalty(param)

lambda_ = 0.9

loss += lambda_ * reg_loss
```

Loop through model parameters to compute L1 regularization term

Pick the lambda value

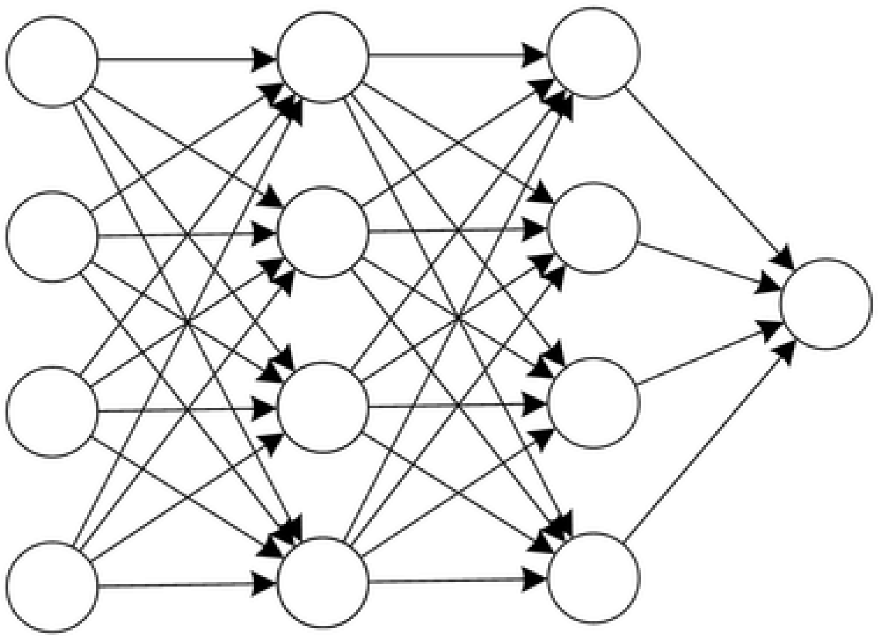
Add the L1 term to loss during training

L2

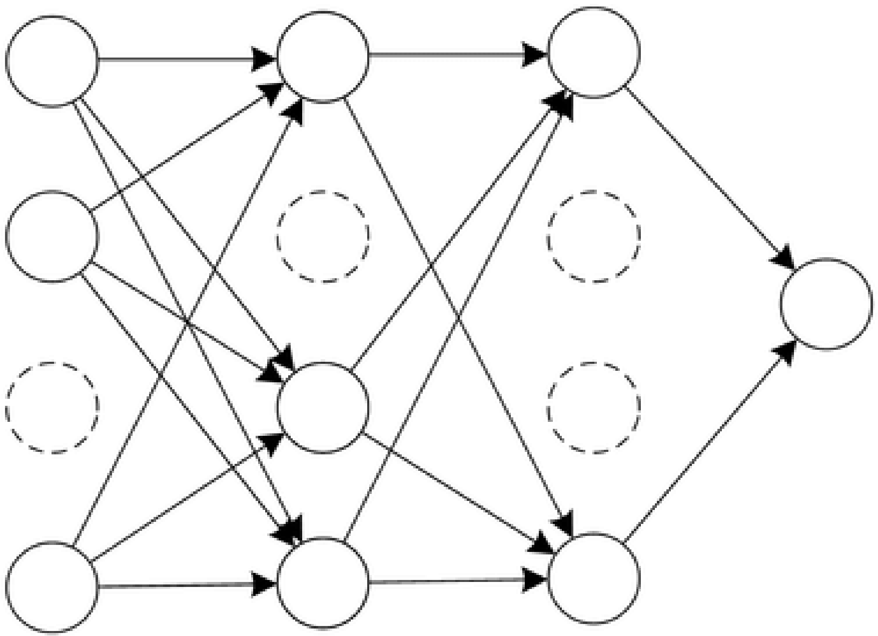
```python
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, weight_decay=0.9)
```

weight_decay sets lambda value for L2 regularization term

Dropout Regularization in PyTorch



Standard Neural Network



Network with Dropout

[Srivastava et al 2014 (~35000 citations!)]

Dropout forces the network to learn more robust features + different random subsets of other neurons

Dropout Regularization in PyTorch

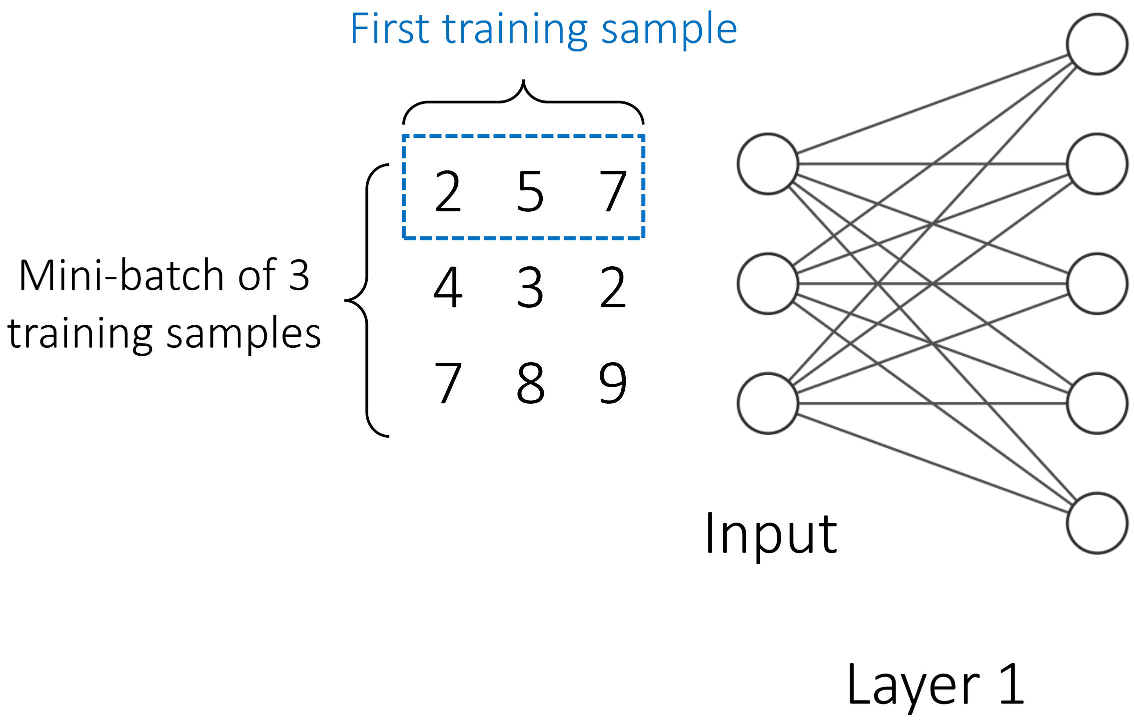
```python
class Model(torch.nn.Module):

