Supervised Machine Learning and Learning Theory

Lecture 13: Implementation of Neural Networks in PyTorch

October 18, 2024

Review: Implementation in PyTorch

- Loading dependencies

Implement a convolutional neural network to recognize handwritten digits

Before you start, make sure to read the problem description in the handout pdf.

```
# Uncomment the below line and run to install required packages if you have not done so

# !pip install torch torchvision matplotlib tqdm
```

```
# Setup
import torch
import matplotlib.pyplot as plt
import torchvision
from torchvision import datasets, transforms
from tqdm import trange

%matplotlib inline
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

# Set random seed for reproducibility
seed = 1234
# cuDNN uses nondeterministic algorithms, set some options for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.manual_seed(seed)
```

Review: Loading dataset

Get MNIST Data

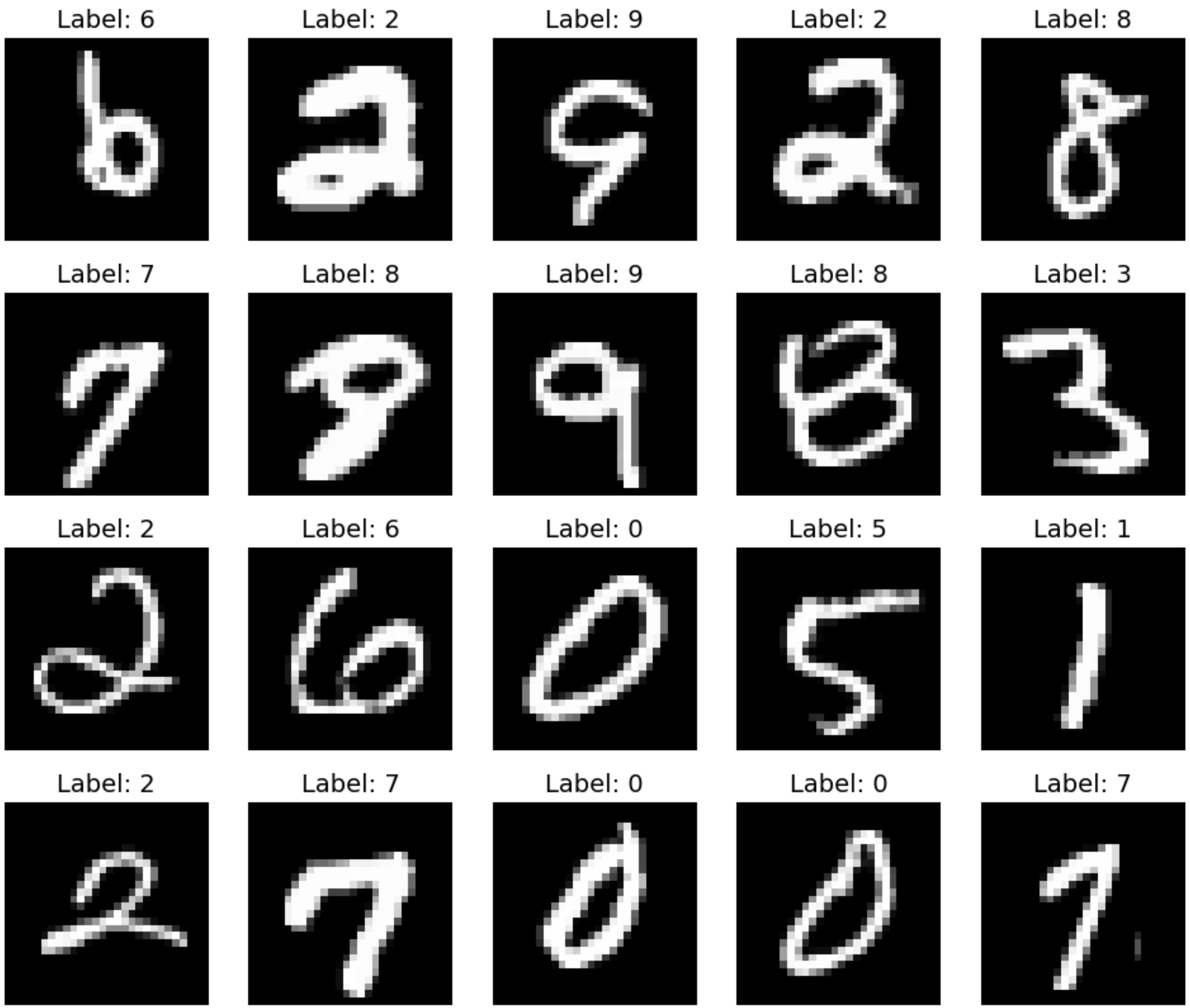
The `torchvision` package provides a wrapper to download MNIST data. The cell below downloads the training and test datasets and creates dataloaders for each.

```
# Initial transform (convert to PyTorch Tensor only)
transform = transforms.Compose([
    transforms.ToTensor(),
])
#torchvision.datasets.MNIST(root=root_dir,download=True)
root_dir = './data'
train_data = datasets.MNIST(root_dir, train=True, download=False, transform=transform)
test_data = datasets.MNIST(root_dir, train=False, download=False, transform=transform)

train_data.transform = transform
test_data.transform = transform

batch_size = 64
torch.manual_seed(seed)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True, num_workers=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=True)
```

Inspect dataset

```
dataiter = iter(train_loader)
images, labels = next(dataiter)

# Print information and statistics of the first batch of images
print("Images shape: ", images.shape)
print("Labels shape: ", labels.shape)
print(f'Mean={images.mean()}, Std={images.std()}')

fig = plt.figure(figsize=(12, 10))
for i in range(20):
    plt.subplot(4, 5, i+1)
    plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
    plt.title(f'Label: {labels[i]}', fontsize=14)
    plt.axis('off')
```

Review: Visualization



Review: Defining network architecture

Implement a two-layer neural network

Write a class that constructs a two-layer neural network as specified in the handout. The class consists of two methods, an initialization that sets up the architecture of the model, and a forward pass function given an input feature.

```
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = torch.nn.Sequential(
            torch.nn.Conv2d(
                in_channels=1,
                out_channels=10,
                kernel_size=5,
                stride=1,
                padding=0,
            ),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = torch.nn.Sequential(
            torch.nn.Conv2d(
                in_channels=10,
                out_channels=20,
                kernel_size=5,
                stride=1,
                padding=0,
            ),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
        )

        # fully connected layer, output 10 classes
        self.fc = torch.nn.Linear(320, 10)
        self.act = torch.nn.ReLU()
#         self.log_softmax = torch.nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        # print(x.shape)
        x = x.view(x.size(0), -1)
        # print(x.shape)
        x = self.fc(x)
#         x = self.log_softmax(x)
        y_output = x

        return y_output
```

1st convolution

2nd convolution

Fully-connected layer

Forward pass (in a couple of slides)

Review: Defining network architecture

```python
model = CNN().to(DEVICE)

# sanity check
print(model)
```

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Linear(in_features=320, out_features=10, bias=True)
  (act): ReLU()
)
```

Number of in-channels: This is one for MNIST, since the image is black-white

Number of out-channels: This is the number of filters at this layer

Review: Training procedure

Implement an optimizer to train the neural net model

Write a method called `train_one_epoch` that runs one step using the optimizer.

