CSE 156 | Lecture 3: Text Classification with FeedForward Neural Networks

Ndapa Nakashole

October 8, 2024

Today

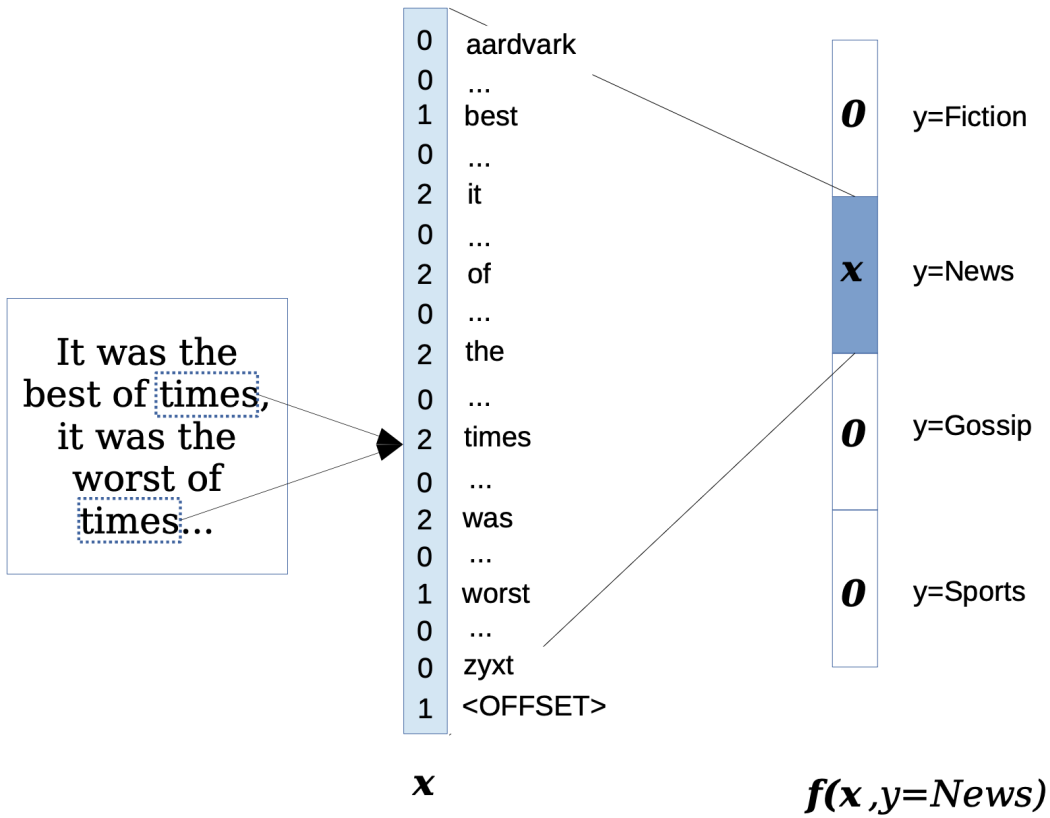
1. Finish up on Linear Classifiers
2. FeedForward Neural Networks
3. Training Neural Networks

Administrative matters

- **PA1**: released Friday, due October 18 at 11.59 pm

Recap: Text Classification, Representation

- Categorize a piece of text into one of many predefined categories
- Representations: create a feature vector from text, e.g., BoW

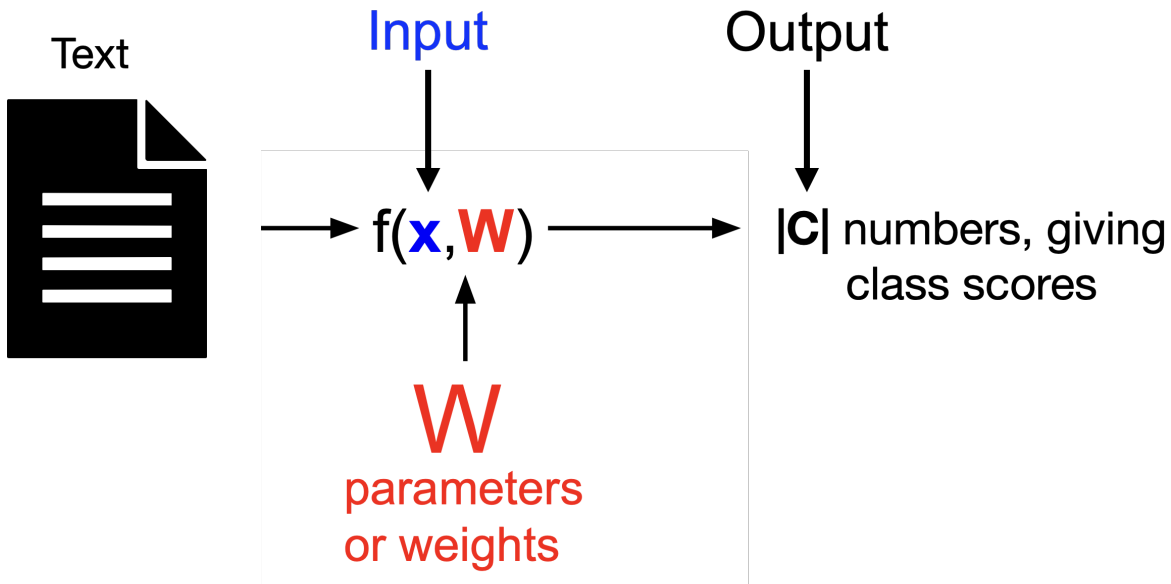


Recap: data splits, linear classifier

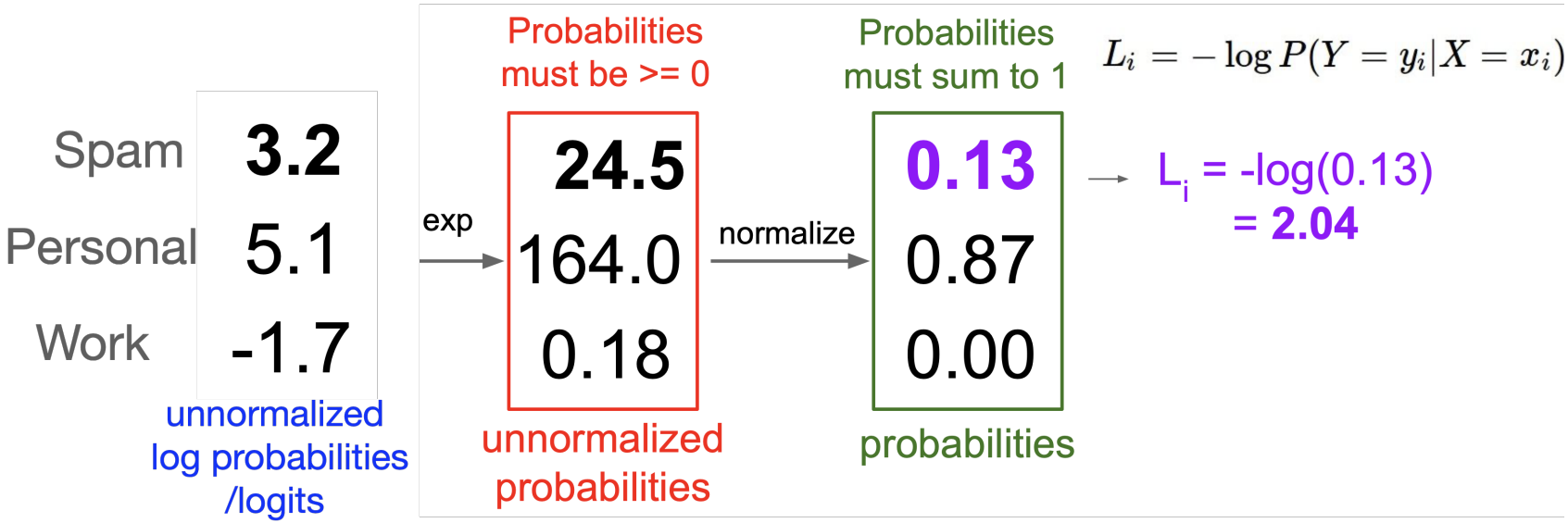
Split data into **train, validation**; choose **hyperparameters on validation** and **evaluate on test**

Linear Classifier: A function f of the input features x, and the parameters W:

$$\mathbf{s} = f\left(x_i; W\right) = \mathbf{W}x_i$$



Recap: softmax; negative log likelihood loss ; Regularization



Loss: quantifies the errors the model is making on the training data (0, 3, 90, ..? lower is better)

Regularization: want to fit the but also want W to be "nice" - where nice is: small values, sparse, etc.

The key components of a machine learning model

Key components of a machine learning model

1. Model parameterized by $\mathbf{W}$: a function f, that maps inputs to outputs
2. Loss function by $L(\mathbf{W})$: that quantifies the errors the model is making on the training data
3. Optimization for finding the best parameters $\mathbf{W}$ that minimize the loss function $L(\mathbf{W})$ as

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} L(\mathbf{W})$$

finding the best $\mathbf{W}$ that minimizes $L(\mathbf{W})$:

Optimization

Strategy #1: A first bad solution: Random search

Guess and check

We can evaluate $L(W)$ for any setting of W,

Randomly sample W and compute $L(W)$ until we find the best W.

```
# assume X_train is the data where each column is an example (e.g., 3073 × 50,000)
# assume the function L evaluates the loss function

import numpy as np

bestloss = float("inf")  # Python assigns the highest possible float value

for num in range(1000):
    W = np.random.randn(10, 3073) * 0.0001  # generate random parameters
    loss = L(X_train, Y_train, W)  # get the loss over the entire training set
    if loss < bestloss:  # keep track of the best solution
        bestloss = loss
        bestW = W
    print(f'in attempt {num} the loss was {loss}, best {bestloss}')
    # Output example:
    # in attempt 0 the loss was 9.401632, best 9.401632
    # in attempt 1 the loss was 8.959668, best 8.959668
    # in attempt 2 the loss was 9.044034, best 8.959668
```

Optimization: Getting to the lowest point in the valley

loss function $L(\mathbf{W})$ is like a landscape with hills and valleys. We want to find the lowest point in the valley.

Random Search: jump around and measure the altitude

We are blindfolded but we have an altitude sensor (tells us the loss, height of the surface, at any point). We can randomly sample points (jump around randomly) and measure the altitude.

Strategy #2: Iterative Improvement

- Core idea: Finding the best set of weights $\mathbf{W}$ is very difficult
- Instead start somewhere and **refine** a specific set of weights $\mathbf{W}$ to be slightly better
  - **Start with a random $\mathbf{W}$** and iteratively **refine it to reduce the loss $L(\mathbf{W})$**
- How to refine $\mathbf{W}$ towards the minimum?

Strategy #2: Iterative Improvement by following the slope

1. **Look Around:** which way is downhill? (mathematically, the gradient of the function at the current point)
2. **Take a Step:** Move downhill in that direction with a step size aka the "learning rate"
3. **Repeat:** Continue taking steps downhill and reassess the direction after each step

Finite difference approximation of the derivative

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- The derivative of a function at a point is the slope of the tangent line to the function at that point
  - The instantaneous rate of change of the function
- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

Numerical gradient: First dimension

current W:		gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] loss 1.25347		[?, ?, ?, ?, ?, ?, ?, ?, ?,…]

First dimension

current W:	**W + h** (first dim)**:**	**gradient dW:**
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[0.34 + **0.0001**, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[?, ?, ?, ?, ?, ?, ?, ?, ?,…]
loss 1.25347	**loss 1.25322**	

First dimension

current W:	W + h (first dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[0.34 + **0.0001**, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[-2.5, ?, ?, ?, ?,…]
loss 1.25347	loss 1.25322	

(1.25322 - 1.25347)/0.0001
= -2.5

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- In this dimension the gradient is negative, we see the loss decreasing, so we want to increase the weights, move in the positive direction of the gradient

Second dimension

current W:	**W + h** (second dim):	**gradient dW:**
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11 + **0.0001**,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25353**	

Second dimension

current W:	**W + h** (second dim):	**gradient dW:**
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[0.34, -1.11 + **0.0001**, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[-2.5, **0.6**, ?, ?, ?,...]
loss 1.25347	**loss 1.25353**	(1.25353 - 1.25347)/0.0001 = 0.6 $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$

- In this dimension the gradient is positive, we see the loss increasing, so we want to decrease the weights, move in the negative direction of the gradient

Third dimension

current W:	**W + h** (third dim)**:**	**gradient dW:**
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11,	0.6,
0.78,	0.78 + **0.0001**,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25347**	

Third dimension

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]
loss 1.25347

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]
loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,...]

