Introduction to Artificial Intelligence

Lecture 6: Neural networks with PyTorch implementation

September 22, 2025

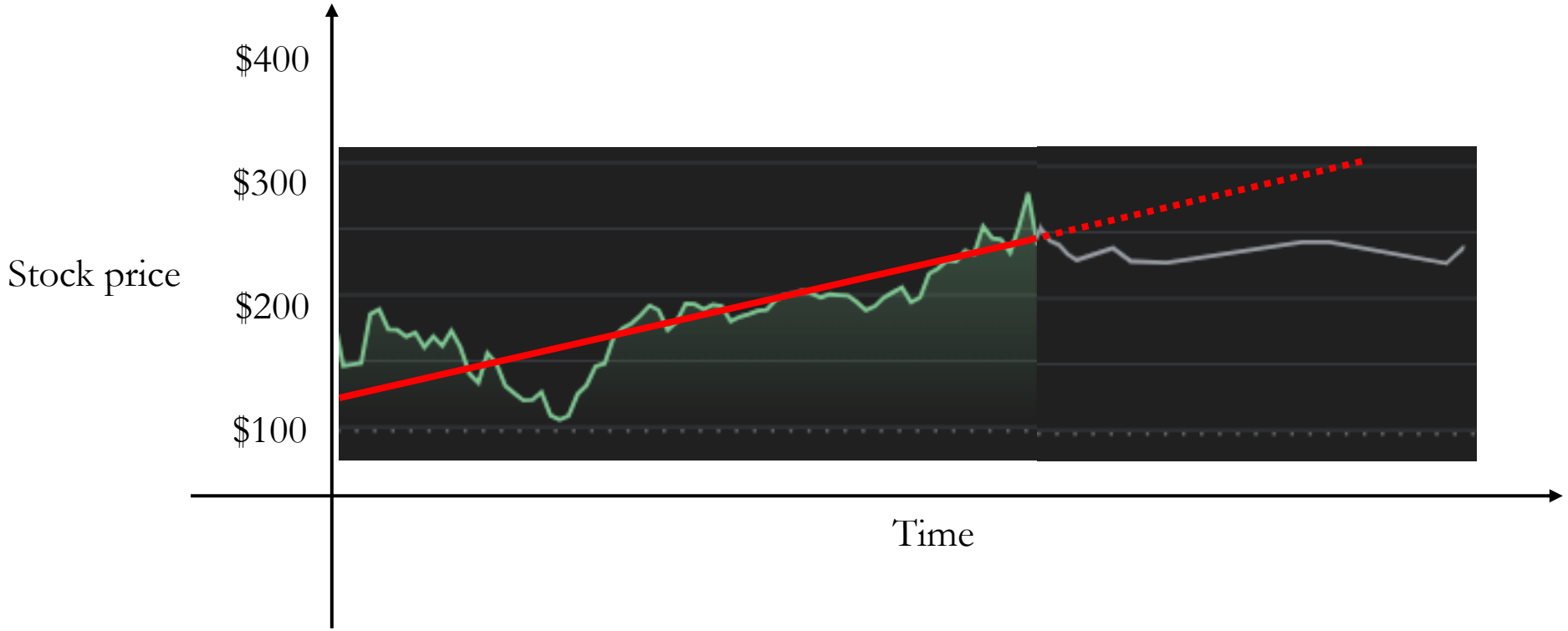
What machine learning can and cannot do

Input (A)	Output (B)	Application
Email	Spam? (0/1)	Spam filtering
Audio	Text transcripts	Speech recognition
English	Chinese	Machine translation
Ad, user info	Click? (0/1)	Online advertising
Image, radar info	Position of other cars	Self-driving cars
Image of phone	Defect? (0/1)	Visual inspection
Sequence of words	The next word	Chatbot

- Simple tasks (mapping A to B) for which humans can easily do (e.g., in a few seconds)

What machine learning can and cannot do

- Predicting the stock market price

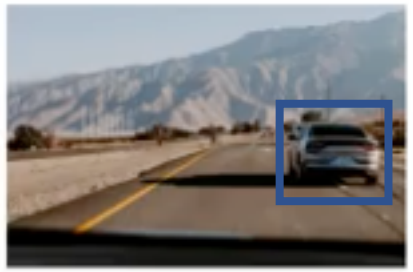


What makes an ML problem easier

- Learning a “simple” concept: relatively easy task for a human
- Lots of training data: mapping A to B