```
def train_one_epoch(train_loader, model, device, optimizer, log_interval, epoch):
    model.train()
    losses = []
    counter = []

    for i, (img, label) in enumerate(train_loader):
        img, label = img.to(device), label.to(device)

        # ------------------
        optimizer.zero_grad()
        output = model(img)
        criterion = torch.nn.CrossEntropyLoss()
        loss = criterion(output, label)

        loss.backward()
        optimizer.step()
        # ------------------

        # Record training loss every log_interval and keep counter of total training images seen
        if (i+1) % log_interval == 0:
            losses.append(loss.item())
            counter.append(
                (i * batch_size) + img.size(0) + epoch * len(train_loader.dataset))

    return losses, counter
```

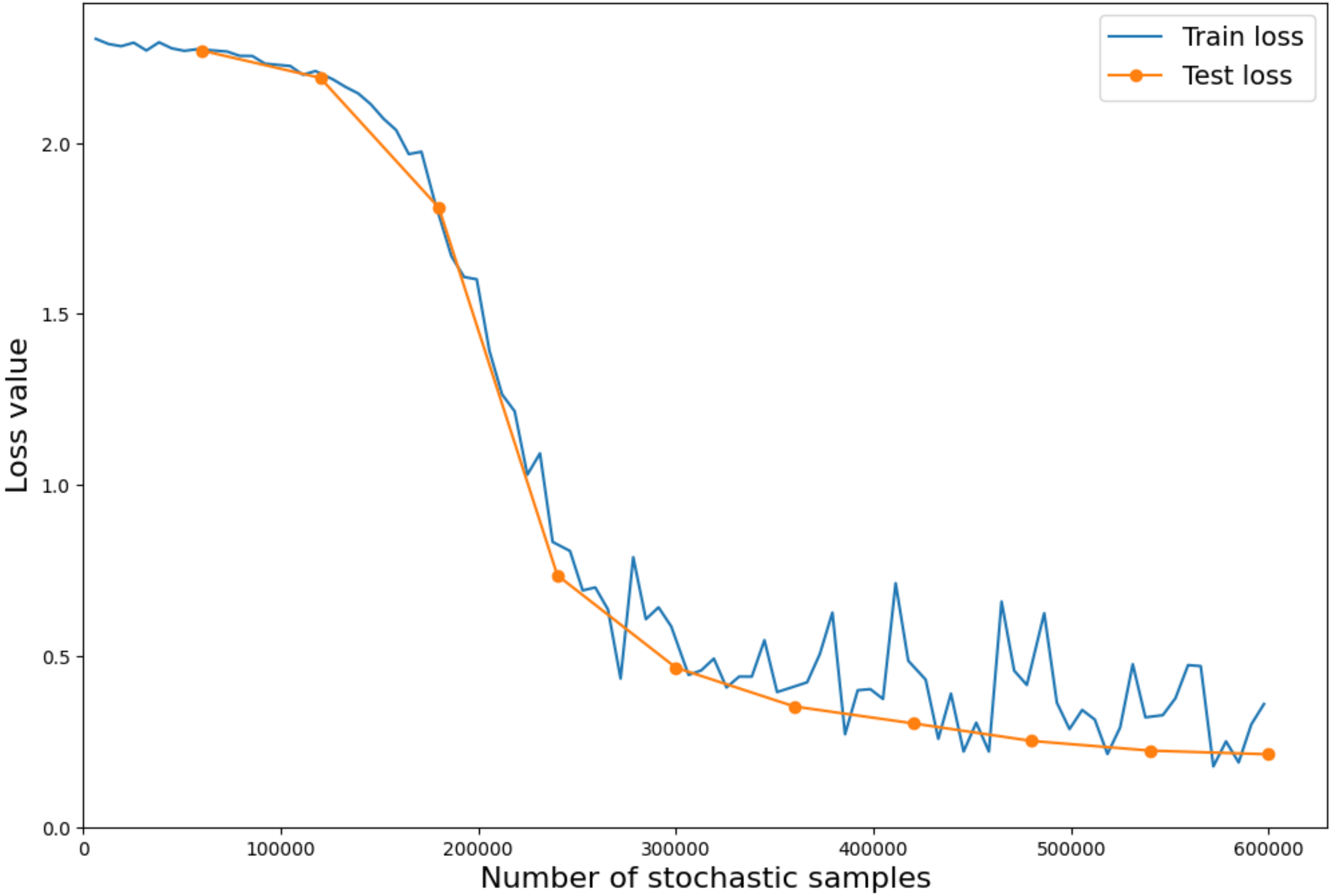
PyTorch implementation of SGD
(will elaborate in today's lecture)

Review: Test accuracy

```python
# Hyperparameters
lr = 0.001
max_epochs=10
gamma = 0.95

# Recording data
log_interval = 100

# Instantiate optimizer (model was created in previous cell)
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

# Use for CNN model
# optimizer = torch.optim.SGD(model.parameters(), lr=lr)

train_losses = []
train_counter = []
test_losses = []
test_correct = []
for epoch in trange(max_epochs, leave=True, desc='Epochs'):
    train_loss, counter = train_one_epoch(train_loader, model, DEVICE, optimizer, log_interval, epoch)
    test_loss, num_correct = test_one_epoch(test_loader, model, DEVICE)