    def __init__(self, input_dim, output_dim):

        super(Model, self).__init__()

        self.layer1 = torch.nn.Linear(input_dim, 5)
        self.layer2 = torch.nn.Linear(5, output_dim)
        self.dropout = torch.nn.Dropout(p = 0.25)

    def forward(self, x):

        out1 = self.layer1(x)
        out1 = self.dropout(out1)
        output = self.layer2(out1)

        return output
```

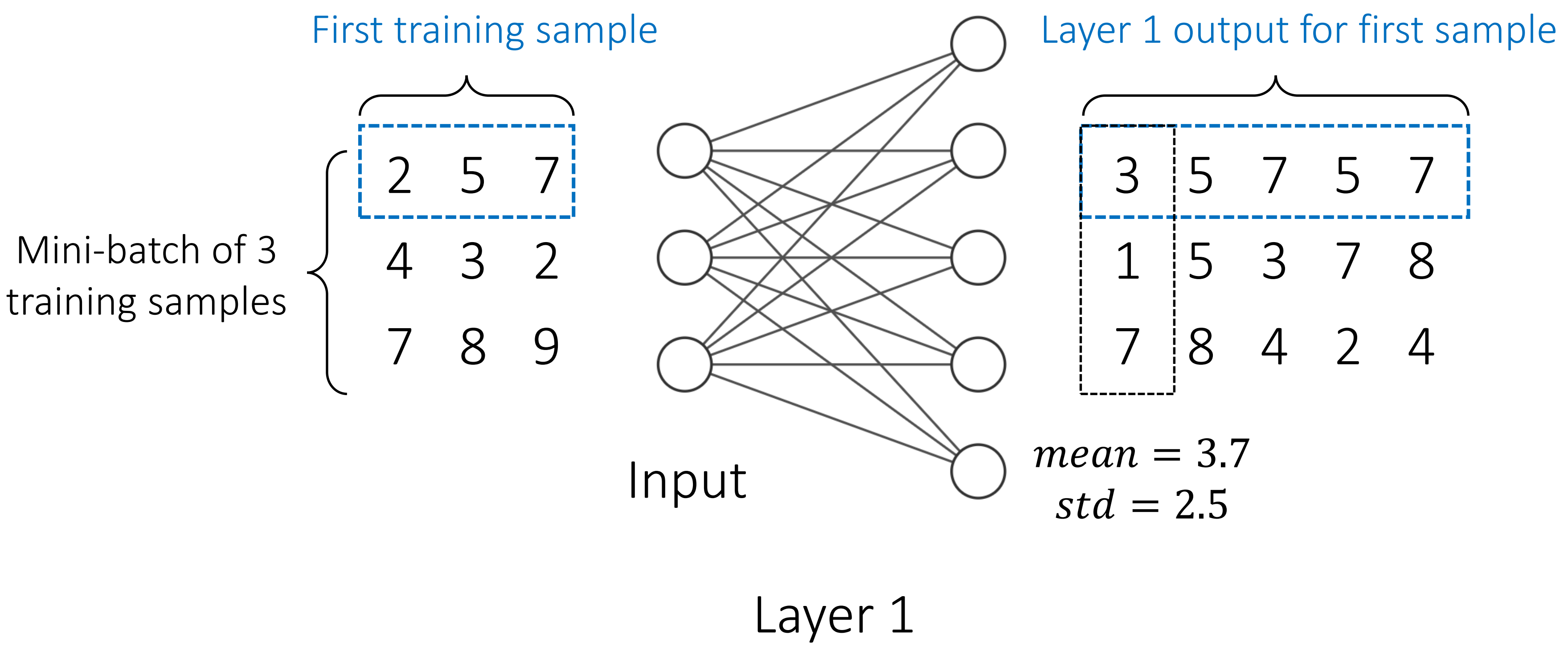
Add the dropout in __init__ with
p = probability of neuron state is set to 0

Apply dropout to the output of the desired layer

Preventing Vanishing/Exploding Gradients: Batch Normalization

[Loffe et al 2015 (>35000 citations)]