Self-driving car

Can do



Cannot do

Stop

Hitchhiker

Biker turning left

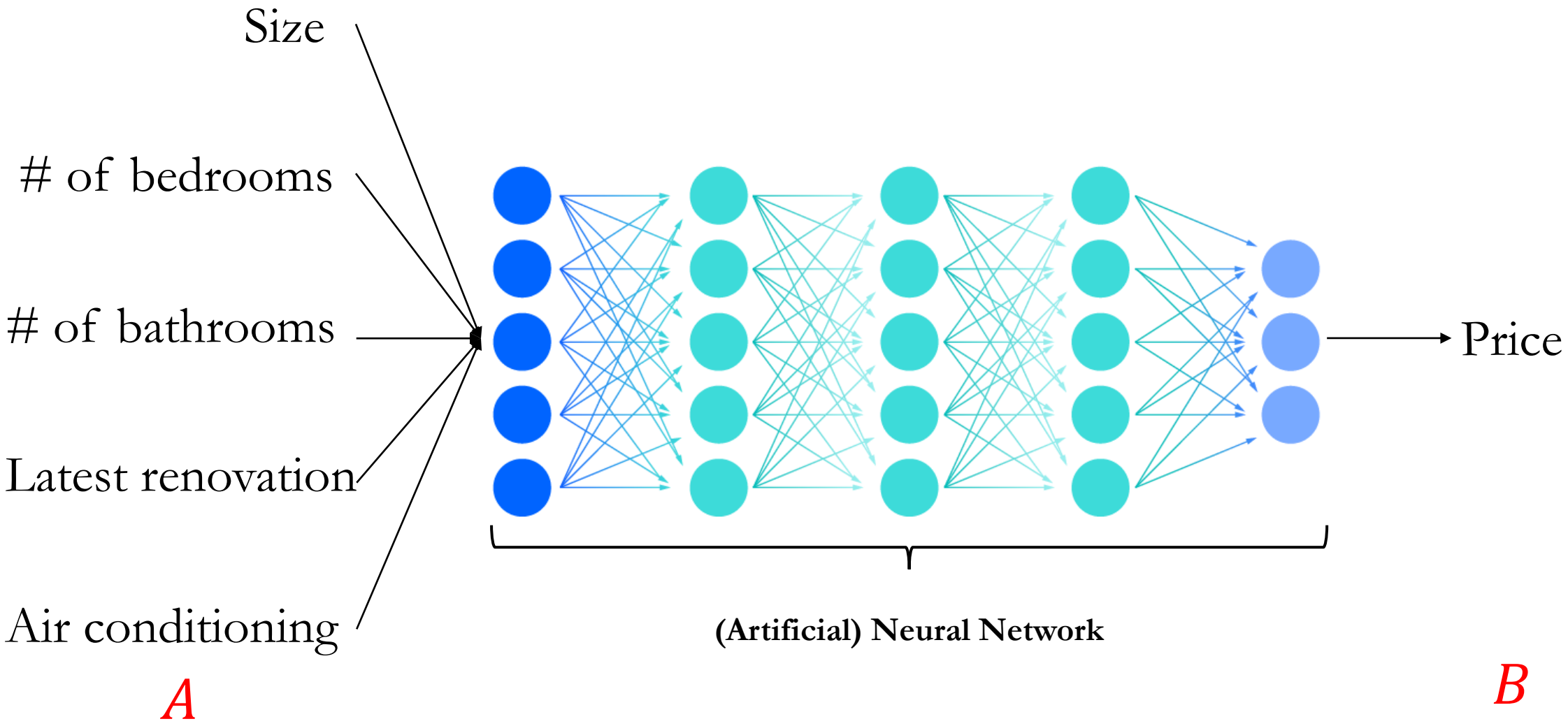
1. Difficult to collect enough data
2. All possible hand gestures are very large
3. Need high accuracy

Strengths and weaknesses of machine learning

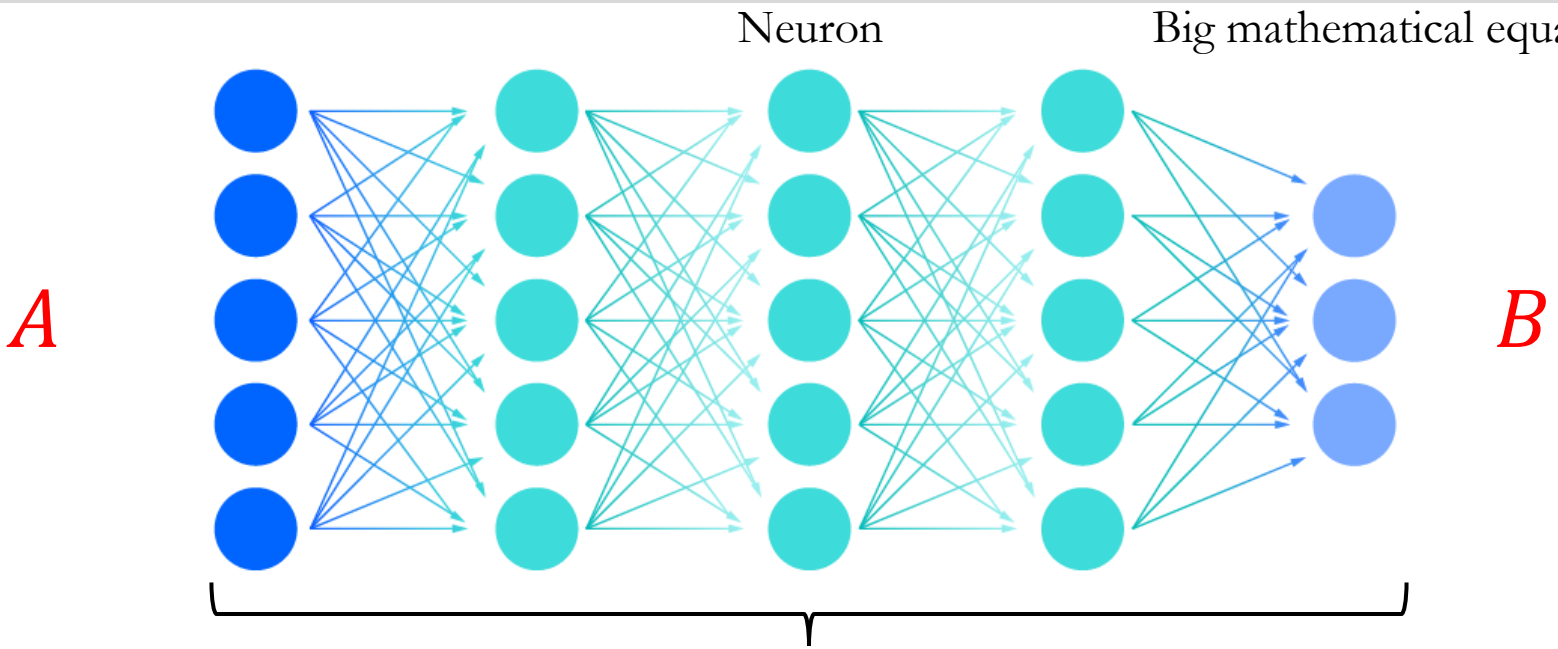
- ML tends to work well when: learning a "simple" concept, and there is a lots of training data