    # Record results
    train_losses.extend(train_loss)
    train_counter.extend(counter)
    test_losses.append(test_loss)
    test_correct.append(num_correct)

    print(train_loss, test_loss, num_correct)

print(f"Test accuracy: {test_correct[-1]/len(test_loader.dataset)}")
```

```
67, 0.7128437161445618, 0.48613211512565613] tensor(0.3032) 8908

Epochs:  80%|████████████████████████████████████████              | 8/10 [01:19<0
0:19,  9.96s/it]

[0.4310001730918884, 0.2578464150428772, 0.390159547328949, 0.2206697016954422, 0.3051441013813019, 0.22070705890655518, 0.659205794334411
6, 0.4572473466396332, 0.41547641158103943] tensor(0.2518) 8995

Epochs:  90%|█████████████████████████████████████████████         | 9/10 [01:30<0
0:10, 10.06s/it]

[0.6253061890602112, 0.3636443614959717, 0.2863709330558777, 0.3423950672149658, 0.3142278790473938, 0.2135738581418991, 0.291072398424148
56, 0.47620293498039246, 0.3207015097141266] tensor(0.2235) 9060

Epochs: 100%|██████████████████████████████████████████████████████| 10/10 [01:40<0
0:00, 10.03s/it]

[0.32710516452789307, 0.376585990190506, 0.4734595715999603, 0.47056400775909424, 0.17729906737804413, 0.25048649311065674, 0.188878461718
55927, 0.30020228028297424, 0.3596789538860321] tensor(0.2127) 9115
Test accuracy: 0.9115
```

Review: Training and test loss curves

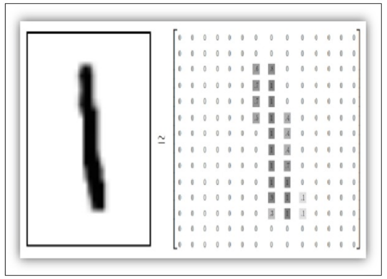


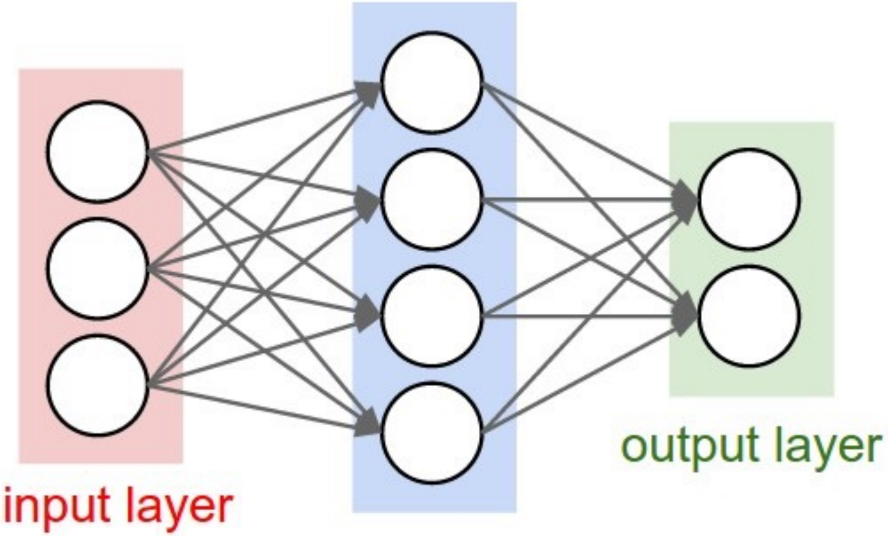
Lecture plan

- **The forward pass**

A single input

- **Forward pass: compute the output of a neural network given an input**





Prediction over {0,1,2,3,4,5,6,7,8,9}

Softmax output [0.01, **0.9**, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.01]

Loss: $-\log\frac{0.9}{1} = 0.045$

How do we get this output?

Forward pass: A single input

- Notations