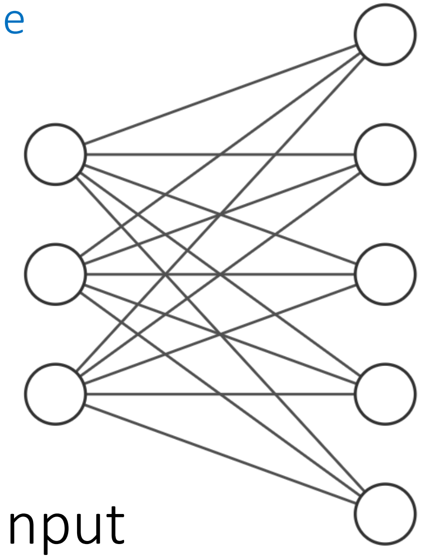



First training sample
Layer 1 output for first sample
2 5 7
4 3 2
7 8 9
Mini-batch of 3 training samples
3 5 7 5 7
1 5 3 7 8
7 8 4 2 4
$mean = 3.7$
$std = 2.5$
Input
Layer 1

First training sample

Mini-batch of 3 training samples

2	5	7
4	3	2
7	8	9



Input

Layer 1

Layer 1 output for first sample

3	5	7	5	7
1	5	3	7	8
7	8	4	2	4

$mean = 3.7$

$std = 2.5$

Normalized layer 1 output

-0.3	-0.7	1.4	0.2	0.4
-1.1	-0.7	-1.0	1.1	1.0
1.3	1.4	-0.4	-1.3	-1.4

$mean \approx 0$

$std \approx 1$

Batch normalization

Batch Normalization in PyTorch

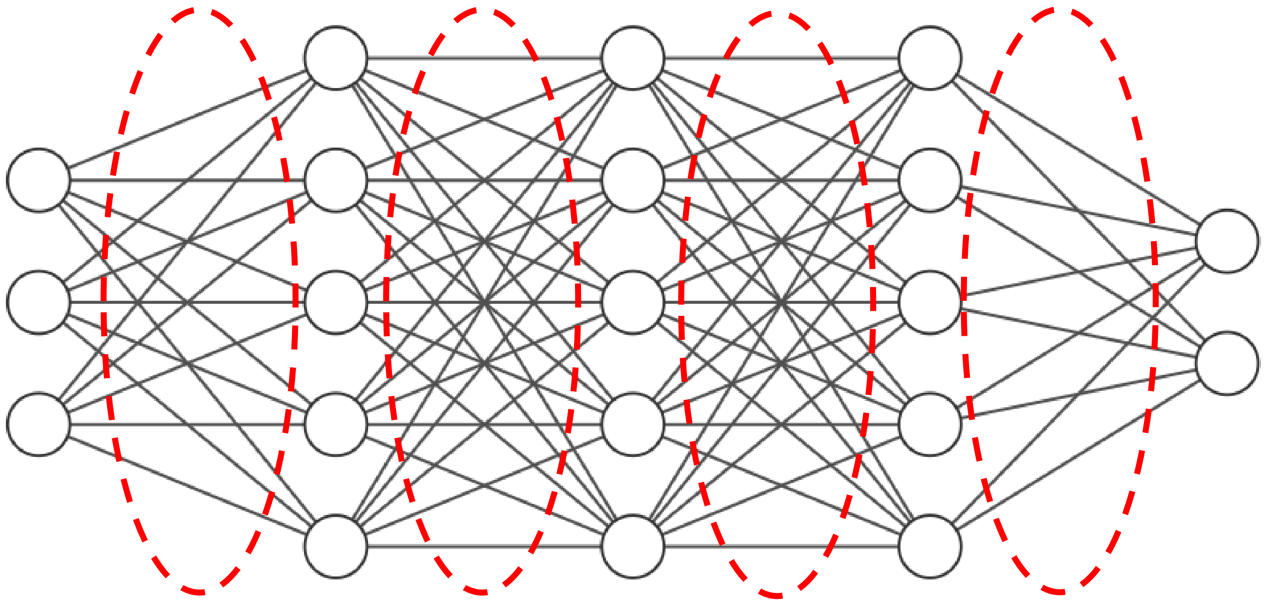
```
class Model(torch.nn.Module):