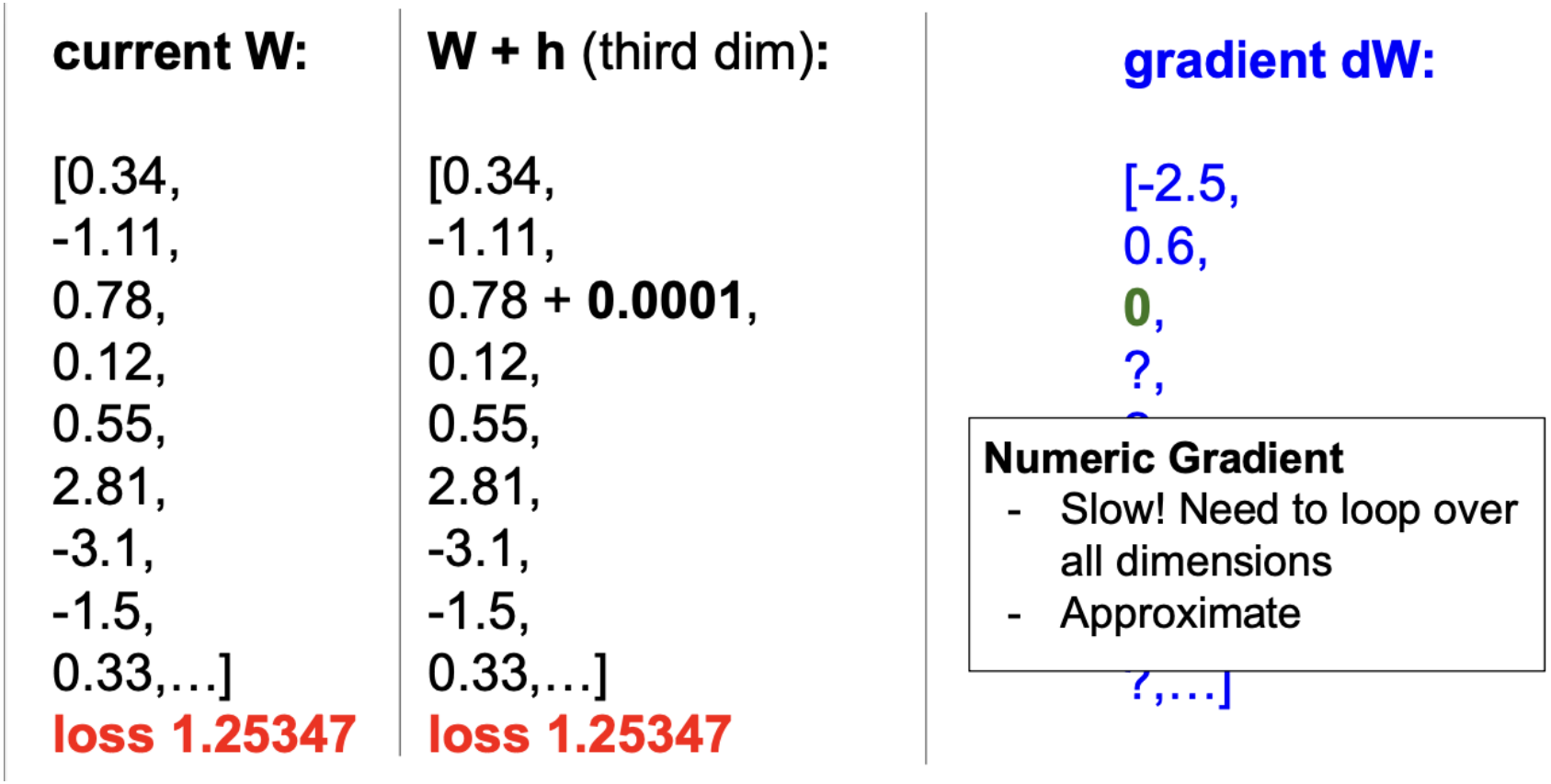
(1.25347 - 1.25347)/0.0001
= 0

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- In this dimension the gradient is zero, we see the loss remains the same, so we want to keep the weights the same in this dimension

Need to loop over all dimensions

Numerical gradient is too slow, and it an approximation



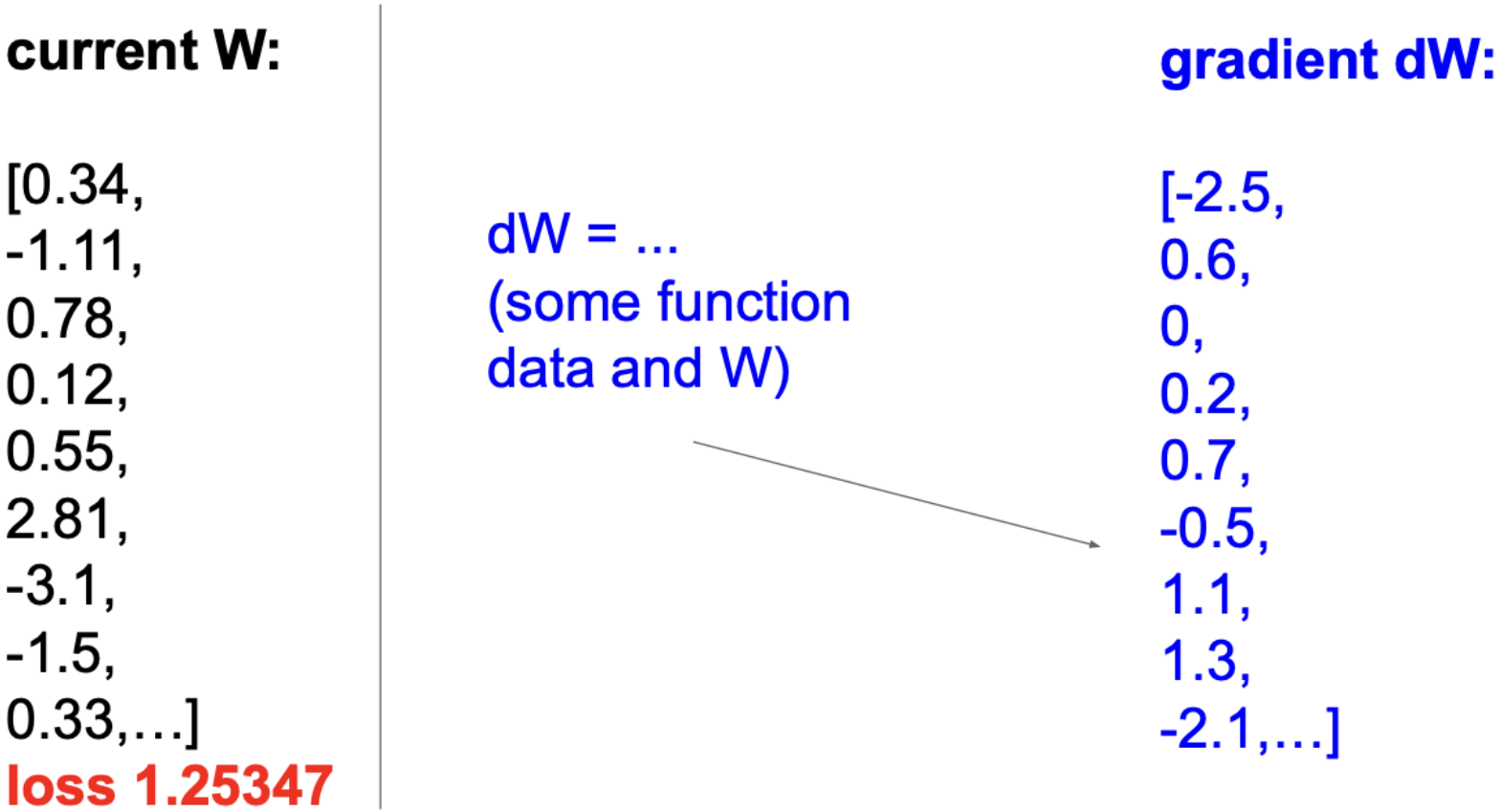
- Use calculus to come up with an expression for the gradient of the loss, then can evaluate **the entire gradient vector at once**

Analytic Gradient: use calculus to get $\nabla_W L$

Remember the softmax loss function:

$$
\begin{aligned}
L(\mathbf{W}) &= -\sum_{i=1}^{n} \log p\left(y_i \mid x_i; \mathbf{W}\right) \\
&= -\sum_{i=1}^{n} \log \left(\frac{\exp\left(\mathbf{W}_{y_i} \cdot x_i\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right)} \right) \\
&= -\sum_{i=1}^{n} \left(\mathbf{W}_{y_i} \cdot x_i - \log \sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right) \right)
\end{aligned}
$$

Compute the gradient of the loss function



Gradient of the Loss Function (1)

Loss Function for a Single Data Point:

$$L_i(\mathbf{W}) = -\left(\mathbf{W}_{y_i} \cdot x_i - \log \sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right)\right)$$

We are interested in the gradient of the loss with respect to the parameters of the correct class $\mathbf{W}_{y_i}$.

Step 1: Differentiate the First Term

$$\frac{\partial}{\partial \mathbf{W}_{y_i}} \left(\mathbf{W}_{y_i} \cdot x_i\right) = x_i$$

Gradient of the Loss Function (2/3)

Step 2: Differentiate the Second Term

$$\log \sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right)$$

Use chain rule to differentiate the log . The derivative of $\log(z)$ wrt z is $\frac{1}{z}$. Therefore:

$$\frac{\partial}{\partial \mathbf{W}_{y_i}} \log \sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right)$$

$$= \frac{1}{\sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right)} \cdot \frac{\partial}{\partial \mathbf{W}_{y_i}} \left(\sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right) \right)$$

Second, for the sum, $\exp(\mathbf{W}_{y'} \cdot x_i)$ only depends on $\mathbf{W}_{y_i}$ when $y' = y_i$, thus

$$\frac{\partial}{\partial \mathbf{W}_{y_i}} \sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right) = \exp\left(\mathbf{W}_{y_i} \cdot x_i\right) x_i$$

Gradient of the Loss Function (3/3)

Combining the Results:

$$\frac{\partial L_i(\mathbf{W})}{\partial \mathbf{W}_{y_i}} = -x_i + \frac{\exp\left(\mathbf{W}_{y_i} \cdot x_i\right) x_i}{\sum_{y' \in \mathcal{Y}} \exp\left(\mathbf{W}_{y'} \cdot x_i\right)}$$

recogning the softmax function in the second term, we can simplify the expression to:

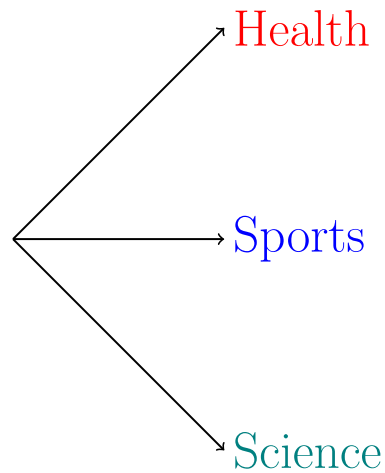
$$\boxed{\frac{\partial L_i(\mathbf{W})}{\partial \mathbf{W}_{y_i}} = -x_i + p(y_i \mid x_i; \mathbf{W}) x_i}$$

This is the gradient of the loss with respect to the correct class weight vector $\mathbf{W}_{y_i}$.

- Update for other classes is the same but without the first term

Example with Text Classifiction: scores

too many drug trials, too few patients



Three features, does the text mention drugs, patients, or baseball?

$x_i =$ I[*drug*], I[*patients*], I[*baseball*] $= [1, 1, 0]$

$\mathbf{W}_{y_{\text{health}}} = [+2.1, +2.3, -5]$
$\mathbf{W}_{y_{\text{sports}}} += [-2.1, -3.8, +5.2]$
$\mathbf{W}_{y_{\text{science}}} += [+1.1, -1.7, -1.3]$

$(\mathbf{W}_y \cdot x_i) =$ Health: +4.4 Sports: -5.9 Science: -0.6

Example with Text Classifiction: gradients

$$\frac{\partial L_i(\mathbf{W})}{\partial \mathbf{W}_{y_i}} = -x_i + p(y_i \mid x_i; \mathbf{W})x_i$$

$$\frac{\partial L_i(\mathbf{W})}{\partial \mathbf{W}_{y'}} = p(y_i \mid x_i; \mathbf{W})x_i$$

"too many drug trials, too few patients" $y_i =$ Health

$x_i = [1, 1, 0]$ $\mathrm{P}_w(y \mid x) = [0.2, 0.5, 0.3]$ (made up values)

gradient $\mathbf{W}_{y_{\text{health}}} = -[1, 1, 0] + 0.2[1, 1, 0]$

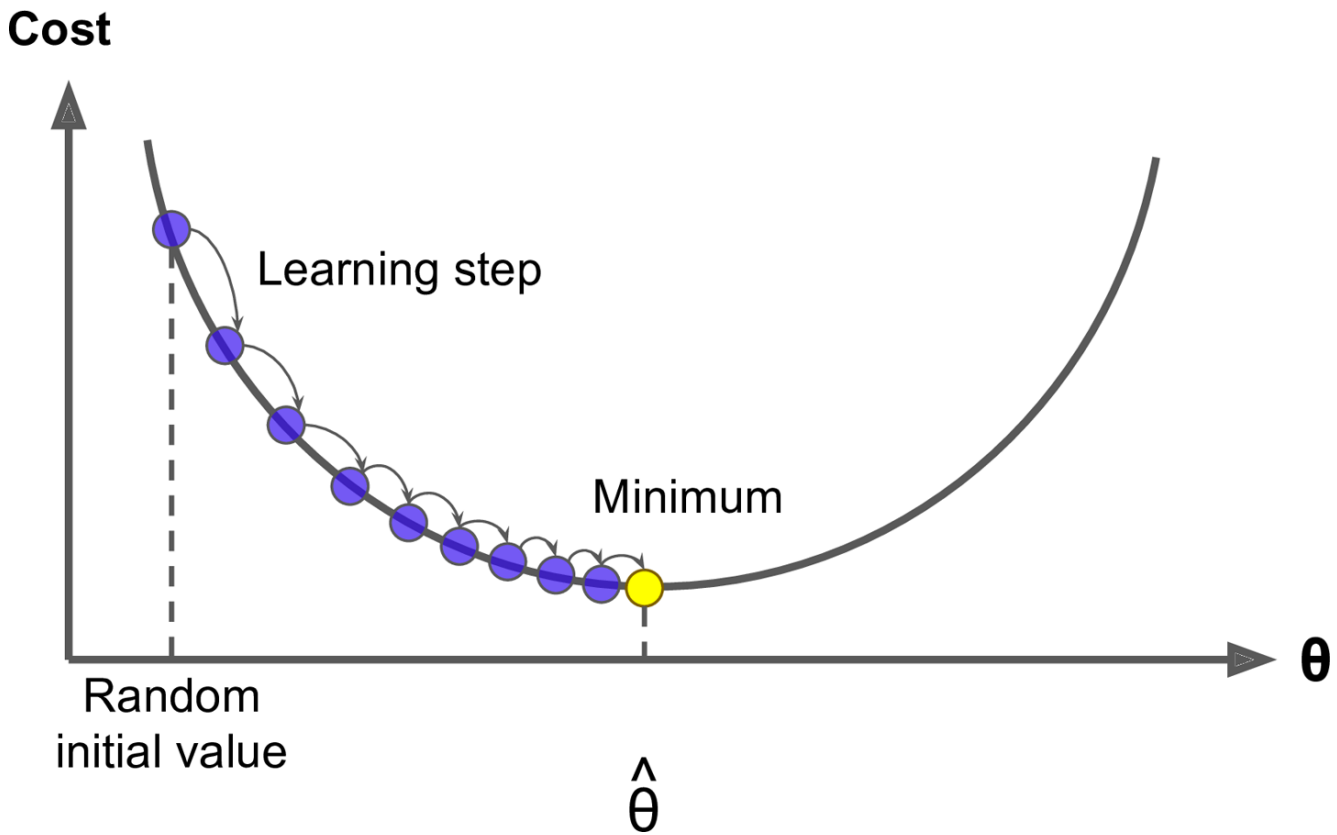
gradient $\mathbf{W}_{y_{\text{sports}}} = 0.5[1, 1, 0]$