  - **Input:** vector $x \in \mathbb{R}^{d_0}$ (e.g., $d_0 = 784$)
  - **First trainable layer:** weight matrix $w_1 \in \mathbb{R}^{d_0 \times d_1}$ (e.g., $d_1 = 100$), bias $b_1 \in \mathbb{R}^{d_1}$
  - **Activation function:** $\sigma: \mathbb{R} \to \mathbb{R}$
  - **Second trainable layer:** weight matrix $w_2 \in \mathbb{R}^{d_1 \times d_2}$ (e.g., $d_2 = 100$), bias $b_2 \in \mathbb{R}^{d_2}$

Illustration of forward pass

- First, apply matrix multiplication to get the input to the **hidden layer**: $x^\top w_1$, of size $1 \times d_1$

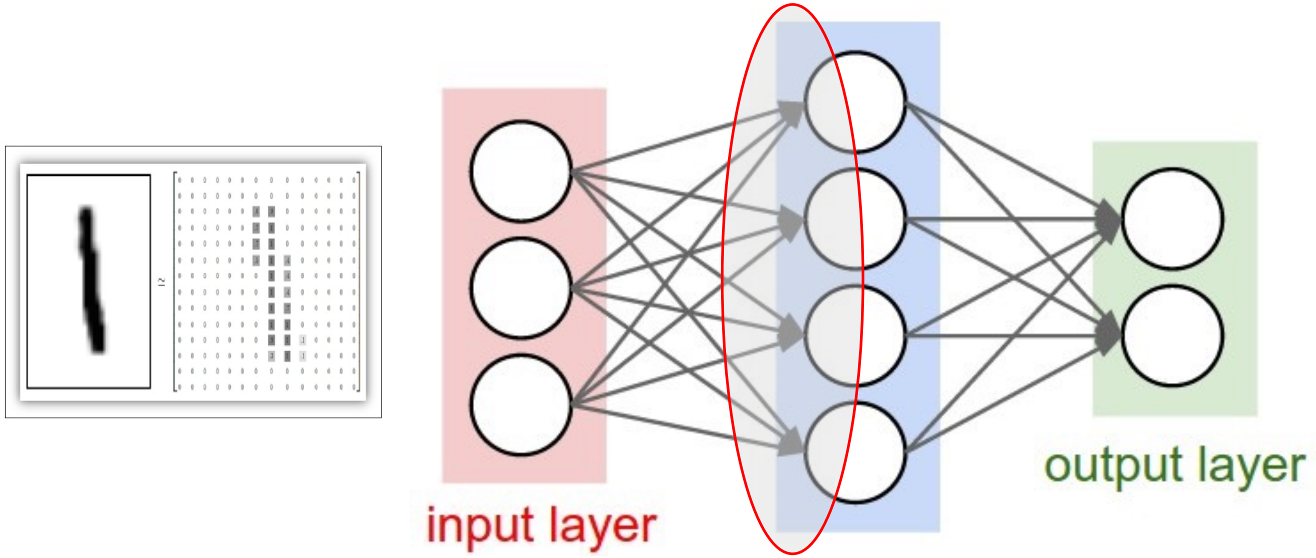


Illustration of forward pass

- Next, apply an activation function
  - Input to the hidden layer: $x^\top w_1$, size $1 \times d_1$
  - Output of the hidden layer: $\sigma(x^\top w_1)$, where $\sigma(\cdot)$ is applied entrywise to every coordinate of the input, size $1 \times d_1$

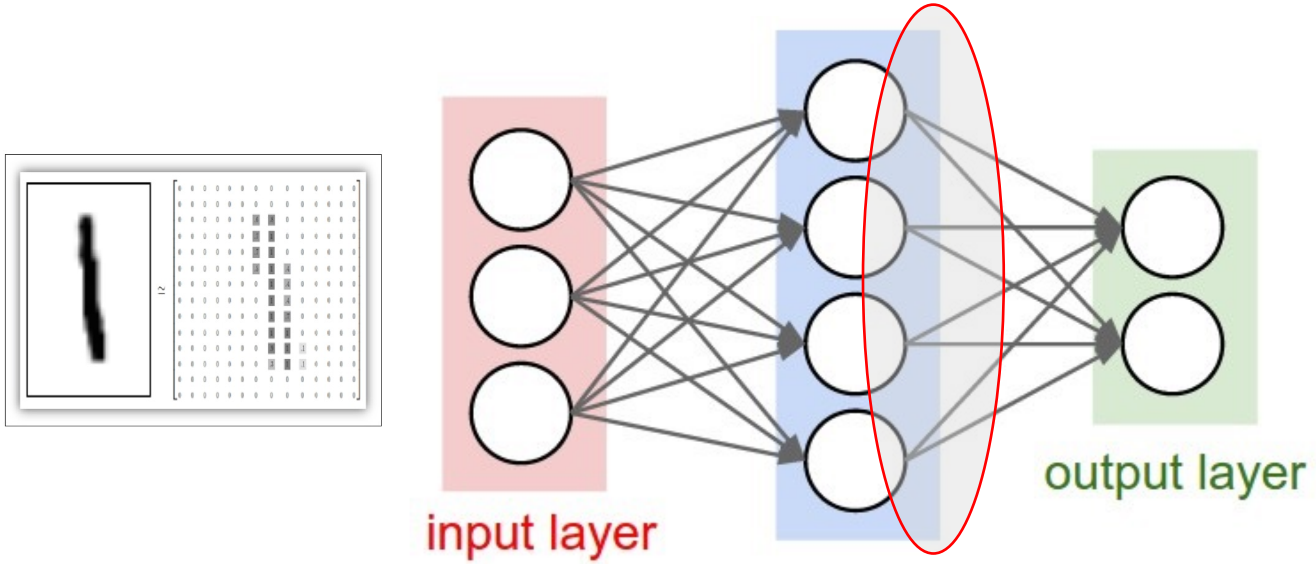


Illustration of forward pass

- Next, apply matrix multiplication again
  - Input to the hidden layer: $x^\top w_1$, size $1 \times d_1$
  - Output of the hidden layer: $\sigma(x^\top w_1)$, size $1 \times d_1$
  - Input to the output layer: $\sigma(x^\top w_1) w_2$, size $1 \times d_2$

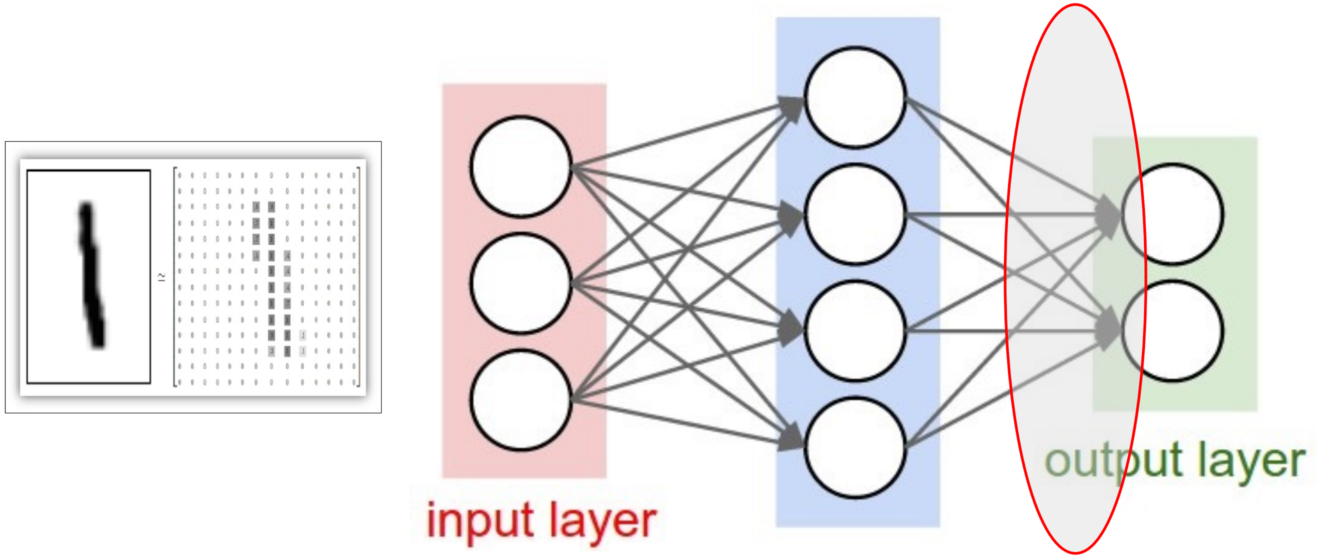
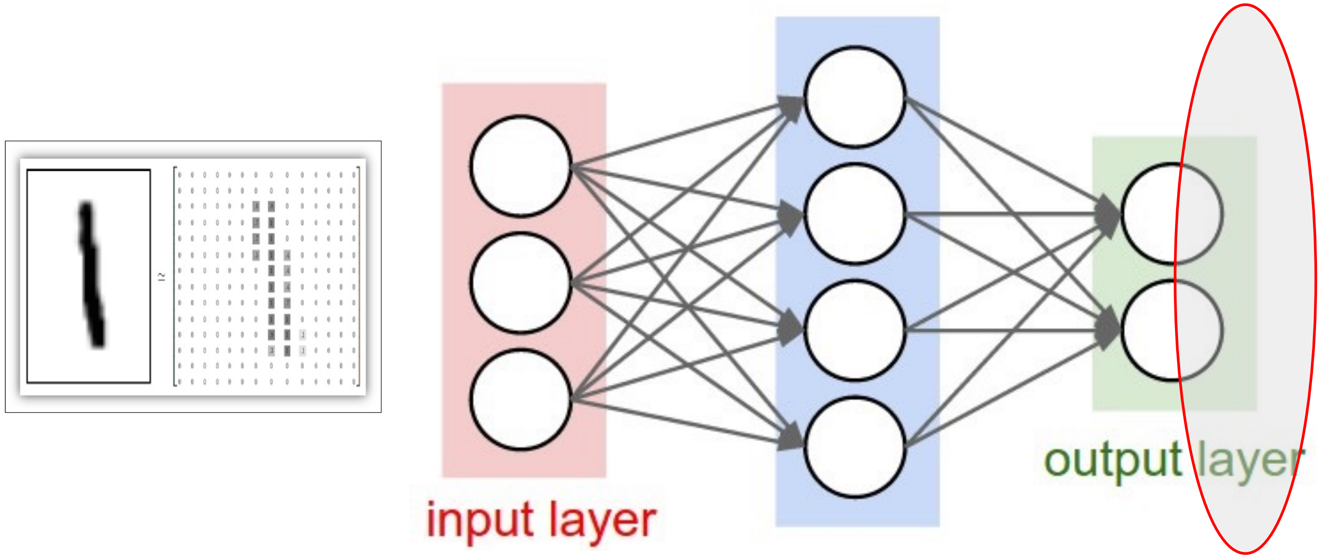


Illustration of forward pass

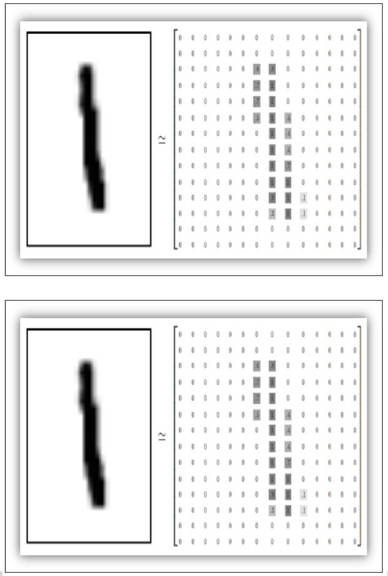
- Finally, apply softmax to get the probability distribution over ten output categories
  - Input to the hidden layer: $x^\top w_1$, size $1{\times}d_1$
  - Output of the hidden layer: $\sigma(x^\top w_1)$, size $1{\times}d_1$
  - Input to the output layer: $\sigma(x^\top w_1) w_2$, size $1{\times}d_2$
  - Final output: softmax($\sigma(x^\top w_1) w_2$)

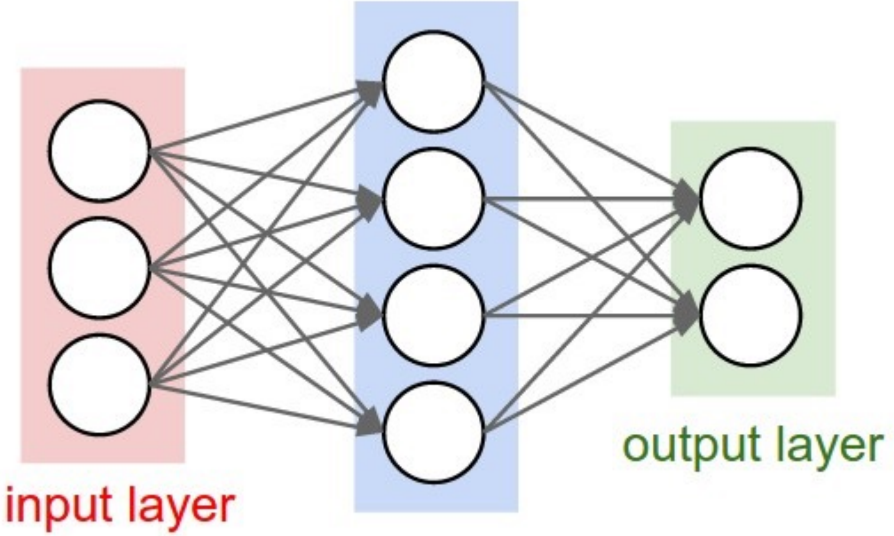


Softmax output [0.01, **0.9**, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.01]