- ML tends to work poorly when: learning concepts from small data, and perform on new types of data

Deep learning



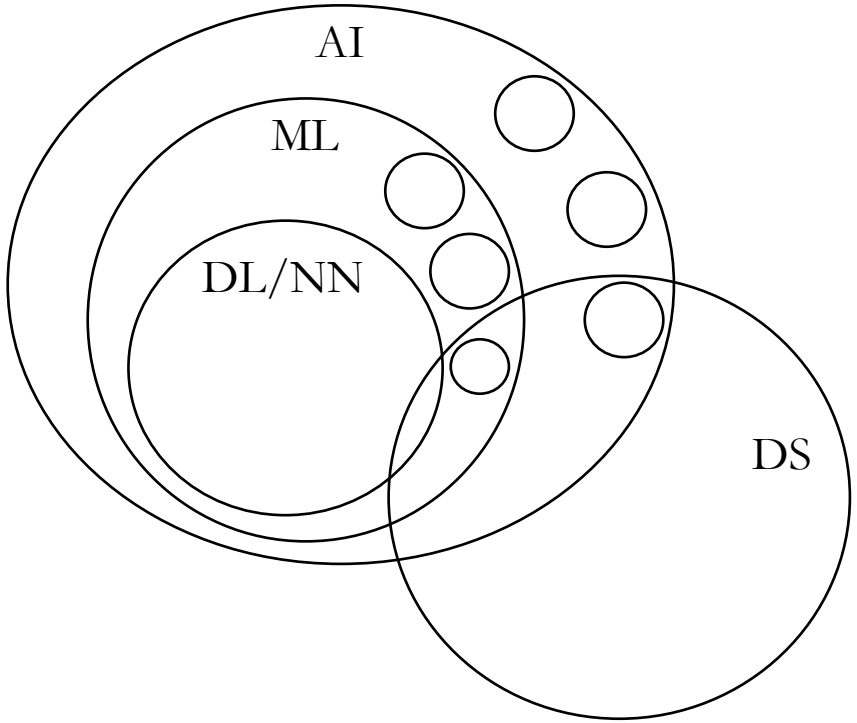
Deep learning



- Deep learning: a much better sounding brand, but essentially interchangeable with neural networks
- Neural networks were originally inspired by the brain, but the details of how they work are almost completely unrelated to how biological brains work

AI tools

- **Machine learning and data science**



- **Deep learning / neural networks**

- Other tools: Generative AI, unsupervised learning, reinforcement learning, graphical models, planning, knowledge graphs, reasoning, …

Developing machine learning systems

1. Problem formulation: "help a user find all photos that match a specific term, such as Paris"

   - What parts of the problem can be solved by machine learning?
   - Learn a function that maps a photo to a set of labels; then, given a label as a query, retrieve all photos with that label

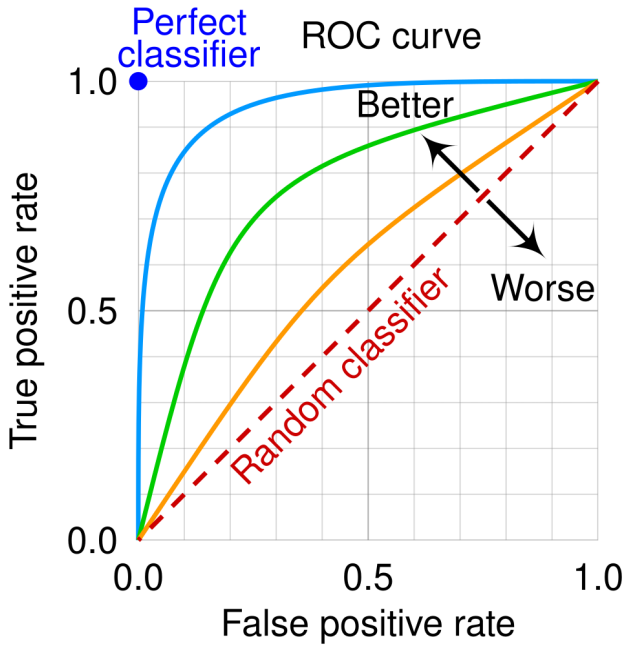
   - Types of learning problems: supervised, unsupervised, or reinforcement learning
     - Semi-supervised
     - Weakly supervised