gradient $\mathbf{W}_{y_{\text{science}}} = 0.3[1, 1, 0]$

$\mathbf{W}$ updates: make $Sports$ and $Science$ look less like the example (in proportion to how wrong we were), make $Health$ look more like it

Gradient Descent: iterative improvement

- We have a function $L(\mathbf{W})$ we want to minimize. Gradient Descent is an algorithm to minimize $L(\mathbf{W})$
- Idea: for current value of $\mathbf{W}$, calculate gradient of $L(\mathbf{W})$, then take small step in the direction of the negative gradient. Repeat until convergence



Gradient Descent Update Rule

$$W_{t+1} = W_t - \alpha \nabla L\left(W_t\right) \quad \text{Gradient Descent Update Rule}$$

where:

α: step size (learning rate)

$\nabla L\left(W_t\right)$: gradient of L at W_t

W_t; W_{t+1}: current and new value of W

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N} \sum_{i} L_i \left(f\left(x_i, W \right), y_i \right)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i \left(f\left(x_i, W \right), y_i \right)$$

- **Full sum:** is expensive when N is large
- **Approximate sum:** using a **minibatch** of examples (32 / 64 / 128 common)

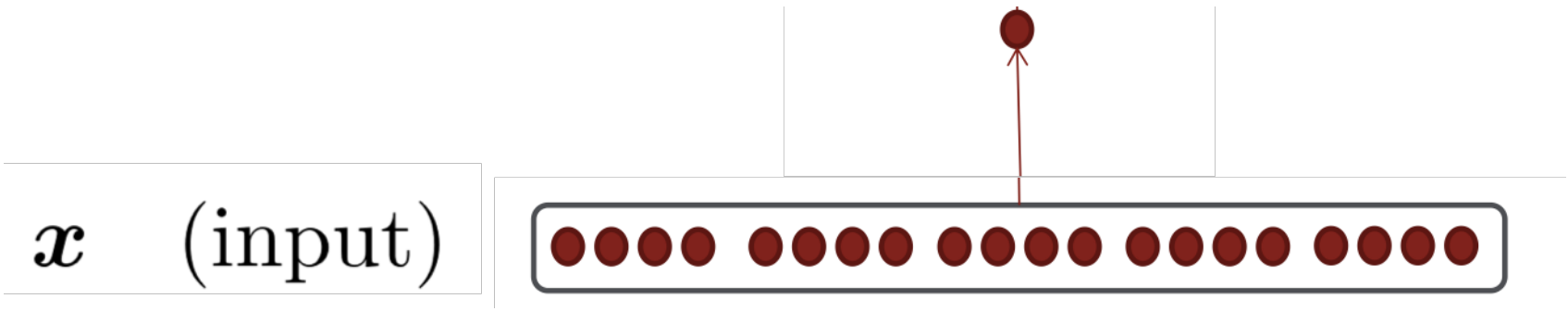
```
# Vanilla Minibatch Gradient Descent
while True:
    data_batch = sample_training_data(data, 256) # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad # perform parameter update
```

Backpropagation

- **Gradient Descent** is a procedure of repeatedly evaluating the gradient and then performing a parameter update.
- **Backpropagation** is a method to compute the gradient of the loss function with respect to the weights. It is an **application of the chain rule of calculus**

Text Classification with Linear Models: Summary

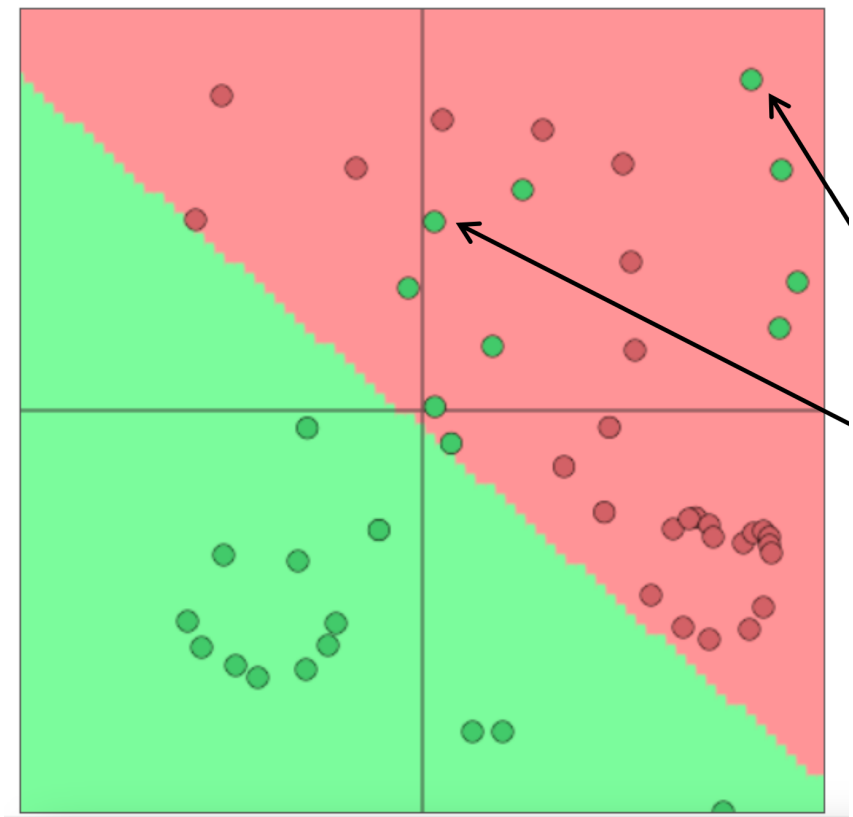
- Convert text to a feature vector
- Use a linear classifier to classify the text (having trained it), and pick the class with the highest score



- Hyperparameters to tune: regularization penalty, step size, number of iterations, etc.

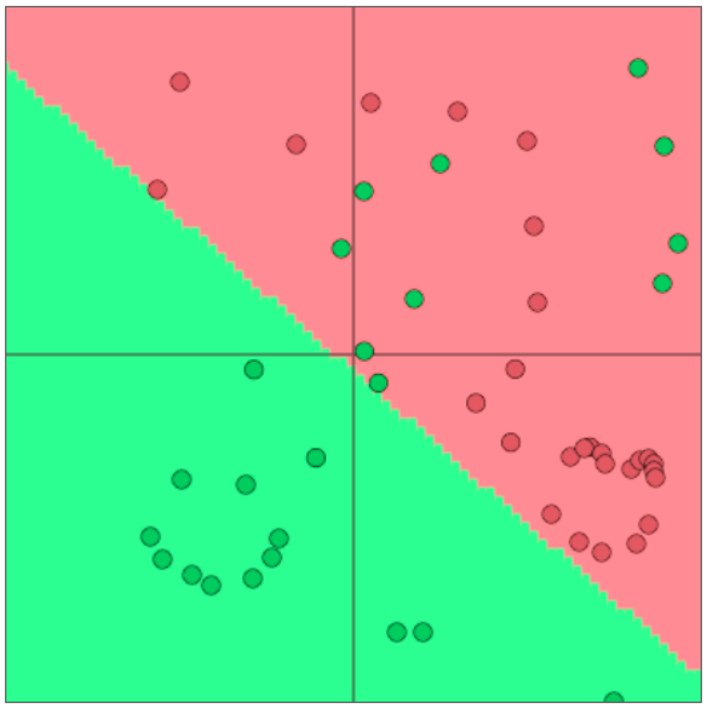
Linear classifiers are limited in their expressiveness

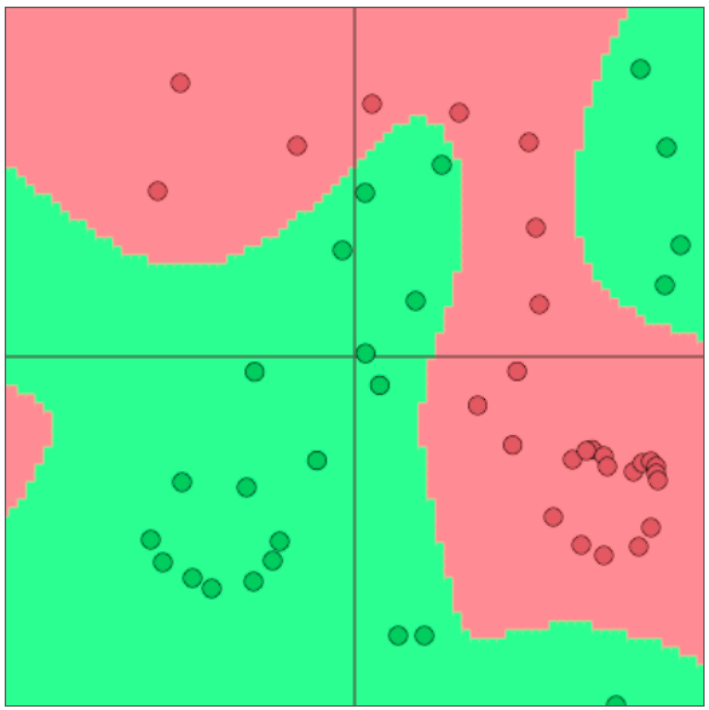
- Softmax is simple, interpretable, and fast but has limitations
- Only gives linear decision boundaries
  - $\rightarrow$ Unhelpful when a problem is complex



FeedForward Neural Networks

- Neural networks can learn much more complex functions with nonlinear decision boundaries!





From multi-class to binary logistic regression

- So far: linear classifiers for multi-class logistic regression (softmax)
- For binary classification, we can use a **binary logistic regression** model (sigmoid)

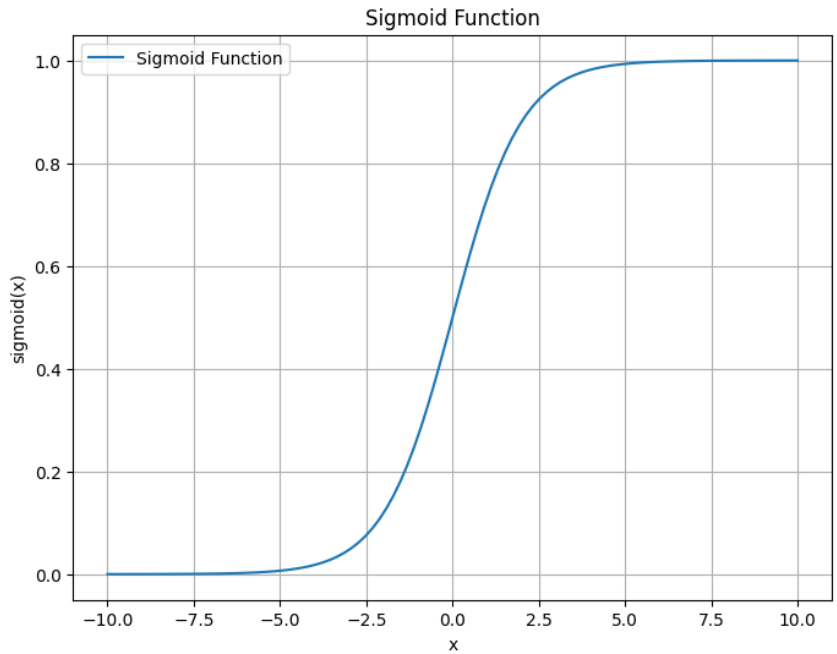
Binary linear classifier: Use Sigmoid instead of Softmax

Softmax classifier:

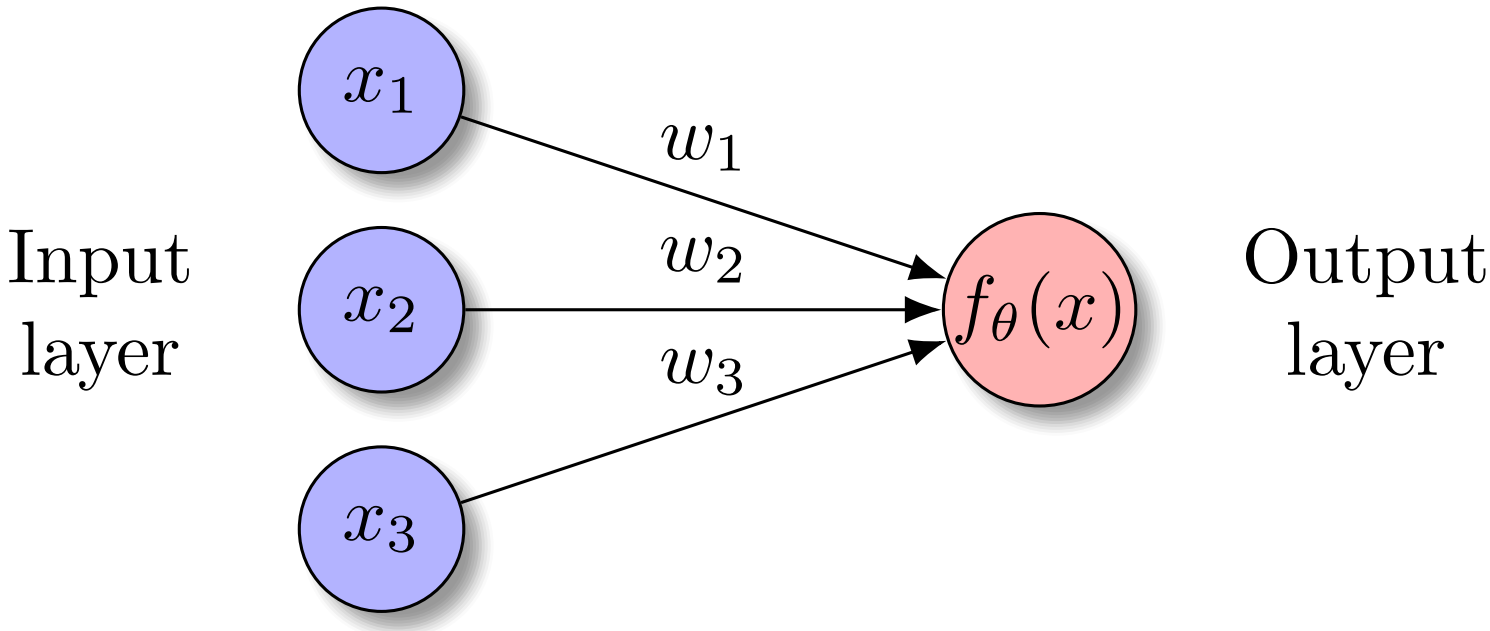
- Scores (vector): $\boldsymbol{s} = \boldsymbol{W} x_i$
- Softmax function for probability: $P(Y = k) = \frac{e^{s_k}}{\sum_j e^{s_j}}$

Sigmoid classifier (binary):

- Score (scalar): $s = \boldsymbol{w} x_i$
- Sigmoid function for probability:
  - Positive class: $P(Y = 1) = \sigma(s) = \frac{1}{1+e^{-s}}$
  - Negative class: $P(Y = 0) = 1 - \sigma(s)$



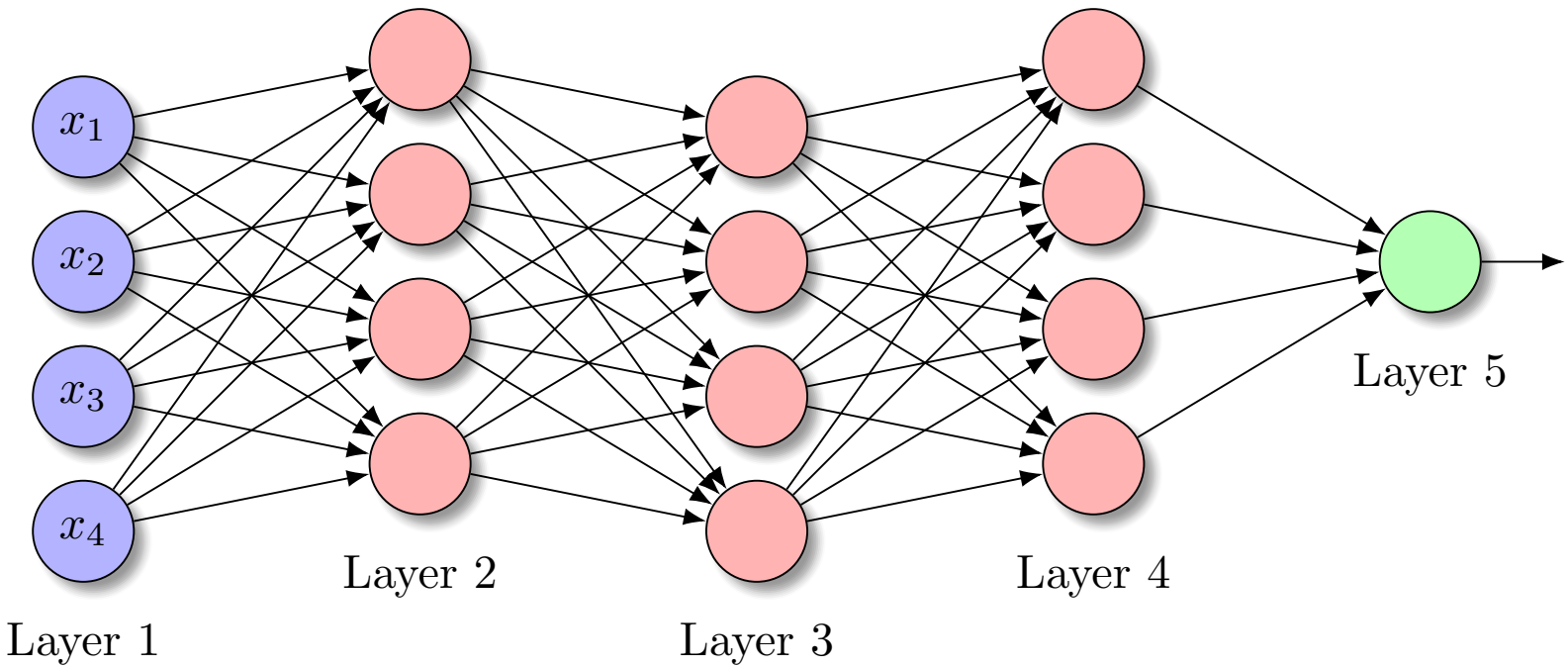
(binary) Linear Classifier Depicted



Output: $f_\theta(x) = \mathbf{w} \cdot x$
Parameters: $\theta = \mathbf{w}$
**binary classification: $\mathbf{w}$ is vector instead of a matrix.

Neural computation

- A neuron: the basic procesisng unit in a neural network, takes in multiple inputs, produces an output
- A neural network is a **collection of neurons that are connected in layers**
- A **neuron can be modeled as a binary logistic regression unit**



A neuron: as binary logistic regression unit

A neuron is defined by:

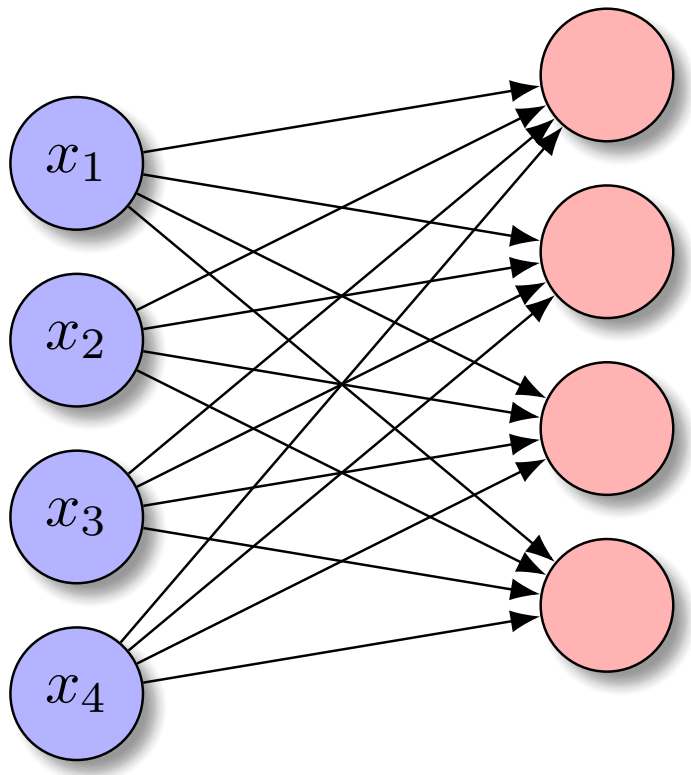
- weight vector $w \in \mathbb{R}^d$
- bias $b \in \mathbb{R}$
- activation $g : \mathbb{R} \to \mathbb{R}$

The neuron maps an input vector $x \in \mathbb{R}^d$ to an output h as follows:

$$h = g(w \cdot x + b) \quad \text{neuron output}$$

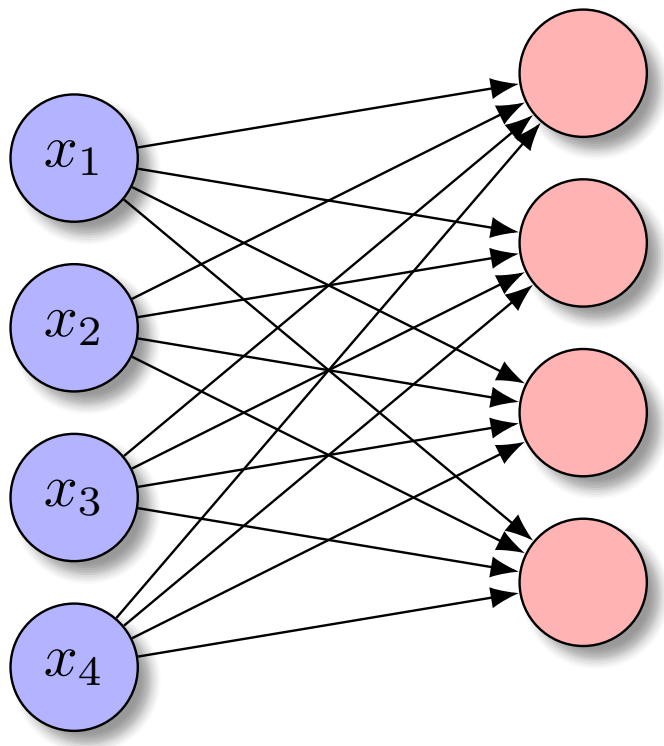
A neural network = running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...



A neural network = running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...



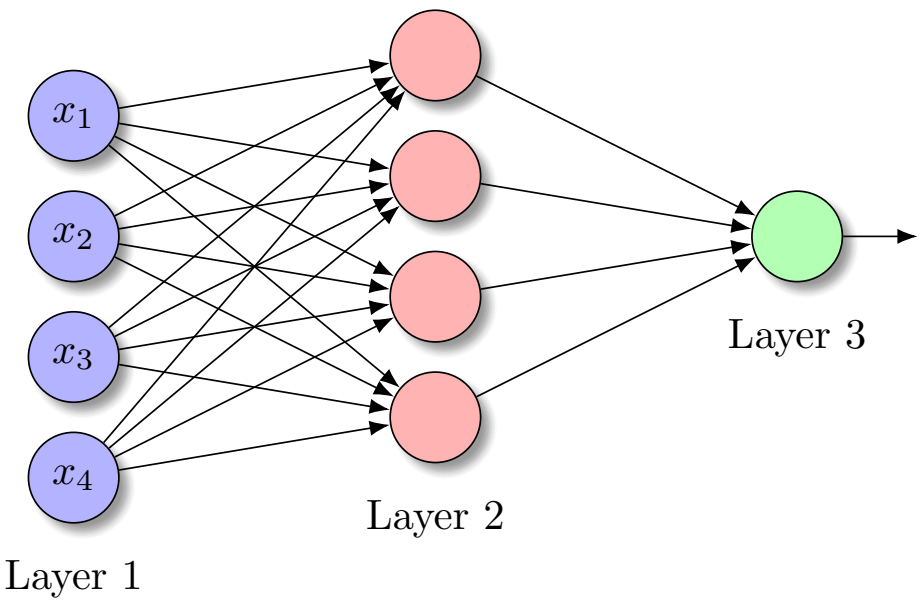
Layer 1

But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

A neural network = running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...