Applying forward pass to a batch of inputs

- Repeat the steps again using matrix multiplication
  - Input: matrix $x \in \mathbb{R}^{B \times d_0}$ (e.g., $B = 128, d_0 = 784$)
  - First trainable layer → activation function → second trainable layer



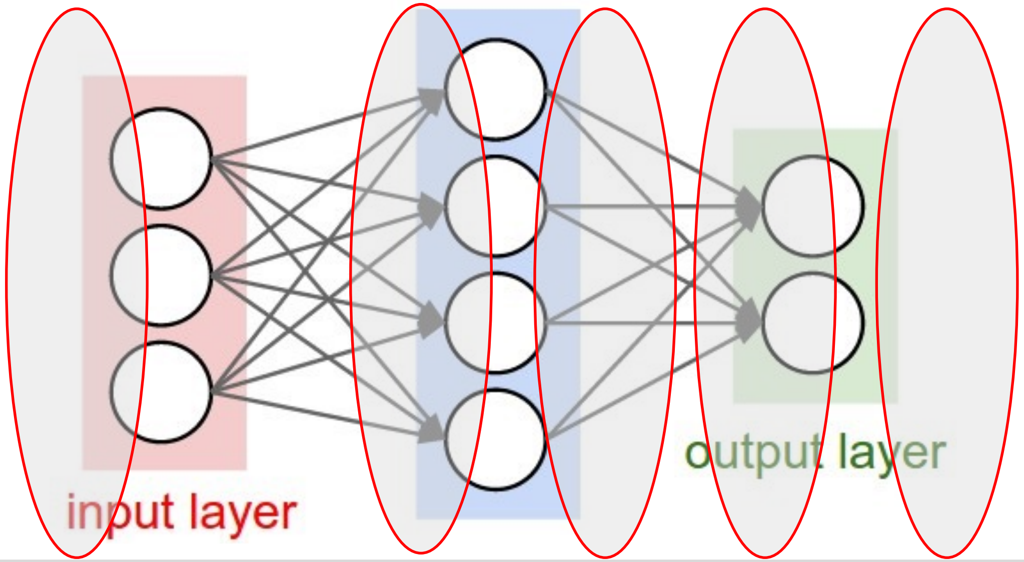


Softmax [0.01, **0.9**, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.01]

Softmax [0.01, **0.91**, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]

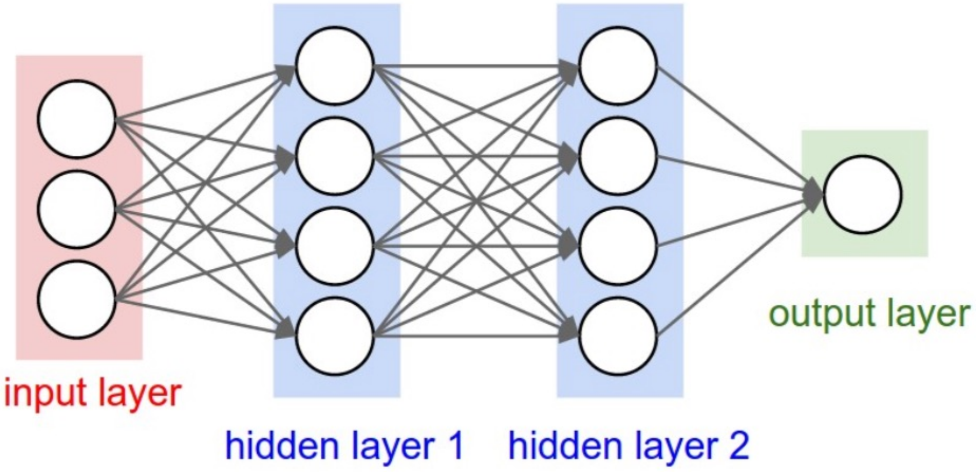
Applying forward pass to a batch of inputs

- Intermediate outputs
  - Input: x
  - Input to the hidden layer: $x^\top w_1$, size $B \times d_1$
  - Output of the hidden layer: $\sigma(x^\top w_1)$, size $B \times d_1$
  - Input to the output layer: $\sigma(x^\top w_1) w_2$, size $B \times d_2$
  - Final output: softmax($\sigma(x^\top w_1) w_2$)



Multiple layers

- Apply matrix multiplication followed by an activation function multiple times
  - Input: matrix $x \in \mathbb{R}^{d_0 \times B}$ (e.g., $B = 128, d_0 = 784$)
  - First trainable layer: weight matrix $w_1 \in \mathbb{R}^{d_0 \times d_1}$, bias $b_1 \in \mathbb{R}^{d_1}$
  - Activation function: $\sigma_1: \mathbb{R} \to \mathbb{R}$
  - …
  - i-th trainable layer: weight matrix $w_i \in \mathbb{R}^{d_{i-1} \times d_i}$, bias $b_i \in \mathbb{R}^{d_i}$
  - Activation function: $\sigma_i: \mathbb{R} \to \mathbb{R}$
  - …



Pseudocode for forward pass

- Input: $o_0 = x^\top$
- For $i = 1, 2, \ldots, L$
  - Input to layer i: $z_i = o_{i-1} w_i + b_i$
  - Output of layer i: $o_i = \sigma_i(z_i)$
- Return o_L

Lecture plan

- **PyTorch implementation of stochastic gradient descent**

Running example in PyTorch

- Calculate gradient via *backpropagation* (an efficient algorithm to compute the gradient---we'll cover this topic next lecture)