Developing machine learning systems

2. Data collection, assessment, and management
   - ImageNet: over 14 million photos with ~20,000 different labels
   - Crowdsourcing: data annotators team (xAI, AI trainer)
   - Transfer learning: use data from another related problem
   - When data are limited: data augmentation helps
   - Unbalanced classes
   - Outliers

Developing machine learning systems

3. Model selection and training
   - Choosing a model class
   - Choosing hyperparameters
   - Evaluation metrics: false positive rate (imagine building a system to classify spam email), receiver operating characteristic (ROC) curve, confusion matrix



		Predicted condition	
	Total 8 + 4 = 12	Cancer 7	Non-cancer 5
Actual condition	Cancer 8	6	2
	Non-cancer 4	1	3

Interpretability and explainability

- **Interpretability**: an ML model is interpretable if you can inspect the actual model and understand why it got a particular answer for a given input, and how the answer would change when the input changes
  - Can you give some examples of interpretable ML models?

- **Explainability**: an explainable model is one that can help you understand "why was this output produced for this input?"

Operation, monitoring, and maintenance

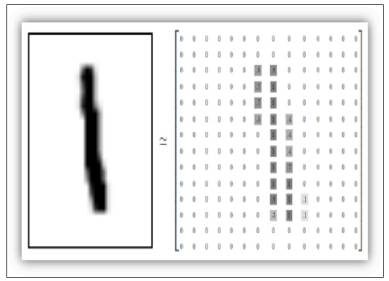
- Long-tail: user inputs / preferences
- Non-stationarity: on a social networks, new connections constantly form
- Machine learning tests: features, data, models, infrastructure

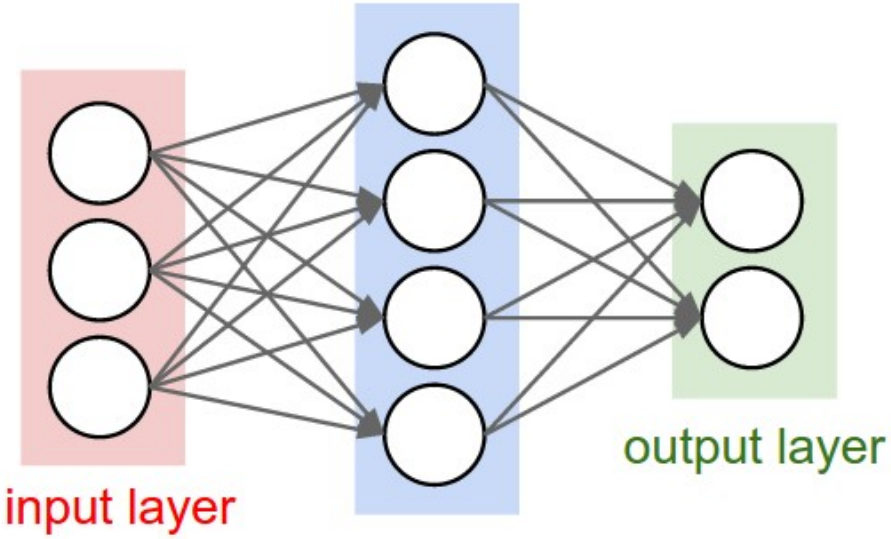
Lecture plan

- **Forward and backward passes of neural networks in PyTorch**
- Examples of neural network classifiers in PyTorch

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:** σ: $\mathbb{R} \to \mathbb{R}$
  - **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 $1 \times d_1$