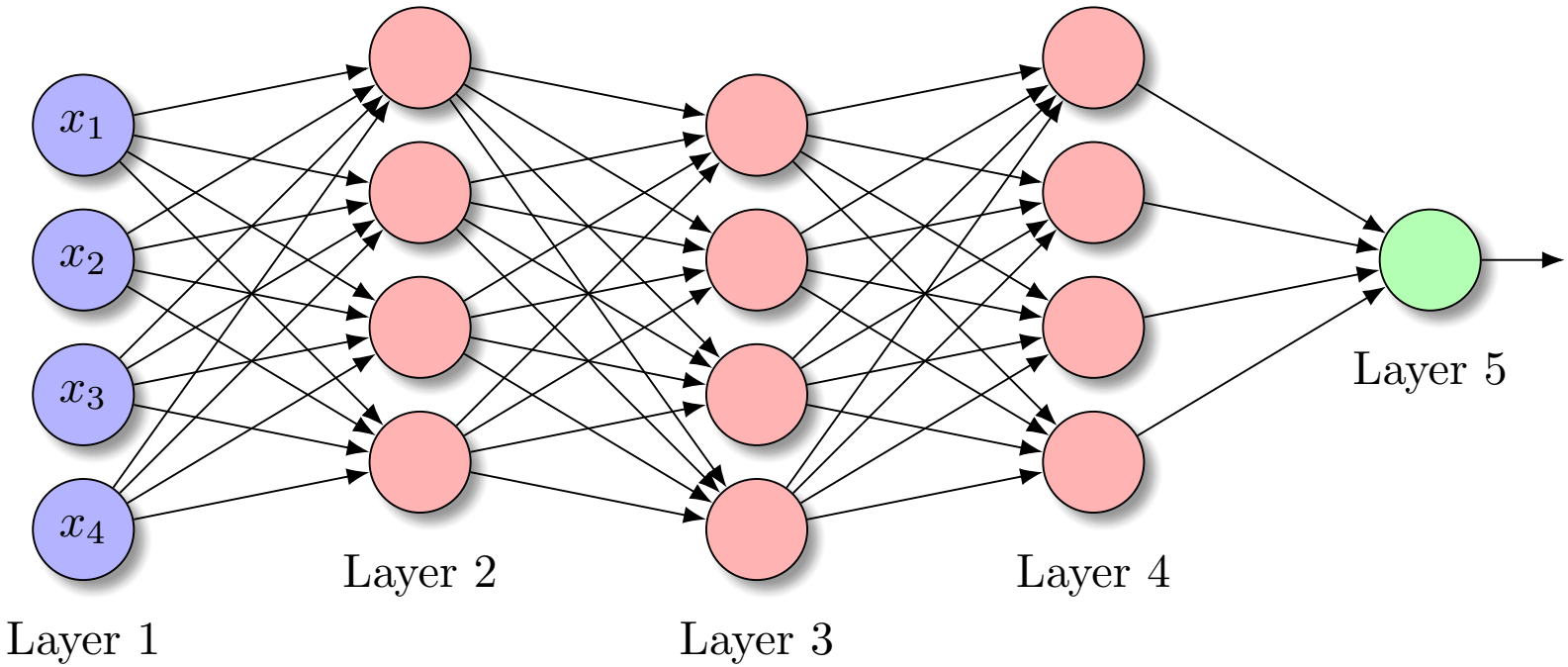
- ... and then we can feed that vector through another logistic regression function!



It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

A neural network = running several logistic regressions at the same time

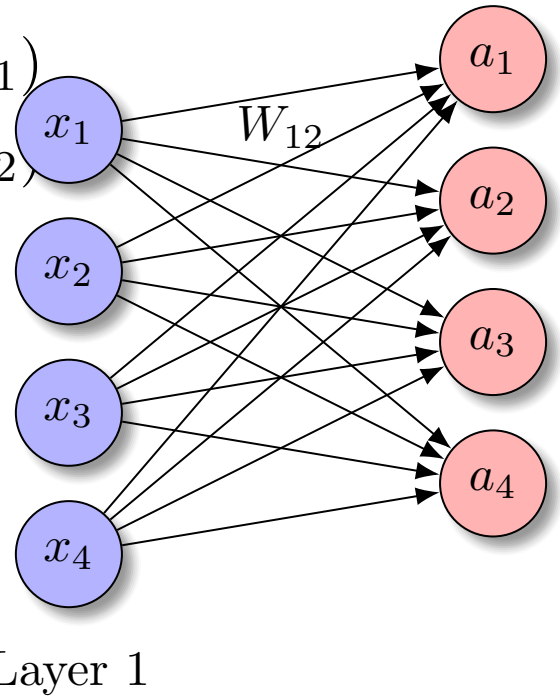
Before we know it, we have a multilayer neural network ...



This allows us to **re-represent and compose our data multiple times and to learn a classifier that is highly non-linear in terms of the original inputs** (but, typically, is linear in terms of the pre-final layer representations)

Matrix notation for a layer

$$a_1 = g\left(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + W_{14}x_4 + b_1\right)$$
$$a_2 = g\left(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + W_{24}x_4 + b_2\right)$$
$a_i = i$-th neuron is given by i-th row of W



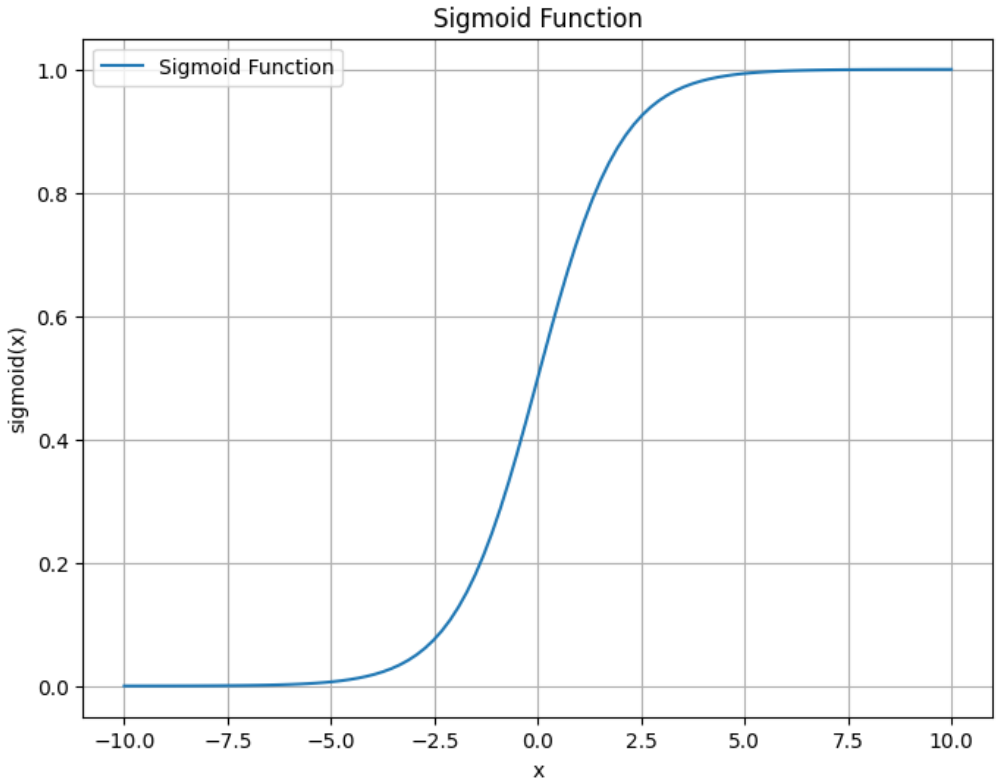
- $a_1, a_2, \ldots$: activations
- g: activation function
- W_{ij}: weight
- b_i: bias (not depicted)

in Matrix notation:
$z = Wx + b$
$a = g(z)$

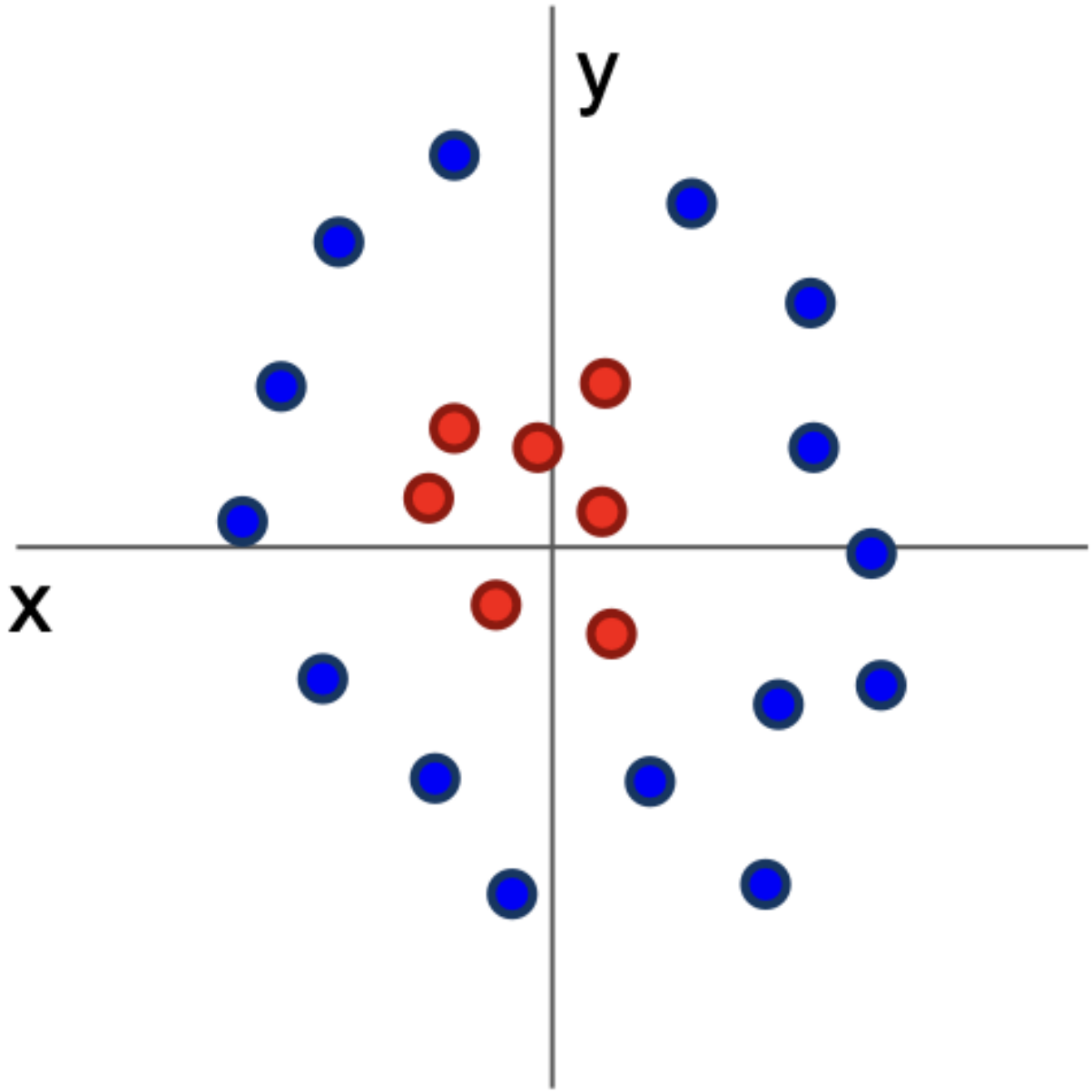
Activation g is applied element-wise:
$g\left([z_1, z_2, z_3]\right) = \left[g\left(z_1\right), g\left(z_2\right), g\left(z_3\right)\right]$

Activation function g

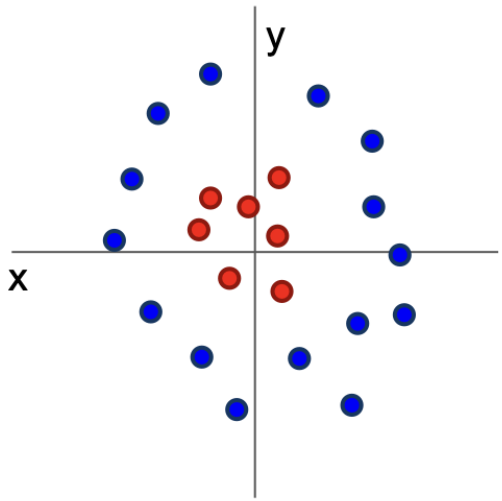
- g: crucial that it be a nonlinear function
- e.g., g can be the **sigmoid function** $\sigma(z) = \frac{1}{1+e^{-z}}$
- sigmoid function squashes the input to the range $[0, 1]$



Why non-linearities?

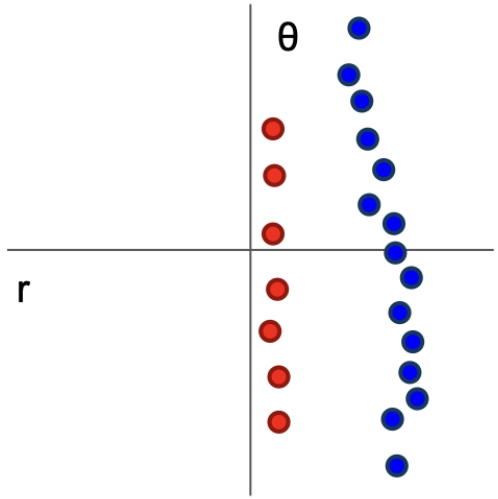


Why non-linearities?



Cannot separate red and blue points with linear classifier

$f(x, y) = (r(x, y), \theta(x, y))$



After applying feature transform, points can be separated by linear classifier

Why is the non-linear activation function important?

(**Linear classifier:**) Linear score function:

$$f = Wx$$
$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

(**FFN:**) 2-layer neural network:

$$f = W_2 g\left(W_1 x\right)$$
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Q: What if we try to build a neural network without an activation function?

$$f = W_2 W_1 x$$

A: We end up with a linear classifier again!

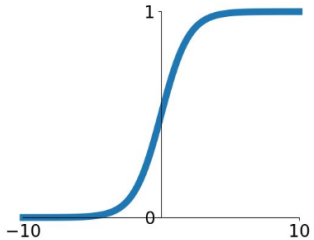
$$f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

The activation function $g(z)$ has to be non-linear to make the neural network more expressive.