```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

Running example in PyTorch

- Update the weights: Based on a learning rate parameter, we apply stochastic gradient descent

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

Running example in PyTorch

- *Stochastic gradient descent* wrapped up in pytorch codes

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

Lecture plan

- **PyTorch implementation of linear/nonlinear classifiers**

Using neural networks for regression and classification

- Neural networks can be used to solve regression and classification problems

- We will consider a toy data setting for training a neural network in PyTorch

- We will use a linear classifier, then a nonlinear classifier, and compare their results

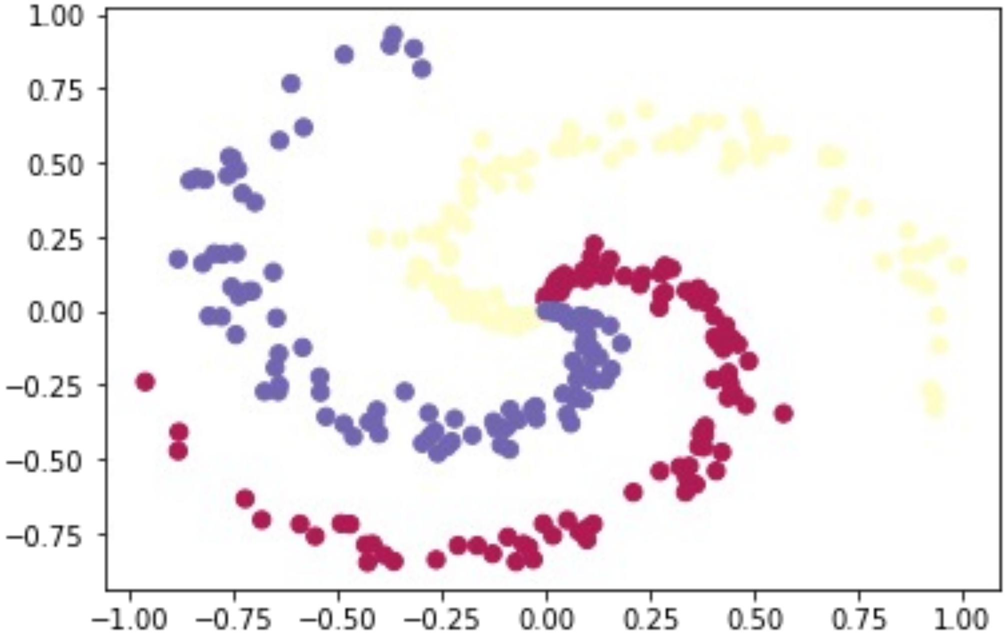
Generating data

- Generate a two-dimensional dataset with nonlinear decision boundaries

generating some data

In [2]:
```
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels

for j in range(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
  y[ix] = j

# lets visualize the data:
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.show()
```