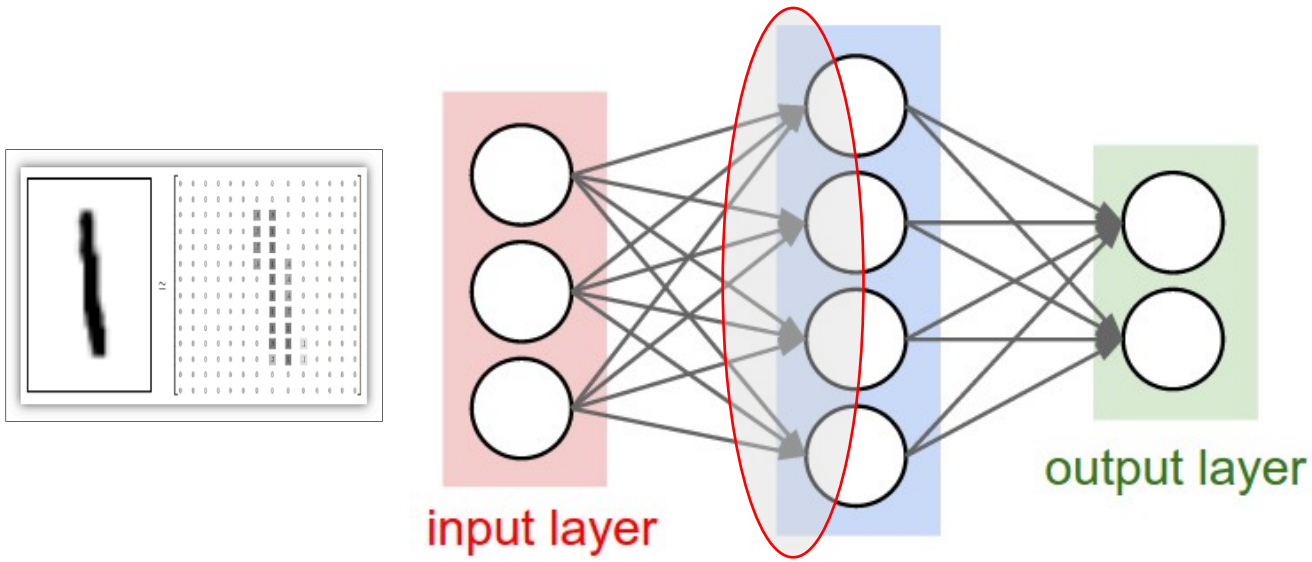


Illustration of forward pass

- 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 $1 \times d_1$

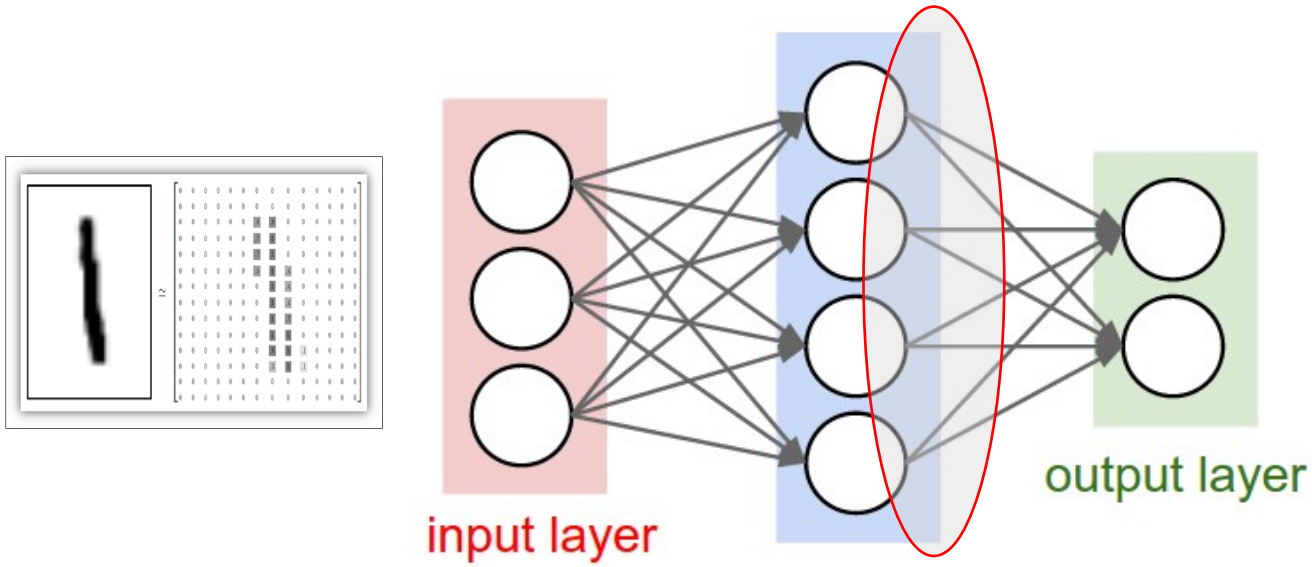


Illustration of forward pass

- 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 $1 \times d_2$

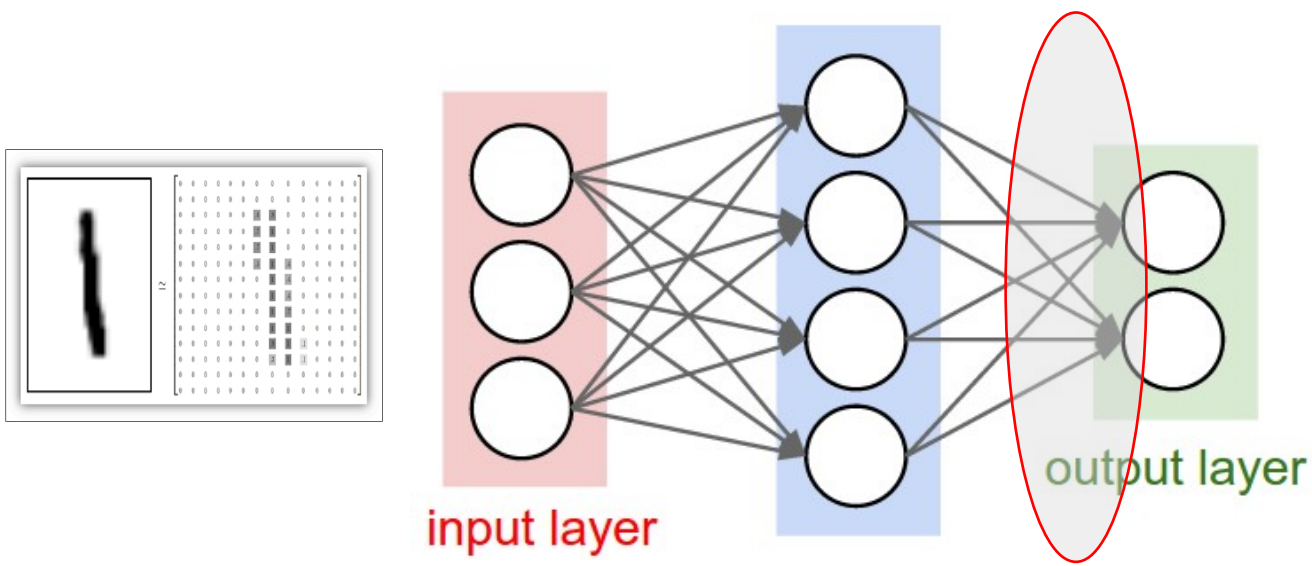
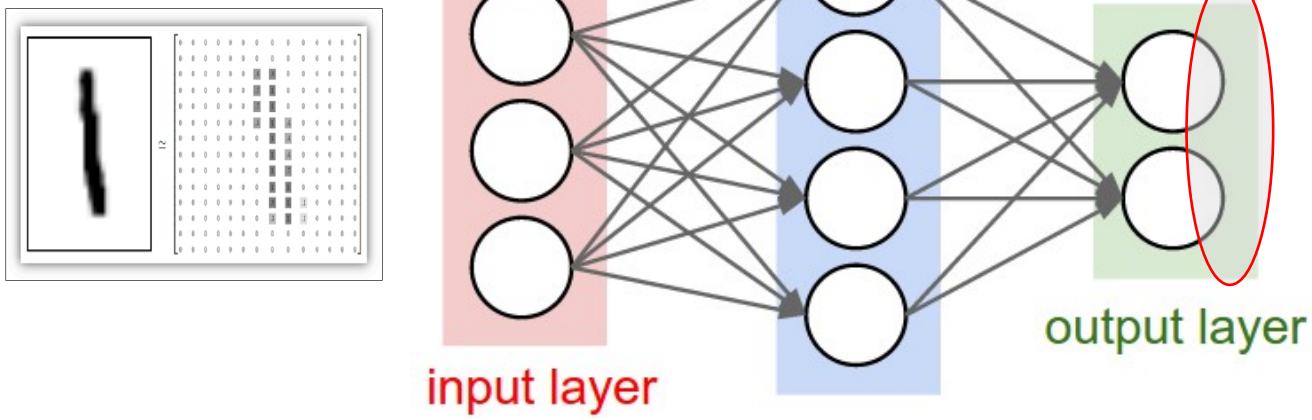


Illustration of forward pass

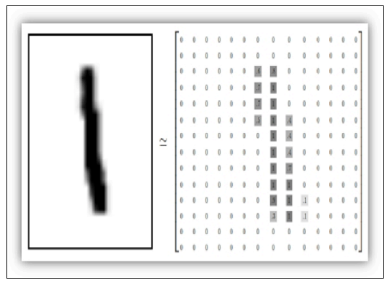