Activation functions

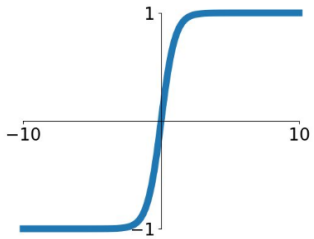
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



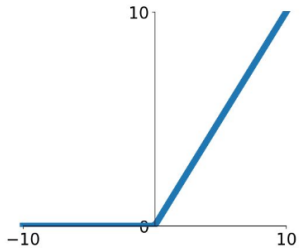
tanh

$$\tanh(x)$$



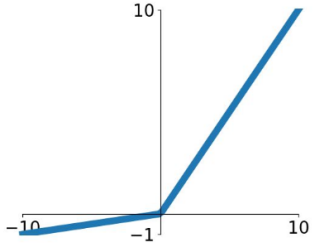
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

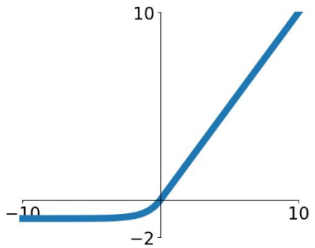


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

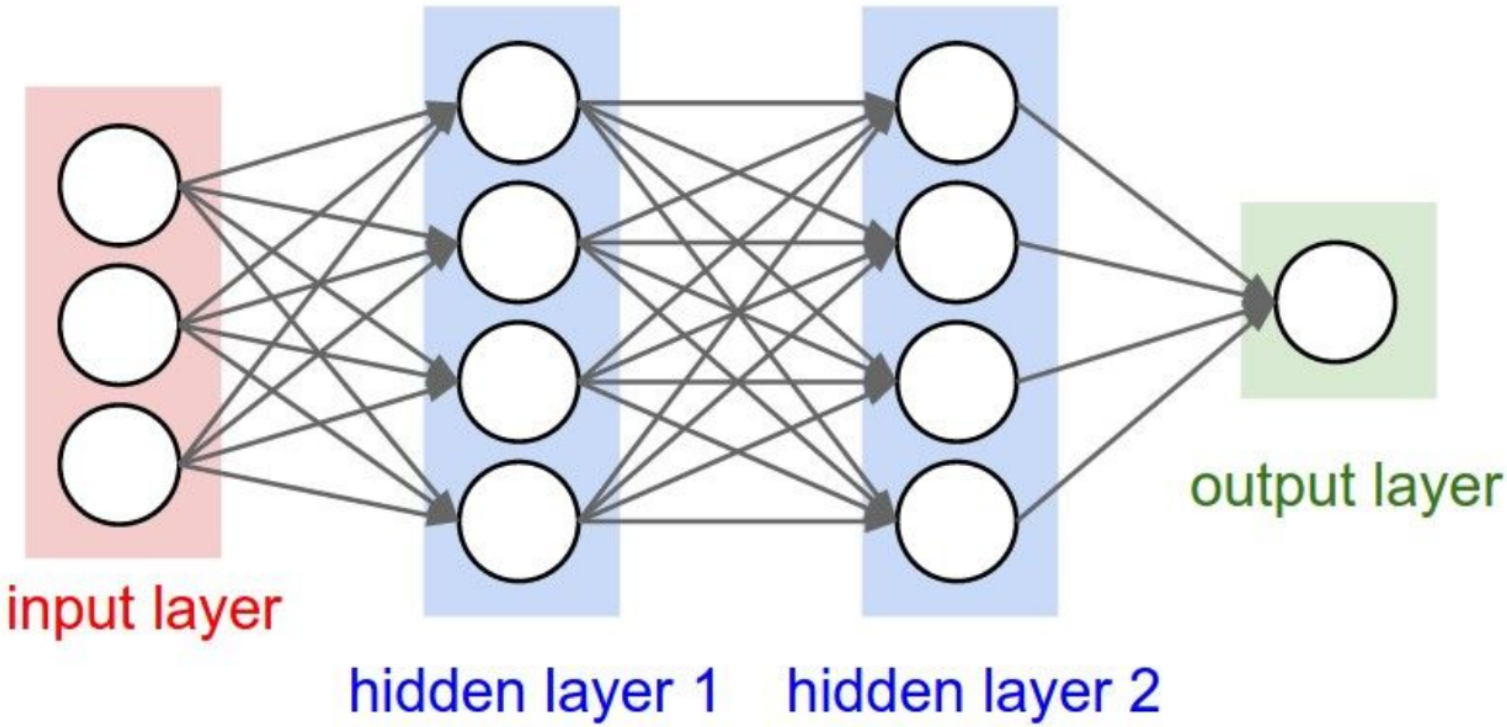
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



ReLU is a good default choice for most problems

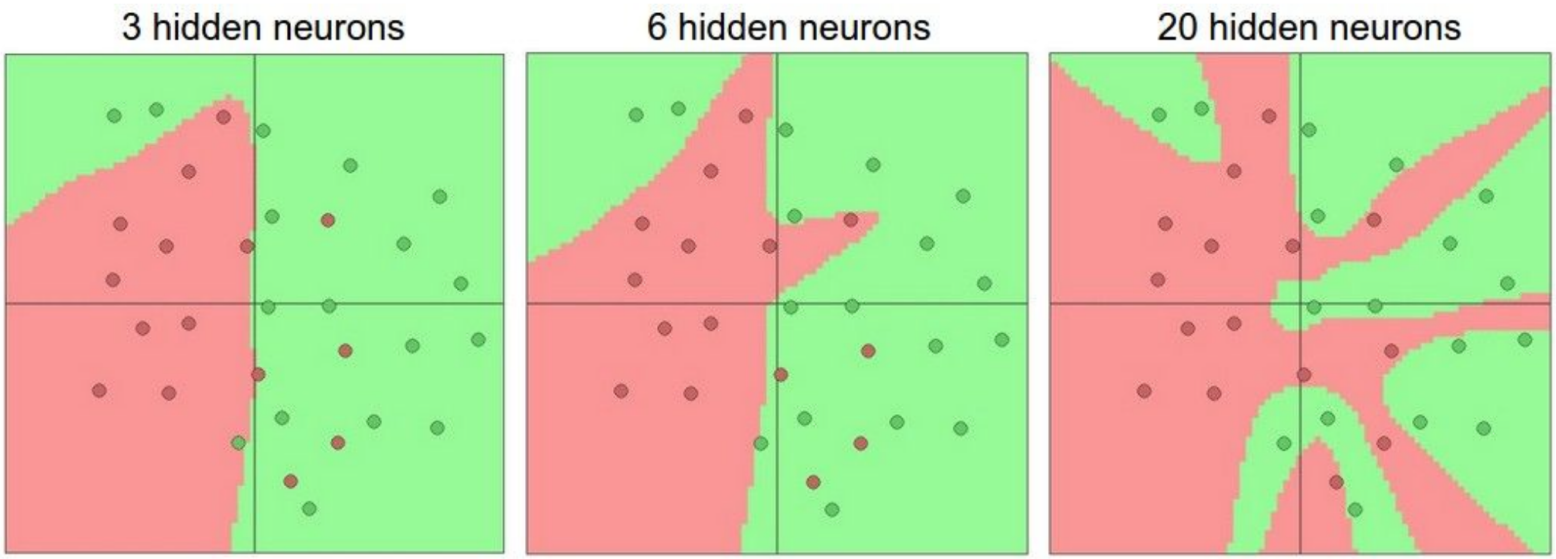
Example feed-forward computation of a neural network