Visualization



Initialization

- **Initialization:** Every entry of W is drawn from a standard Gaussian with mean zero and variance one
  - D: input dimension
  - K: number of classes
  - W: classifier parameters

Initialize the parameters

In [3]:
```
# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))

step_size = 1e-0
reg = 1e-3
```

Matrix multiplication

- X: dimension 300×2
- W: dimension 2×3
- b: dimension 1×3

Compute the output

In [4]:
```
# compute class scores for a linear classifier
scores = X @ W + b
```

Adds b into every row of X @ W (means matrix multiplication in numpy)

Loss function

- **Training loss:** Averaged cross-entropy loss plus an ℓ_2 penalty

- **Averaged cross-entropy loss** (average over training dataset)
  - Given a prediction for every label $y \in \{1,2,\ldots,K\}$, let u be this vector
  - $\ell(u) = -\log \frac{\exp(u_y)}{\sum_{i=1}^{K} \exp(u_i)}$ (Fact: $\ell(u) \geq 0$)

- **ℓ_2 penalty:** Sum of squared values of W and b

Cross-entropy loss

```python
num_examples = X.shape[0]
# get unnormalized probabilities
exp_scores = np.exp(scores)
# normalize them for each example
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

correct_logprobs = -np.log(probs[range(num_examples),y])
```

ℓ_2 penalty

```
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
```

Compute gradient

- Notations
  - u_k: output for label k
  - p_k: softmax probability for label k
  - $\ell(W, b)$: cross-entropy loss
- **Chain rule:** $\frac{\partial \ell(W,b)}{\partial W} = \frac{\partial \ell}{\partial u} \cdot \frac{\partial u}{\partial W}$ (next lecture)
- **Claim:** $\frac{\partial \ell}{\partial u_k} = p_k - 1_{y=k}, \frac{\partial u}{\partial W} = X^\top$ (next lecture)

Compute the analytic gradient

```
In [8]: dscores = probs
        dscores[range(num_examples),y] -= 1
        dscores /= num_examples

        dW = X.T @ dscores
        db = np.sum(dscores, axis=0, keepdims=True)
        dW += reg*W # don't forget the regularization gradient
```

Gradient on bias adds up all the log probabilities

Compute the gradient

- Gradient of ℓ_2 penalty (weight decay)

Compute the analytic gradient

In [8]:

```
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

dW = X.T @ dscores
db = np.sum(dscores, axis=0, keepdims=True)
dW += reg*W # don't forget the regularization gradient
```

Trainng loss

```
iteration 10: loss 0.9134056496088602
iteration 20: loss 0.8323889971607258
iteration 30: loss 0.7955967913635283
iteration 40: loss 0.7762634535759677
iteration 50: loss 0.7651042787584552
iteration 60: loss 0.7582423095449976
iteration 70: loss 0.7538293272190891
iteration 80: loss 0.7508959335854734
iteration 90: loss 0.7488963644108956
iteration 100: loss 0.7475063136555101
iteration 110: loss 0.7465247676838905
iteration 120: loss 0.7458228704214372
iteration 130: loss 0.7453157377782931
iteration 140: loss 0.7449461859000616
iteration 150: loss 0.7446749691022985
iteration 160: loss 0.744474730614621
iteration 170: loss 0.7443261494995304
iteration 180: loss 0.7442154278913563
iteration 190: loss 0.7441326186704039
iteration 200: loss 0.7440704927051738
```

This is quite high for three classes:

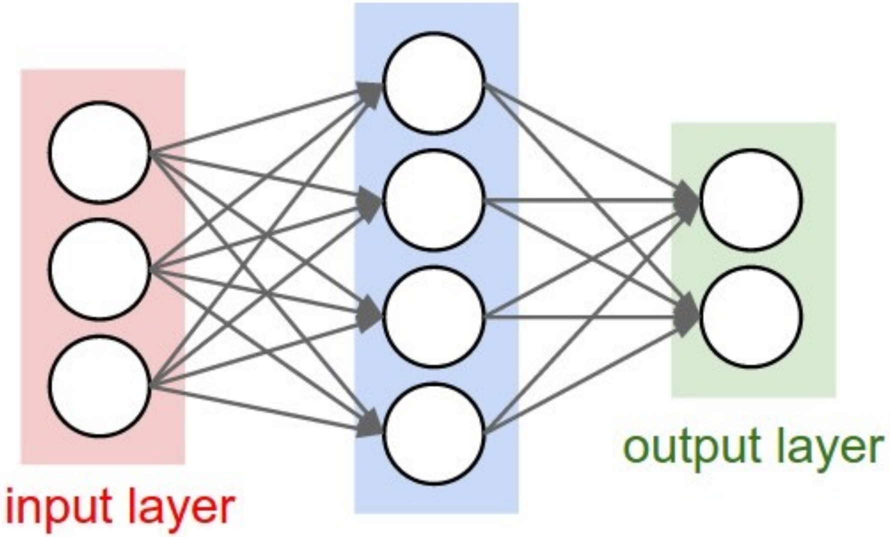
$$-\log\frac{1}{3} = 1.10$$

Can we do better?

- First trainable layer: weight matrix $w_1 \in \mathbb{R}^{D \times h}$, bias $b_1 \in \mathbb{R}^h$
- **Add activation function:** $\sigma: \mathbb{R} \to \mathbb{R}$
- **Add a second trainable layer:** weight matrix $w_2 \in \mathbb{R}^{h \times K}$, bias $b_2 \in \mathbb{R}^K$

Initialize the parameters

In [3]:
```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))

step_size = 1e-0
reg = 1e-3
```