    def __init__(self, input_dim, output_dim):

        super(Model, self).__init__()

        self.layer1 = torch.nn.Linear(input_dim, 5)
        self.layer2 = torch.nn.Linear(5, output_dim)
        self.bn1 = torch.nn.BatchNorm1d(5)

    def forward(self, x):

        out1 = self.layer1(x)
        out1 = self.bn1(out1)
        out1 = torch.nn.functional.relu(out1)
        output = self.layer2(out1)

        return output
```

Add BatchNorm1D in __init__ with number of features in desired layer output

Apply batch normalization to the output of the desired layer.

NOTE: Batch normalization is done **BEFORE** feeding into activation function

Preventing Vanishing/Exploding Gradients: Weight Initialization



Proper weight initialization plays essential roles in preventing exploding/vanishing gradients

↓

Faster convergence

Weight Initialization Methods

Uniform	Uniform distribution	torch.nn.init.uniform_()
Normal	Gaussian distribution	torch.nn.init.normal_()
Xavier [Xavier et al 2010]	Suitable for tanh() activation	torch.nn.init.xavier_uniform_()
Kaiming [He et al 2015]	Suitable for ReLU() activation	torch.nn.init.kaiming_uniform_()

More initializations: https://pytorch.org/docs/stable/nn.init.html

Weight Initialization in PyTorch

```python
class Model(torch.nn.Module):

    def __init__(self, input_dim, output_dim):

        super(Model, self).__init__()

        self.layer1 = torch.nn.Linear(input_dim, 5)
        self.layer2 = torch.nn.Linear(5, output_dim)
        torch.nn.init.kaiming_uniform_(self.layer2.weight)

    def forward(self, x):

        out1 = torch.nn.functional.relu(self.layer1(x))
        output = torch.nn.functional.relu(self.layer2(out1))

        return output
```

Manually apply Kaiming He Initialization to layer2

NOTE: PyTorch already applies good default initialization for most layers.
We suggest using manual initialization for experimental purposes

TRAINING FULLY CONNECTED NETWORKS

Iris Classification Example

Neural Network Workflow in PyTorch

Prepare Data

Define Model

Select Hyperparameter

Identify Tracked Values

Train Model

Visualization and Evaluation

Prepare Data

```python
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

iris = load_iris()

X = iris['data']
y = iris['target']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
                                                    test_size=0.2,
                                                    random_state=2)

X_validation = X_train[:int(len(X_test))]
y_validation = y_train[:int(len(X_test))]

X_train = X_train[int(len(X_test)):]
y_train = y_train[int(len(X_test)):]
```

Testing data ratio (0.2 = 20%)

Random seed to use for splitting

Load Iris dataset

Extract features (X) and target labels (y)

Scale features using scikit-learn provided standard scaler

Split the dataset into Training (80%) and Testing (20%)

Assign subset of the training dataset as validation data (same size as testing)

Use the remaining dataset as training

Final split ratios = Training: 60%, Testing: 20%, Validation: 20%

Define Model