- Finally, apply softmax to get the probability distribution over ten output categories
  - Input to the hidden layer: $x^\top w_1$, size $1 \times d_1$
  - Output of the hidden layer: $\sigma(x^\top w_1)$, size $1 \times d_1$
  - Input to the output layer: $\sigma(x^\top w_1) w_2$, size $1 \times d_2$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

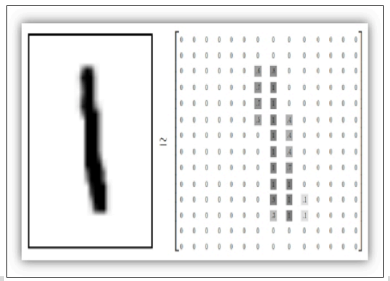


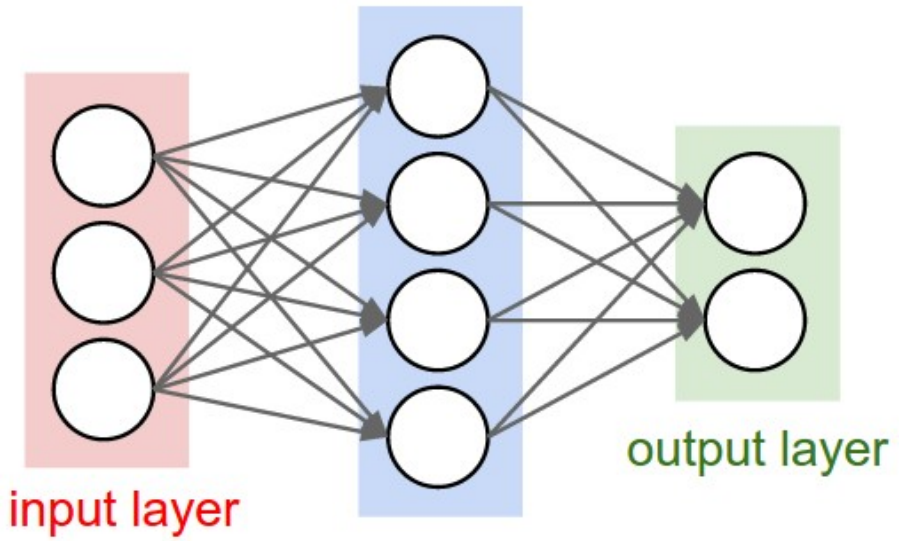
Softmax output [0.01, **0.9**, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.01]