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0 / (1.0 + np.exp(-x))  # activation function (use sigmoid)
x = np.random.randn(3, 1)  # random input vector of three numbers (3 × 1)
h1 = f(np.dot(W1, x) + b1)  # calculate first hidden layer activations (4×1)
h2 = f(np.dot(W2, h1) + b2)  # calculate second hidden layer activations (4×1)
out = np.dot(W3, h2) + b3  # output neuron (1×1)
```

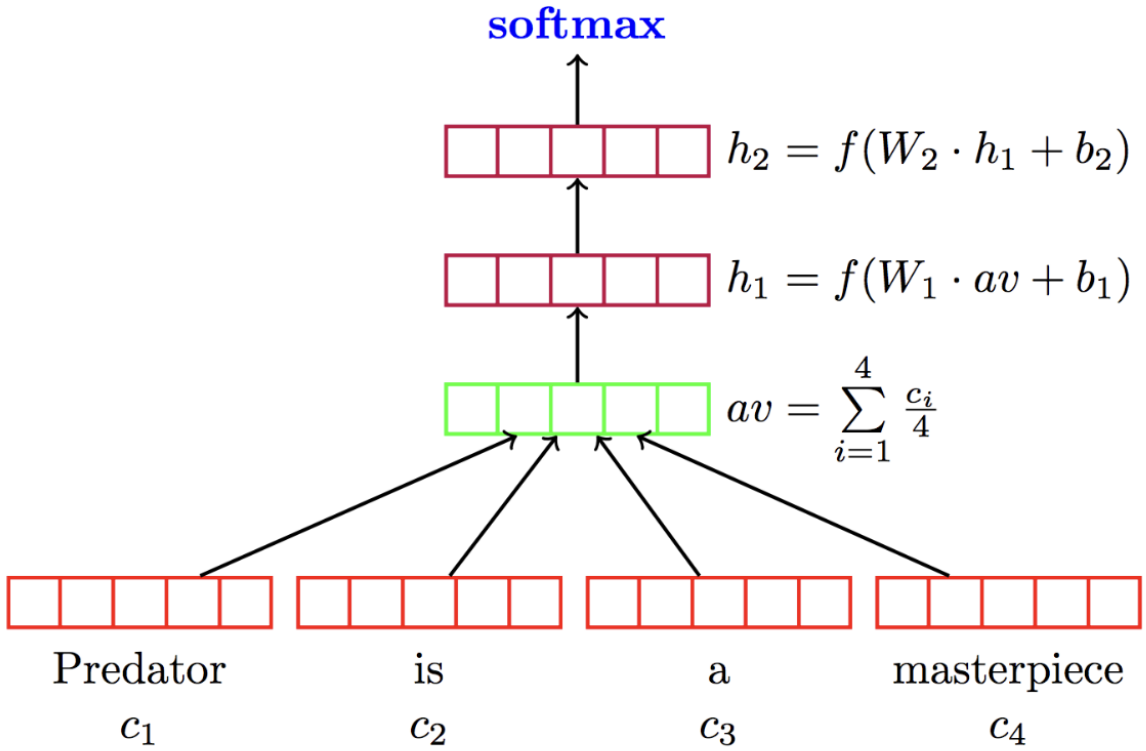
Setting the number of layers and their sizes

More layers, more neurons $\rightarrow$ more capacity to learn complex functions



FFN Architectures in NLP: Deep Averaging Networks (PA1)

- Deep Averaging Networks: feedforward neural network on average of word embeddings from input text



- Widely-held view: need to model syntactic structure to represent language
- DAN: Surprising that averaging can work as well as it does

Iyyer et al. (2015)

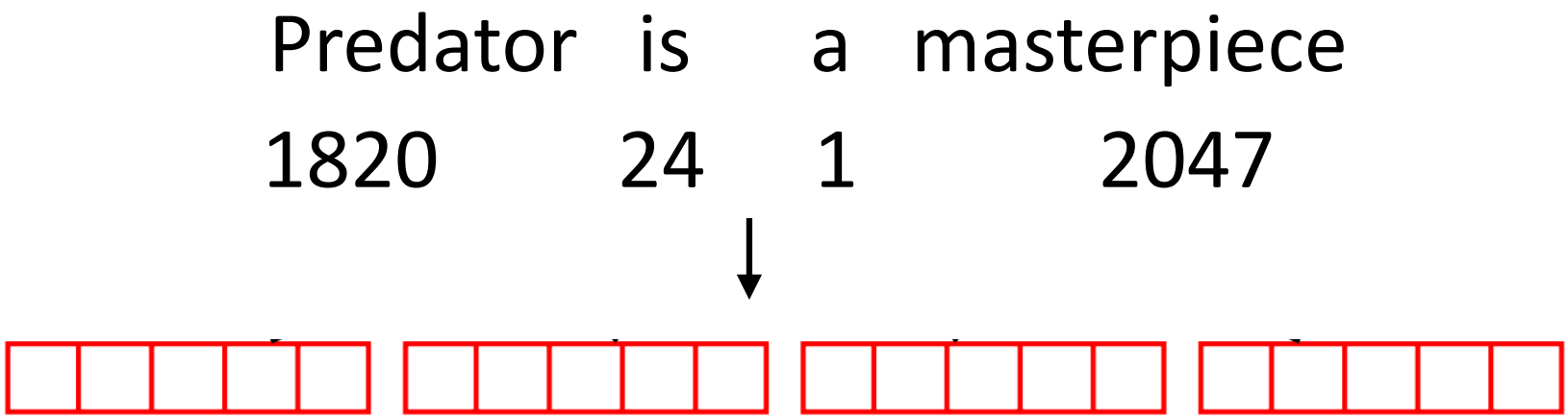
Deep Averaging Networks

Sentiment Analysis

	Model	RT	SST fine	SST bin	IMDB	Time (s)	
	DAN-ROOT	—	46.9	85.7	—	**31**	
	DAN-RAND	77.3	45.4	83.2	88.8	136	
	DAN	80.3	47.7	86.3	89.4	136	Iyyer et al. (2015)
Bag-of-words	NBOW-RAND	76.2	42.3	81.4	88.9	91	
	NBOW	79.0	43.6	83.6	89.0	91	
	BiNB	—	41.9	83.1	—	—	Wang and Manning (2012)
	NBSVM-bi	79.4	—	—	91.2	—	
Tree RNNs / CNNS / LSTMS	RecNN*	77.7	43.2	82.4	—	—	
	RecNTN*	—	45.7	85.4	—	—	
	DRecNN	—	49.8	86.6	—	431	
	TreeLSTM	—	**50.6**	86.9	—	—	
	DCNN*	—	48.5	86.9	89.4	—	
	PVEC*	—	48.7	87.8	**92.6**	—	
	CNN-MC	**81.1**	47.4	**88.1**	—	2,452	Kim (2014)
	WRRBM*	—	—	—	89.2	—	

Iyyer et al. (2015)

Word Embeddings in PyTorch



- `torch.nn.Embedding`: maps vector of indices to matrix of word vectors
- n indices $=>$ $n \times d$ matrix of d-dimensional word embeddings
- For PA1: see BOWModels.py for a simple example of a feedforward neural network in PyTorch
- PA1: but create your own Dataset class to load data

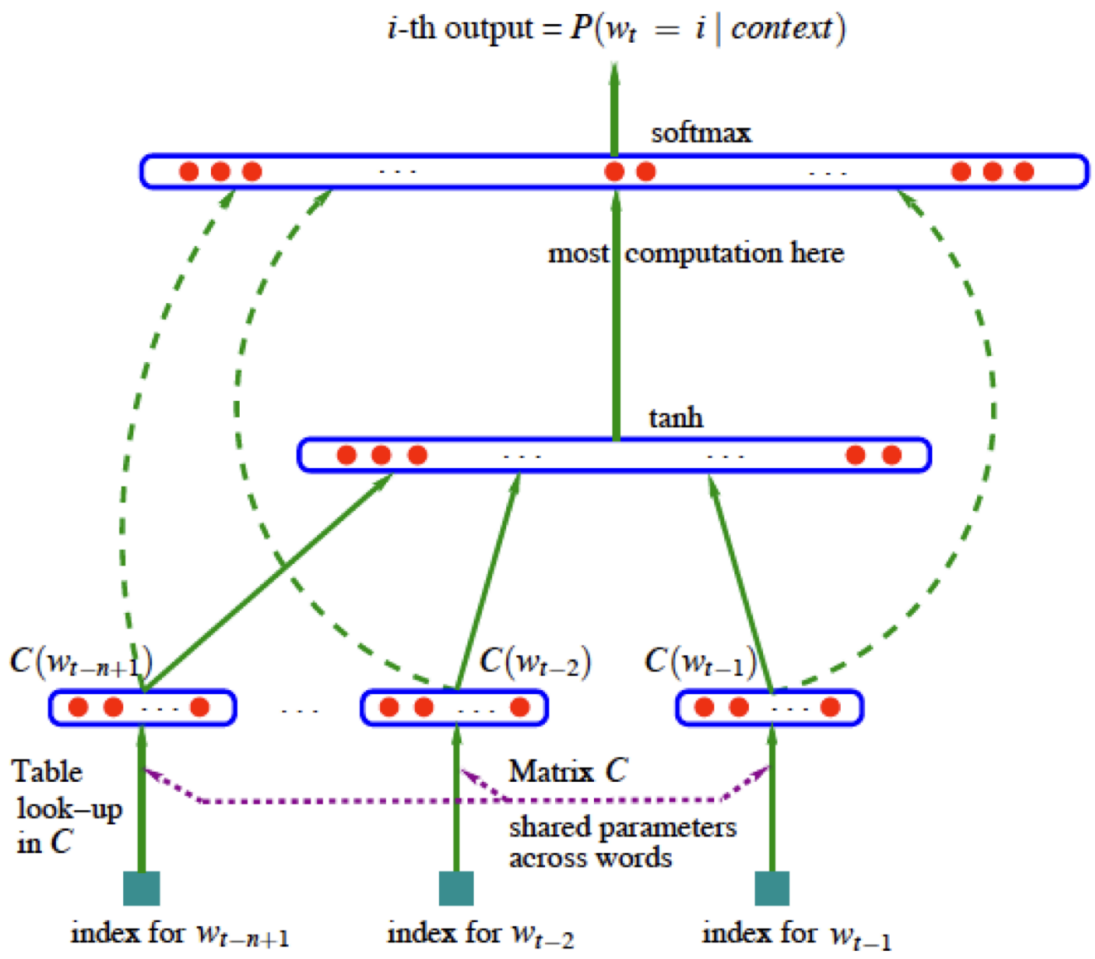
PA1 Tips

- see BOWModels.py for a simple example of a feedforward neural network in PyTorch
- Need to create your own Dataset class to feed data to the model
- As with the linear model, most minor tweaks like dropout, etc. will make $< 1\%$ difference. If you're 10% off the performance target, it's likely due to a mis-sized network, poor optimization, bugs, etc.

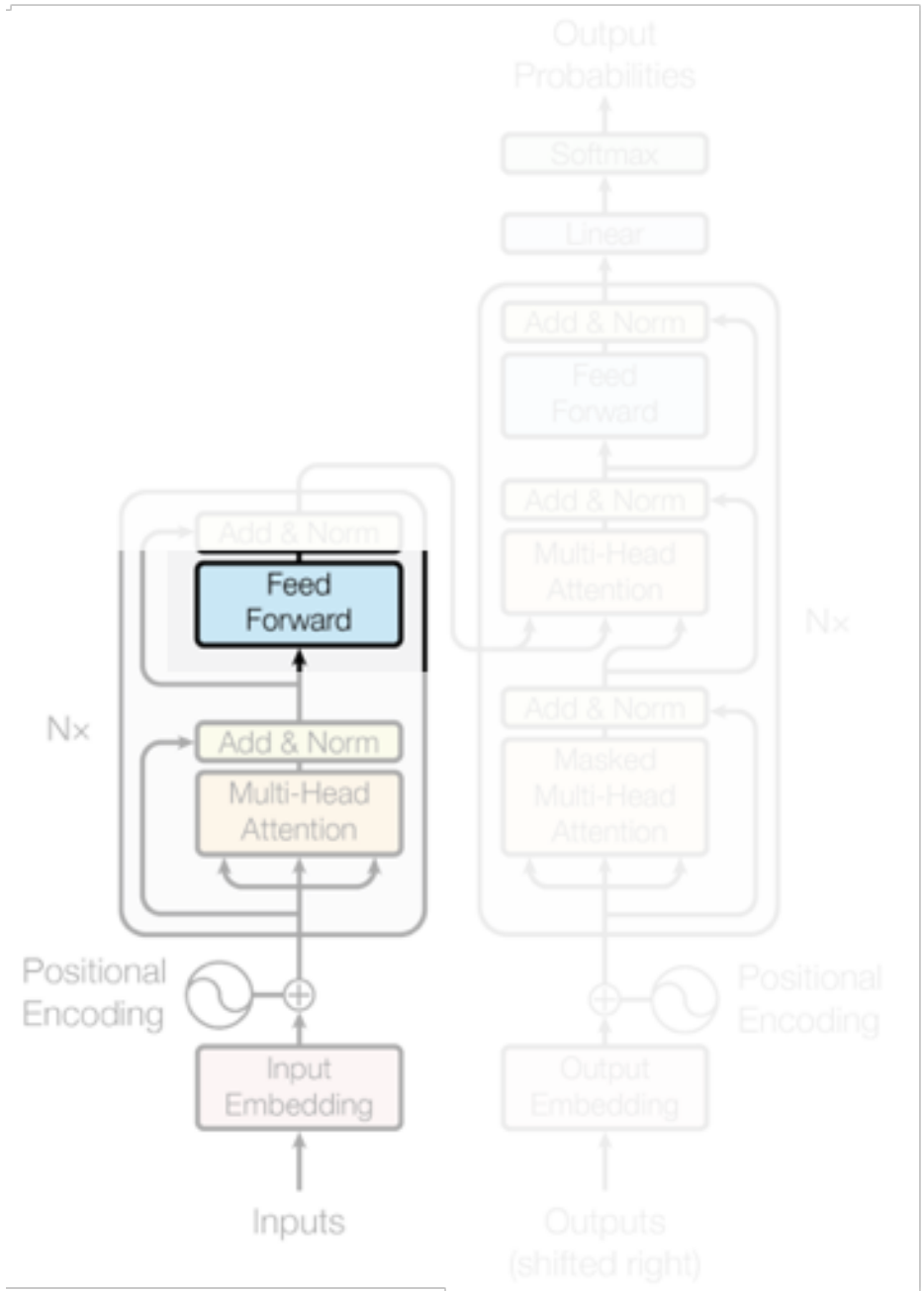
FFN Architectures in NLP: Seminal Neural Language Model

Bengio et al. (2003) proposed first neural language model: 1-hidden layer feed-forward neural network.

$$p = \text{softmax}(s)$$
$$s = \boldsymbol{W}_2 \boldsymbol{h}$$
$$\boldsymbol{h} = g\left(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}\right)$$
$$\boldsymbol{x} \text{ (input)}$$



FF Layers as Building Blocks in Deep Models



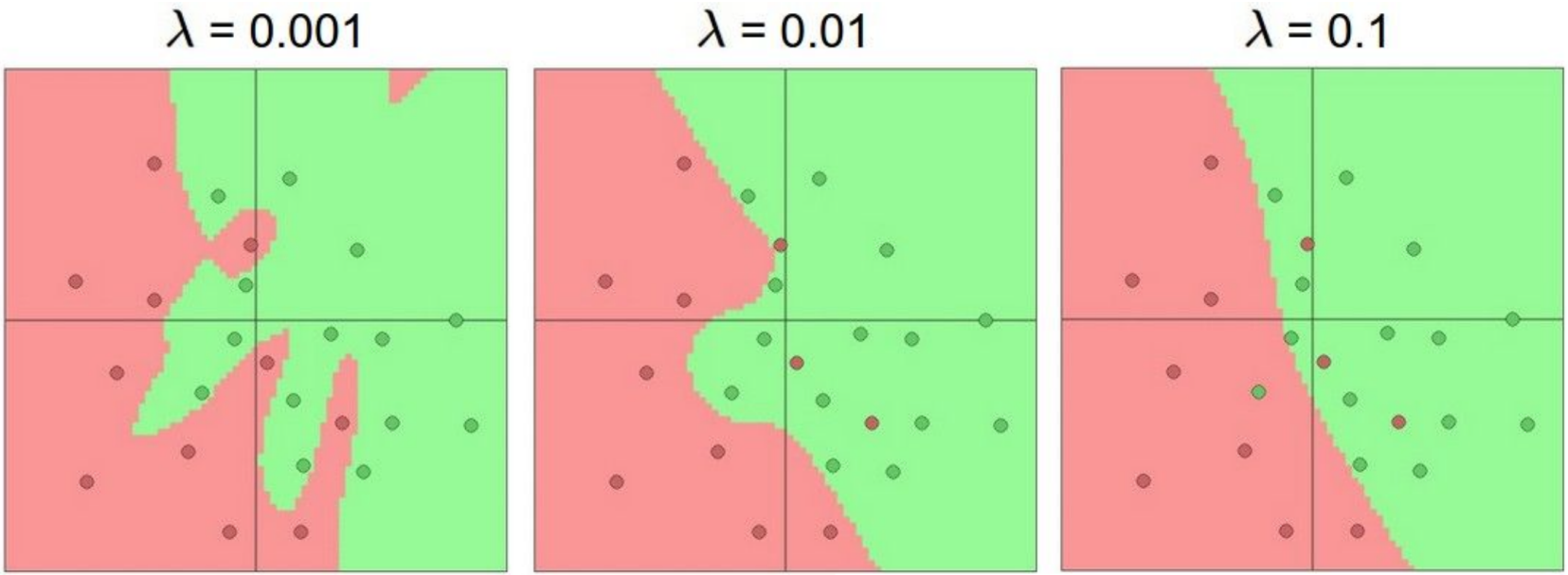
Loss: Plug in neural network in loss functions

$$s = f(x; W_1, W_2) = W_2 g(W_1 x) \quad \text{non-linear score function}$$

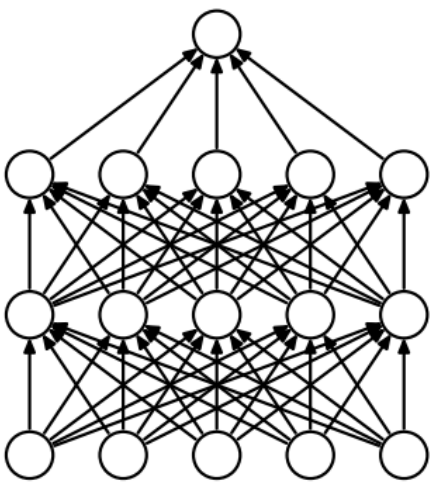