```python
class irisClassificationFCN(torch.nn.Module):

    def __init__(self, input_dim, output_dim, hidden1_dim, hidden2_dim):

        super(irisClassificationFCN, self).__init__()

        self.layer1 = torch.nn.Linear(input_dim, hidden1_dim)
        self.layer2 = torch.nn.Linear(hidden1_dim, hidden2_dim)
        self.layer3 = torch.nn.Linear(hidden2_dim, output_dim)

    def forward(self, x):

        out1 = torch.nn.functional.relu(self.layer1(x))
        out2 = torch.nn.functional.relu(self.layer2(out1))
        output = self.layer3(out2)

        return output
```

Input layer (input_dim = 4)

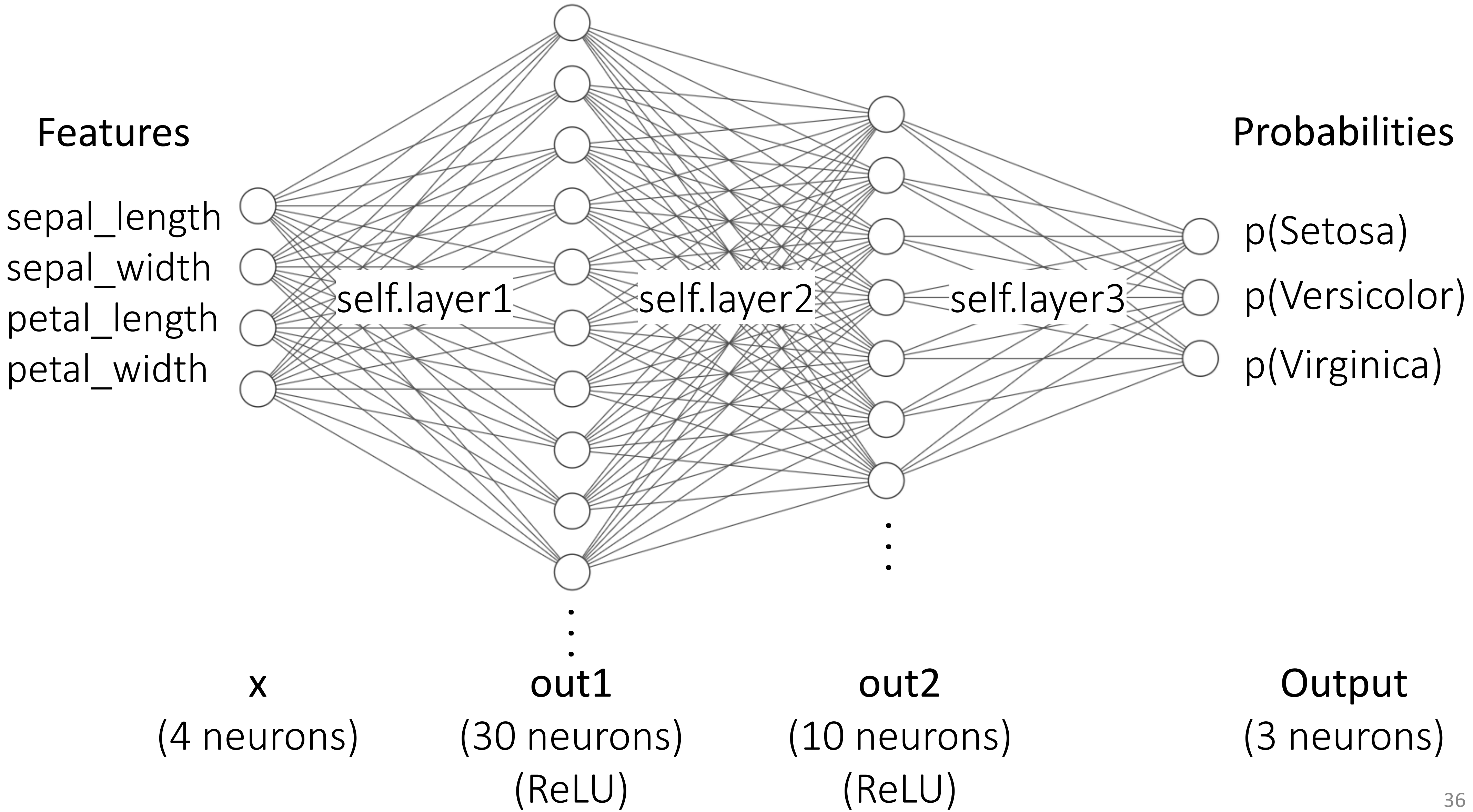
2 hidden layers (hidden1_dim = 30, hidden2_dim = 10)

output layer (output_dim = 3)

ReLU activation for the outputs of each hidden layer

Return raw final output

Define Model



Select Hyperparameter