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





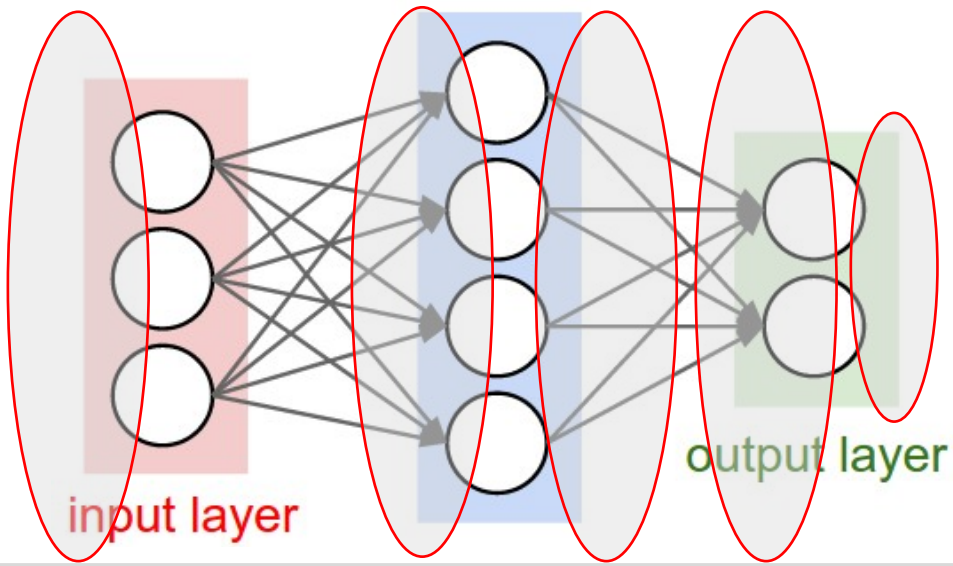


Softmax [0.01, **0.9**, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.01]

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

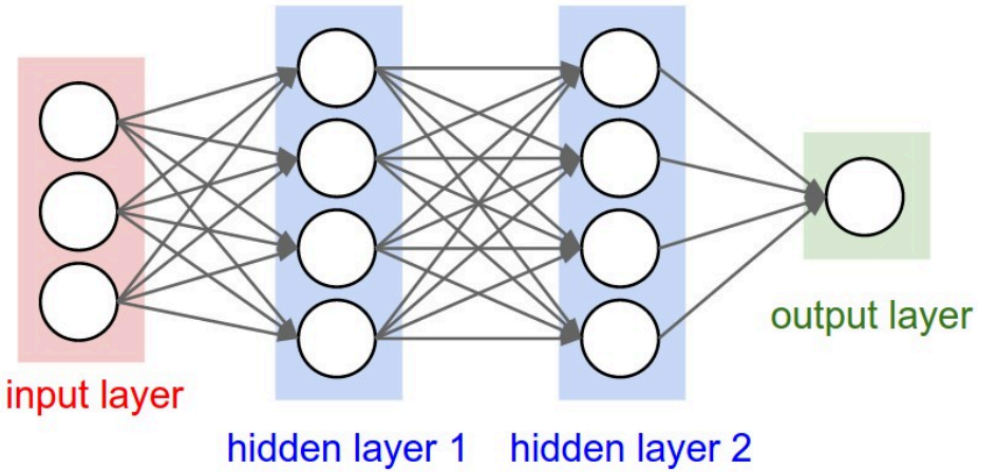
Applying forward pass to a batch of inputs

- 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

Implementation of stochastic gradient

- 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)
```

- 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)
```

Implementation of stochastic gradient

- *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

- Forward and backward passes of neural networks in PyTorch
- **PyTorch implementation of neural network 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