Compute output

- Rectified linear units (ReLU): $\sigma(z) = \max(z, 0)$

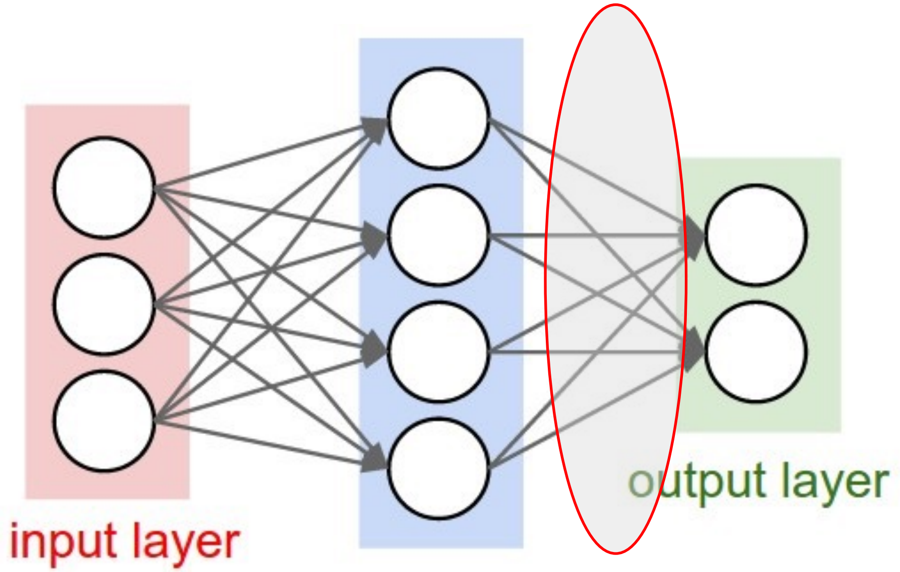
$$u = \sigma(XW_1 + 1 \cdot b_1)W_2 + 1 \cdot b_2$$

Compute the output

```
In [4]: # evaluate class scores with a 2-layer Neural Network
        hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
        scores = hidden_layer @ W2 + b2
```

Compute gradient

- **Gradient of the second layer:** Similar to the linear layer case since it is only for the cross-entropy loss. **Treat the hidden layer output as input**
- Gradient of the first layer



Compute the analytic gradient

In [6]:
```
# backpropate the gradient to the parameters
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# first backprop into parameters W2 and b2
dW2 = hidden_layer.T @ dscores
db2 = np.sum(dscores, axis=0, keepdims=True)

dhidden = dscores @ W2.T

# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0

# finally into W,b
dW = X.T @ dhidden
db = np.sum(dhidden, axis=0, keepdims=True)
```

Training results

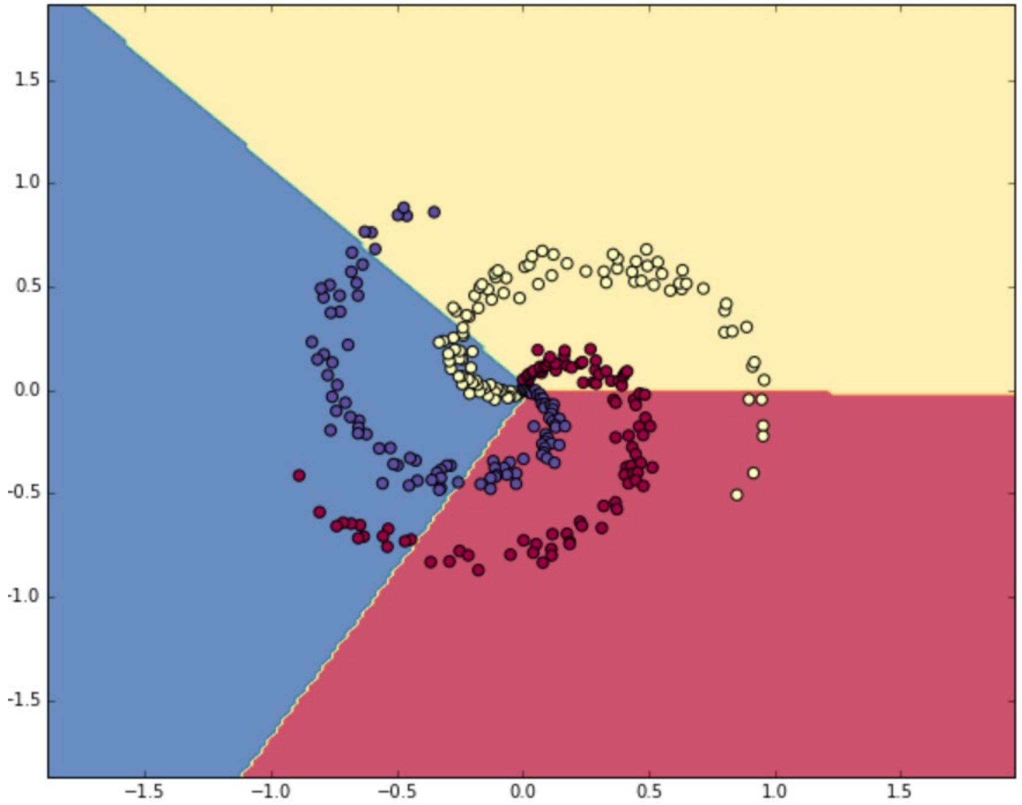
- Comparing loss between linear classifier (left) and **ReLU classifier** (**right**)

```
iteration 10: loss 0.9134056496088602
iteration 20: loss 0.8323889971607258
iteration 30: loss 0.7955967913635283
iteration 40: loss 0.7762634535759677
iteration 50: loss 0.7651042787584552
iteration 60: loss 0.7582423095449976
iteration 70: loss 0.7538293272190891
iteration 80: loss 0.7508959335854734
iteration 90: loss 0.7488963644108956
iteration 100: loss 0.7475063136555101
iteration 110: loss 0.7465247676838905
iteration 120: loss 0.7458228704214372
iteration 130: loss 0.7453157377782931
iteration 140: loss 0.7449461859000616
iteration 150: loss 0.7446749691022985
iteration 160: loss 0.744474730614621
iteration 170: loss 0.7443261494995304
iteration 180: loss 0.7442154278913563
iteration 190: loss 0.7441326186704039
iteration 200: loss 0.7440704927051738
```

```
iteration 1000: loss 0.40454021503681153
iteration 2000: loss 0.26346369806692593
iteration 3000: loss 0.25607811374045586
iteration 4000: loss 0.25410664245334263
iteration 5000: loss 0.2526010149171124
iteration 6000: loss 0.25198089929407874
iteration 7000: loss 0.25155952434511186
iteration 8000: loss 0.2512825150552082
iteration 9000: loss 0.2511044228402025
iteration 10000: loss 0.2509892383094693
```

Visualization

- Visualizing decision boundaries



Announcements

- Instructions for the course project will be released this afternoon