```python
model = irisClassificationFCN(input_dim = 4, output_dim = 3,
                              hidden1_dim = 30, hidden2_dim = 10)

learning_rate = 0.025
epochs = 25

loss_func = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

model
```

```
irisClassificationFCN(
  (layer1): Linear(in_features=4, out_features=30, bias=True)
  (layer2): Linear(in_features=30, out_features=10, bias=True)
  (layer3): Linear(in_features=10, out_features=3, bias=True)
)
```

Initialize the model with **input_dim** = 4, **output_dim** = 3, **hidden1_dim** = 30, **hidden2_dim** = 10

Using learning rate of 0.025 and 25 as epochs

Using Cross Entropy Loss and Adam Optimizer

Model variable correctly shows our network structure

Identify Tracked Values

```
train_loss_list = np.zeros((epochs,))
validation_accuracy_list = np.zeros((epochs,))
```

Create empty list or NumPy arrays to hold training loss and validation accuracy

Train Model

```
import tqdm

train_inputs = torch.from_numpy(X_train).float()
train_targets = torch.from_numpy(y_train).long()

validation_inputs = torch.from_numpy(X_validation).float()
validation_targets = torch.from_numpy(y_validation).long()

testing_inputs = torch.from_numpy(X_test).float()
testing_targets = torch.from_numpy(y_test).long()
```

Import tqdm to visualize the progress of your training

Convert training/validation/testing datasets into PyTorch Tensors

Convert the targets into int64 form via .long()

Train Model

```python
# Training Loop ---------------------------------------------------

for epoch in tqdm.trange(epochs):

    optimizer.zero_grad()

    train_outputs = model(train_inputs)

    loss = loss_func(train_outputs, train_targets)

    train_loss_list[epoch] = loss.item()

    loss.backward()

    optimizer.step()

    # Compute Validation Accuracy ----------------------------

    with torch.no_grad():

        validation_outputs = model(validation_inputs)

        correct = (torch.argmax(validation_outputs, dim=1) ==
                   validation_targets).type(torch.FloatTensor)

        validation_accuracy_list[epoch] = correct.mean()
```

Training loop

- Empty gradient buffer
- Forward propagation
- Compute loss
- Save loss value to a list
- Backward propagation
- Update weights/biases

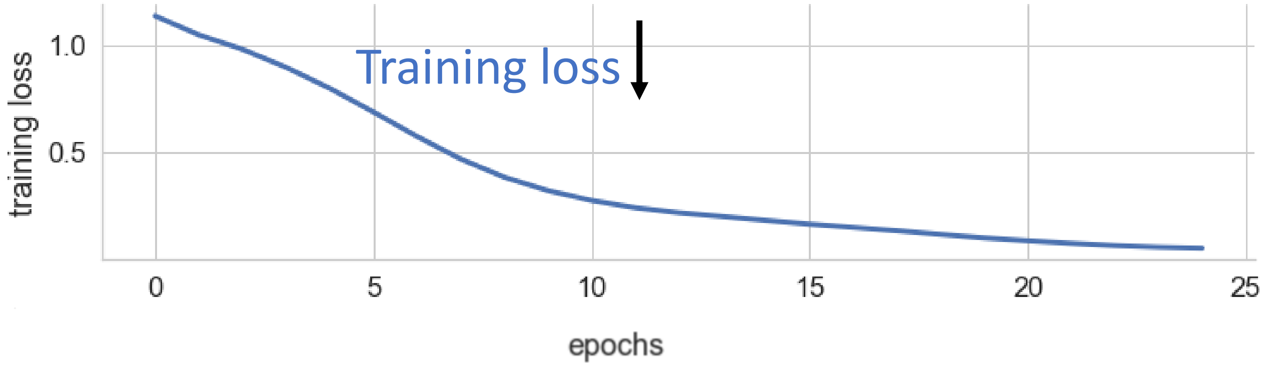
Compute Validation accuracy per epoch

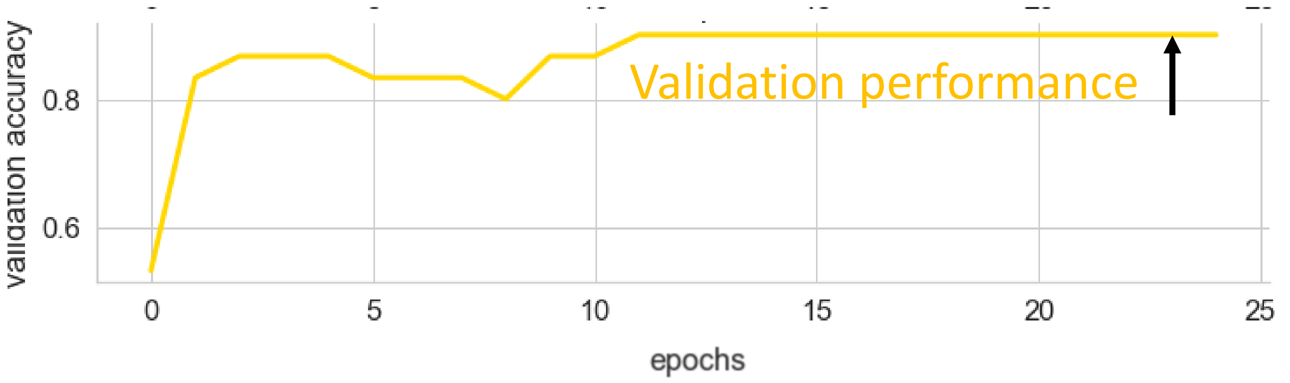
- Forward pass validation inputs to network
- Compare the outputs (index with the highest probability) against the validation target
- Compute and append the validation accuracy

Visualization and Evaluation