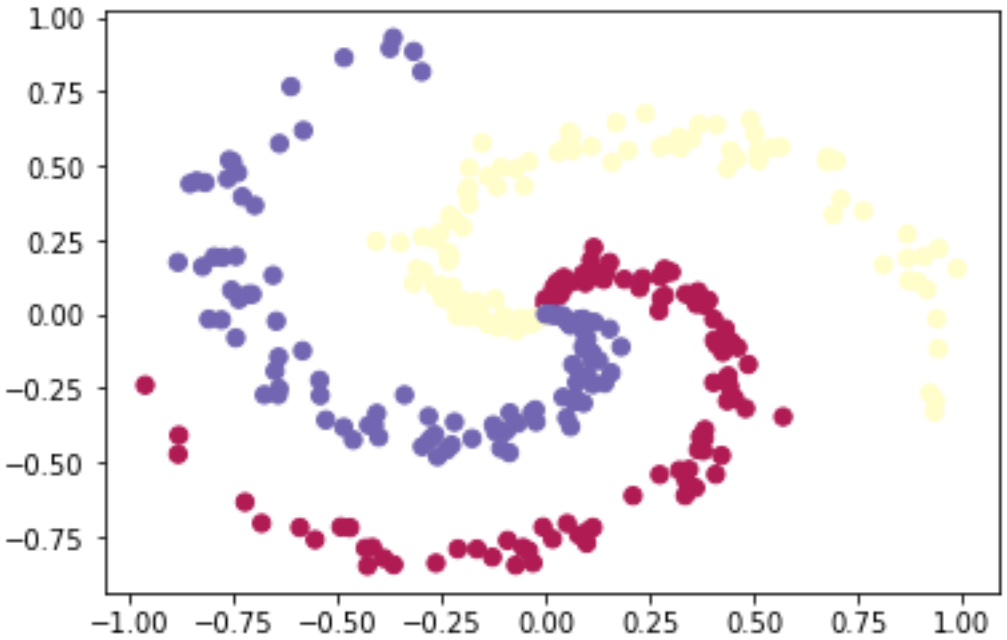
generating some data

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 + ℓ_2 penalty

```
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])
```

```
# compute the loss: average cross-entropy loss and regularization
data_loss = np.sum(correct_logprobs)/num_examples
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: $\frac{\partial \ell(W,b)}{\partial W} = \frac{\partial \ell}{\partial u} \cdot \frac{\partial u}{\partial W}$ (optional)
  - Claim: $\frac{\partial \ell}{\partial u_k} = p_k - 1_{y=k}$, $\frac{\partial u}{\partial W} = X^\top$ (optional)

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)

```
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
```

- Training 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
```

This is quite high for three classes:

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

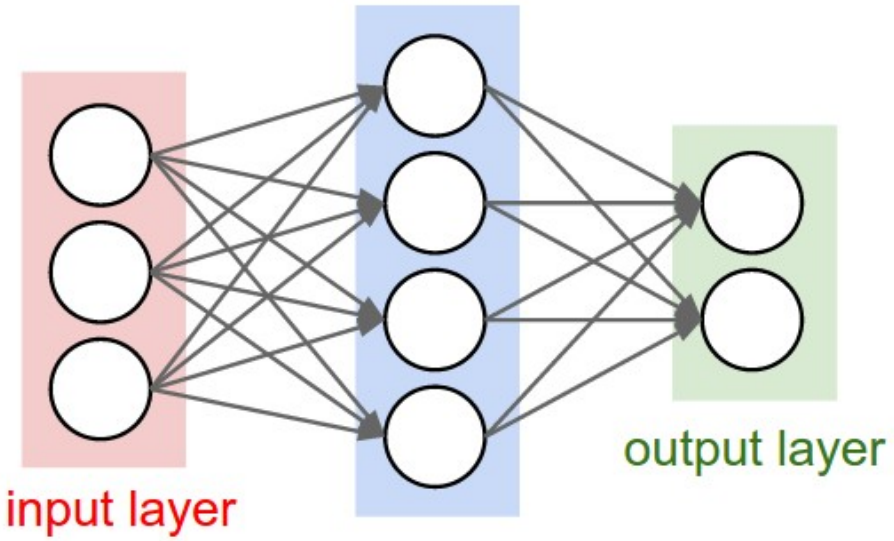
Can we do better?

- 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): $\sigma(z) = \max(z, 0)$

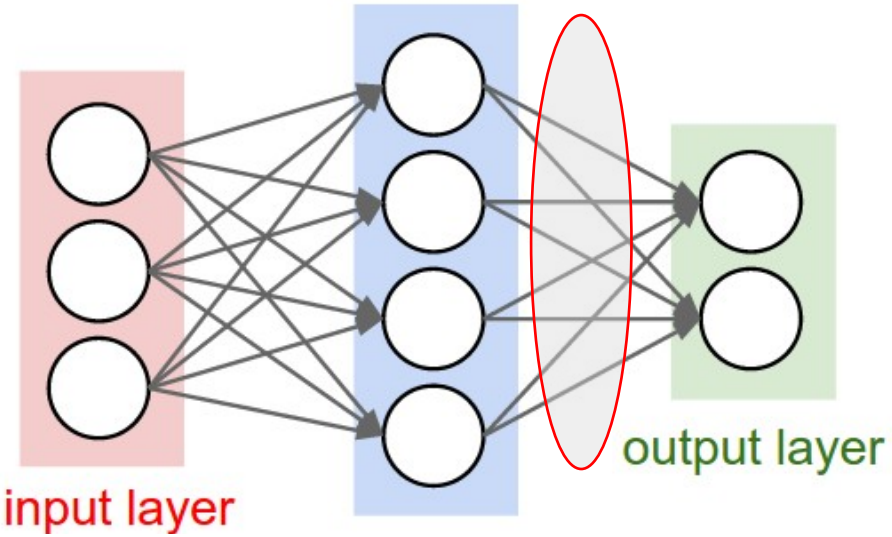
$$u = \sigma(XW_1 + 1 \cdot b_1)W_2 + 1 \cdot b_2$$

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 uses chain rule




```
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
```

```
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)
```

Visualization

- Visualizing decision boundaries