```python
plt.subplot(2, 1, 2)
plt.plot(train_loss_list, linewidth = 3)
plt.ylabel("training loss")
plt.xlabel("epochs")
sns.despine()
```

```python
plt.figure(figsize = (12, 6))

plt.subplot(2, 1, 1)
plt.plot(validation_accuracy_list, linewidth = 3, color = 'gold')
plt.ylabel("validation accuracy")
```






```python
with torch.no_grad():

    # Pass the testing feature data (30 samples) to the network to produce model predictions
    y_pred_test = model(testing_inputs)

    # Use the same technique as above to commpute the testing classification accuracy
    correct = (torch.argmax(y_pred_test, dim=1) == testing_targets).type(torch.FloatTensor)

    print("Testing Accuracy: " + str(correct.mean().numpy()*100) + '%')
```

```
Testing Accuracy: 93.33333373069763%
```

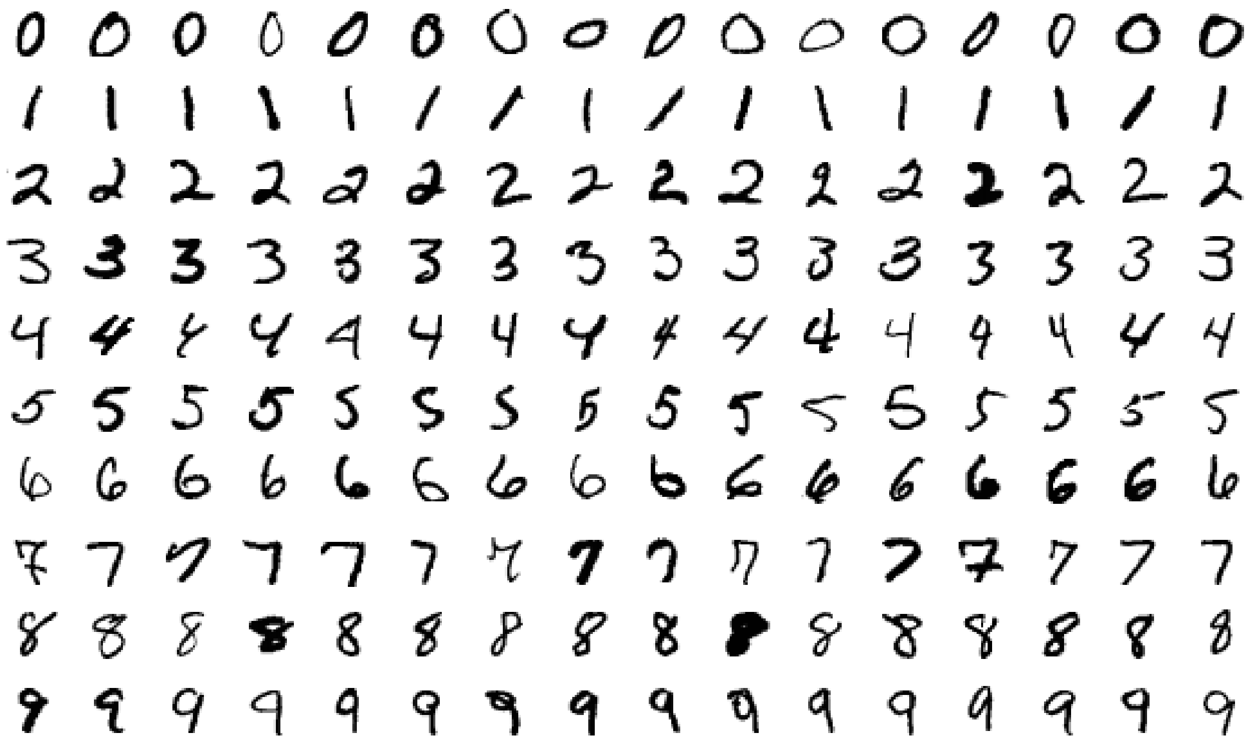
93% classification accuracy

Testing performance ↑

LAB 3 ASSIGNMENT:

MNIST Classification using Fully Connected Network

MNIST Dataset



Handwritten digits 0-9

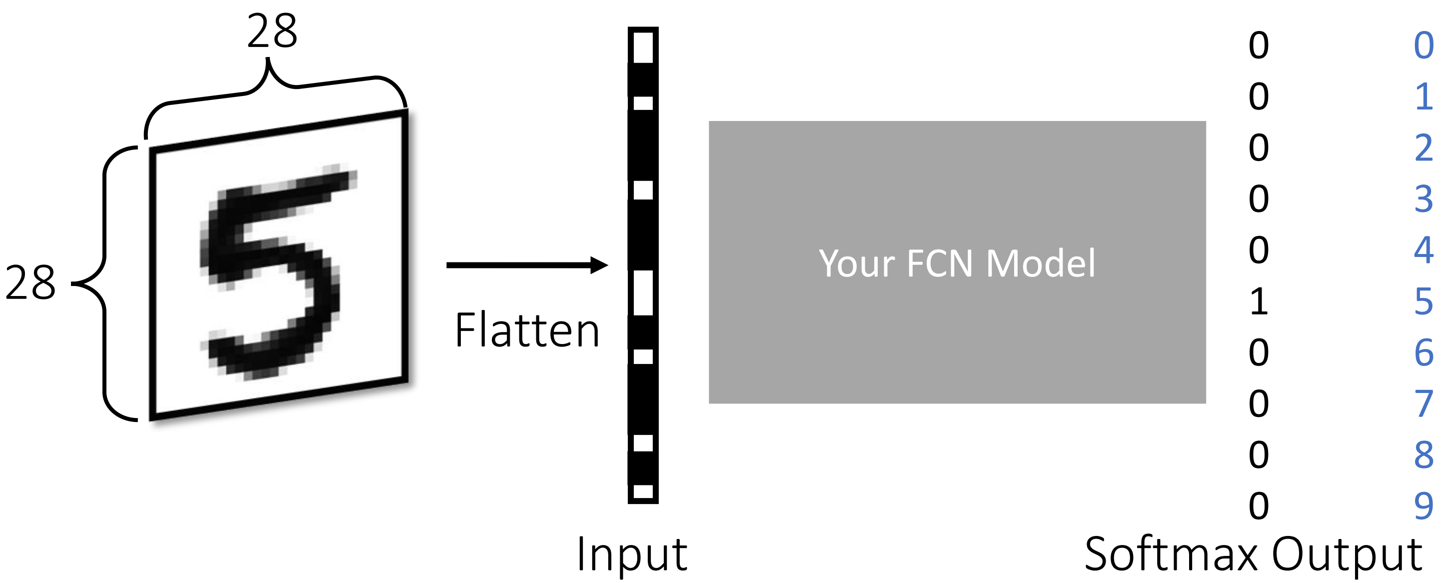
Target labels are the correct values of the digit

Data consists of grayscale images of fixed size (28x28) – flattens to 784

Canonical dataset for machine learning

1000 training samples, 100 testing samples

MNIST Classification with FCN



In this exercise, you will classify handwritten digits (28 x 28) using your own **Fully Connected Network Architecture.**

Prior to training your neural net, 1) Flatten each digit into 1D array of size 784, 2) Normalize the dataset using standard scaler and 3) Split the dataset into train/validation/test.

Design your own neural net architecture with your choice of hidden layers, activation functions, optimization method etc.

Your goal is to **achieve a testing accuracy of >90%**, with no restrictions on epochs.

Demonstrate the performance of your model via plotting the **training loss, validation accuracy** and printing out the **testing accuracy.**

Plot the testing samples where your model failed to classify correctly and print your model's best guess for each of them

Tips for Training Your Model

First things to decide

- Number of layers
- Neurons in each layer
- Activation function
  (ReLU, Tanh, sigmoid)
- Training batch size
  (SGD, Mini-batch, Batch Gradient)
- Learning rate
- Optimizer
  (SGD, Adam, RMS Prop etc)
- Number of training epochs

If your model is overfitting
(high training performance but low validation/testing performance)

- Add dropout layers
- Add regularization terms
- Stop training early
- Make network smaller (fewer layers or neurons)