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \quad \text{loss on predictions}$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization}$$

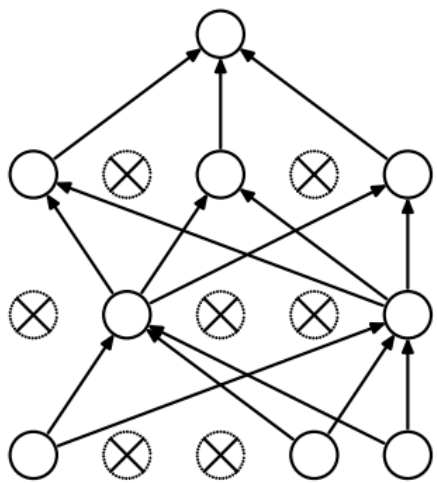
Regularization in Neural Networks



Regularization in Neural Networks: Dropout



(a) Standard Neural Net



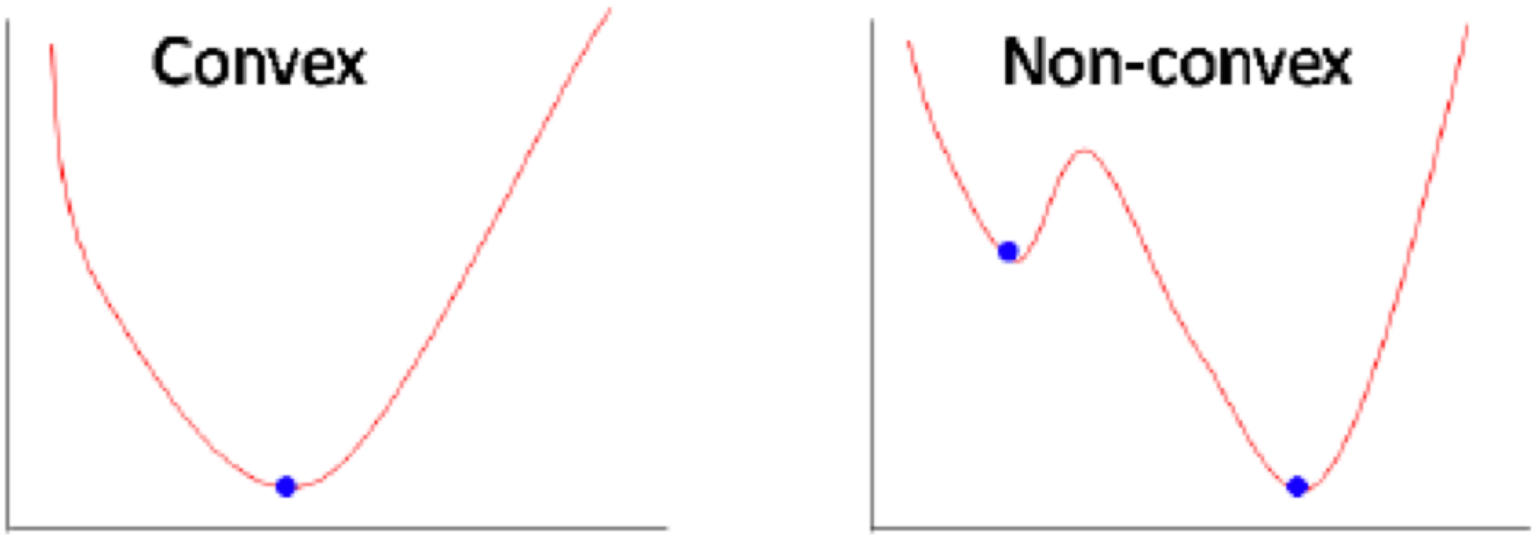
(b) After applying dropout.

- Training: randomly set some activations to zero to prevent overfitting
- Test time: use whole network
- Intuition: network needs to be robust to missing signals, so it has redundancy
- One line in Pytorch/Tensorflow

Srivastava et al. (2014)

Parameter Estimation in Neural Networks: Gradient Descent

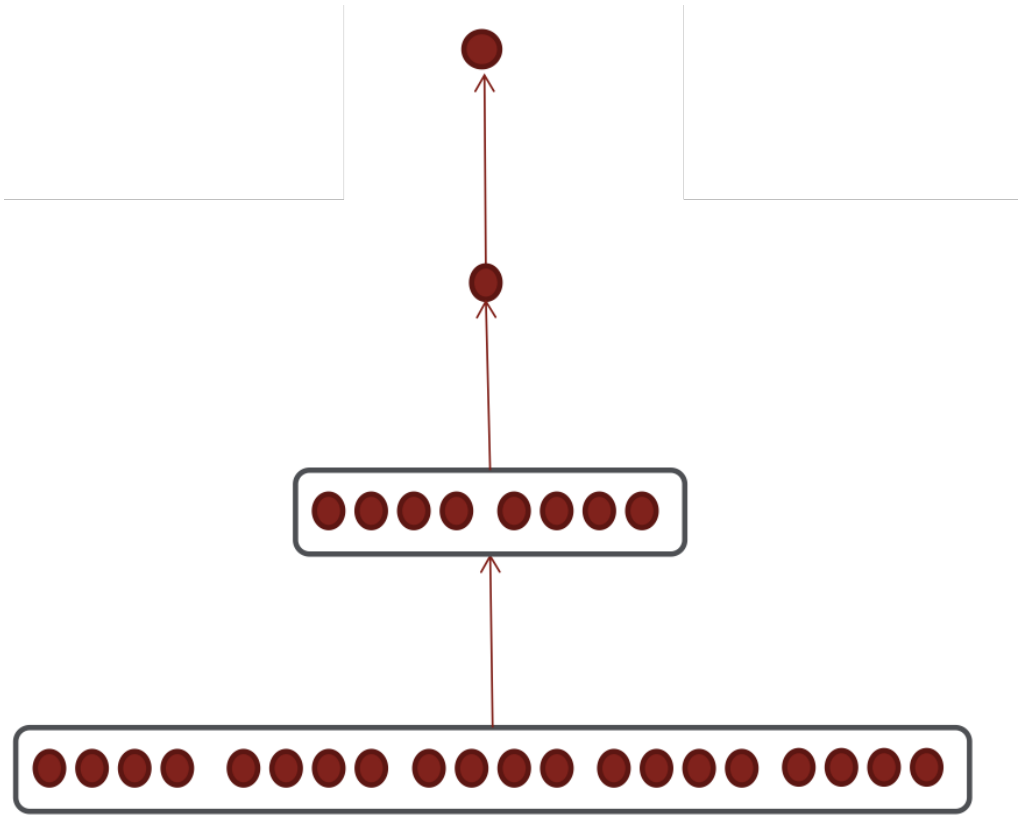
- Due to non-linearities, the neural network functions are not convex, and **gradient descent can get stuck in local minima**



In practice gradient descent works well on big complex neural models, why? there are **many winning tickets**. see *"The lottery Ticket hypothesis", ICLR 2019*

Text Classification with a FFN: Summary

Convert text to vector, apply FFN, pick label with highest score



$p = \text{softmax}(s)$

$s = \boldsymbol{W}_2 \boldsymbol{h}$

$\boldsymbol{h} = g(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b})$

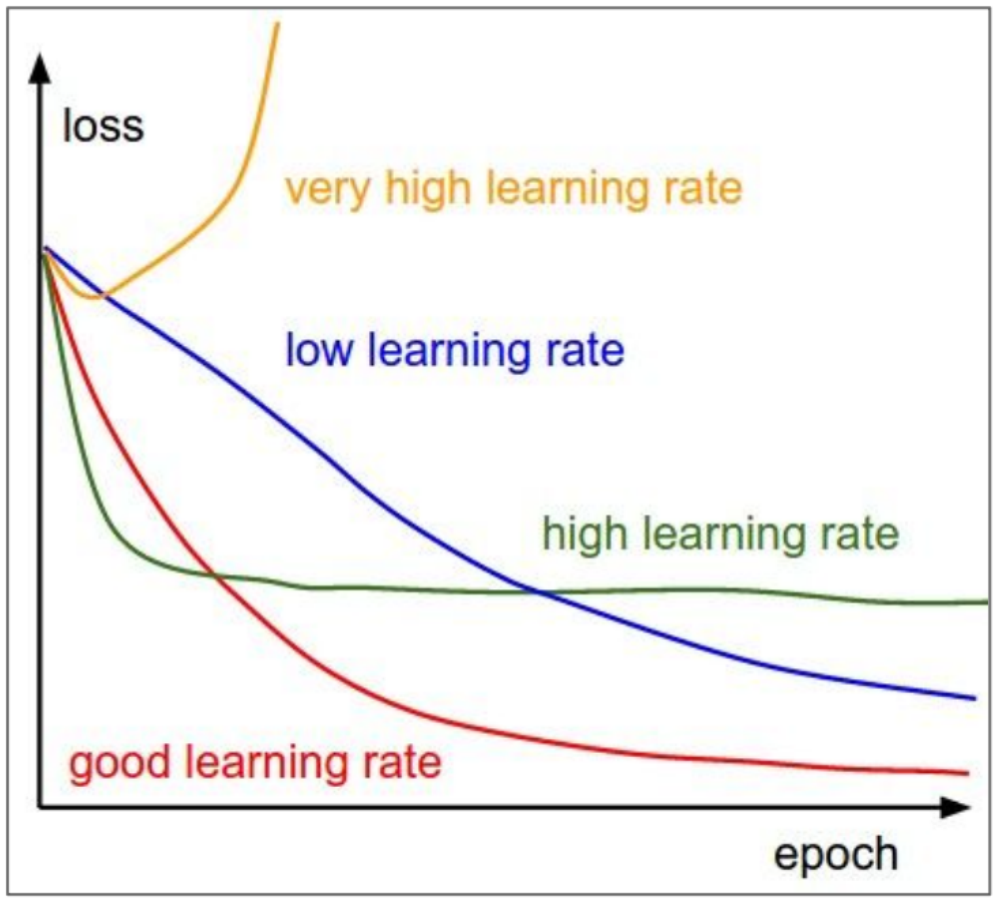
$\boldsymbol{x}$ (input)

Hyperparamters of Feedforward Neural Network

- **Number of layers**: 1, 2, 3, ...
- **Size of layers**: Number of neurons in each layer
- **Activation function**: ReLU, Sigmoid, ...
- **Regularization**: L2, Dropout, ...
- **Learning rate**: Step size in gradient descent
- ...

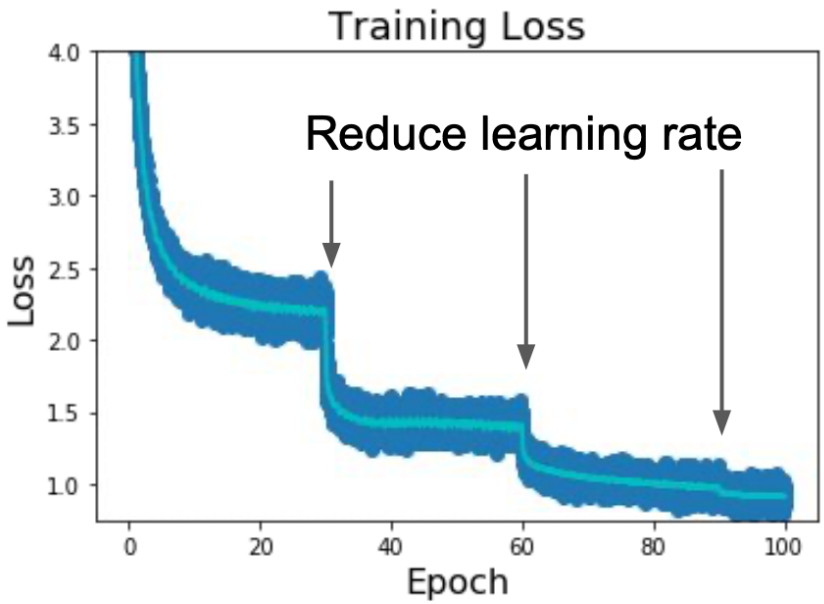
Learning rate is a crucial hyperparameter

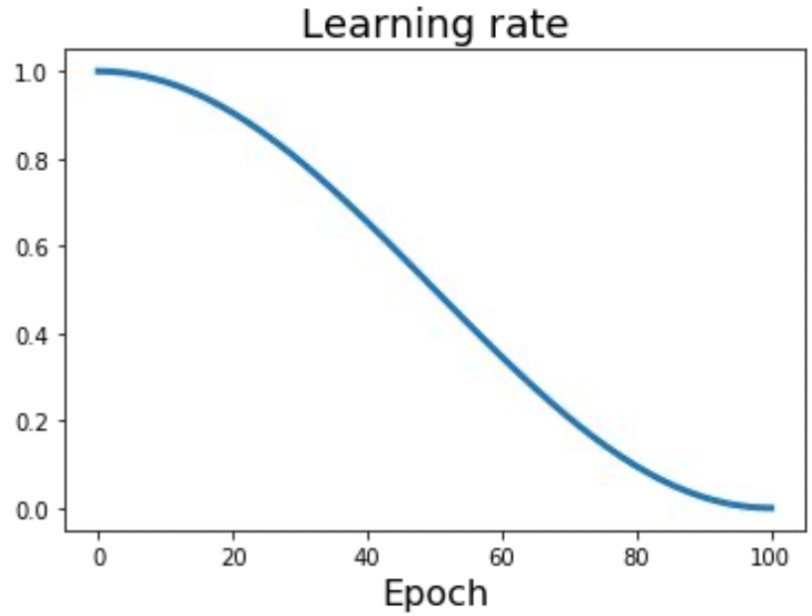
- **Learning rate**: Step size in gradient descent. Too small: slow convergence. Too large: overshoots the minimum



Learning Rate Schedules: change the learning rate over time

Can start with a high learning rate and then decrease it over time to improve convergence





- **Step**: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30,60 , and 90 .

- **Cosine**:
  $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(t\pi/T))$
  - α_0 : Initial learning rate
  - α_t : Learning rate at epoch t
  - T : Total number of epochs

Training Basics

Batching

- **Batching**: Training on a single example at a time is slow, batching data gives speedups due to more efficient matrix operations
  - **Stochastic Gradient Descent (SGD)**: Mini-batch size of 1, noisy updates
  - **Mini-Batch Gradient Descent**:Training on a small batch of examples at a time
- Batch size is a hyperparameter (32, 64, 128, ...)

Training a Model

Define your model

- For each epoch of training (one pass through the dataset)
  - For each batch of data
    - Compute loss on batch
    - Autograd to compute gradients
    - Take step with optimizer
- Evaluate model on validation set, adjust hyperparameters, then train again

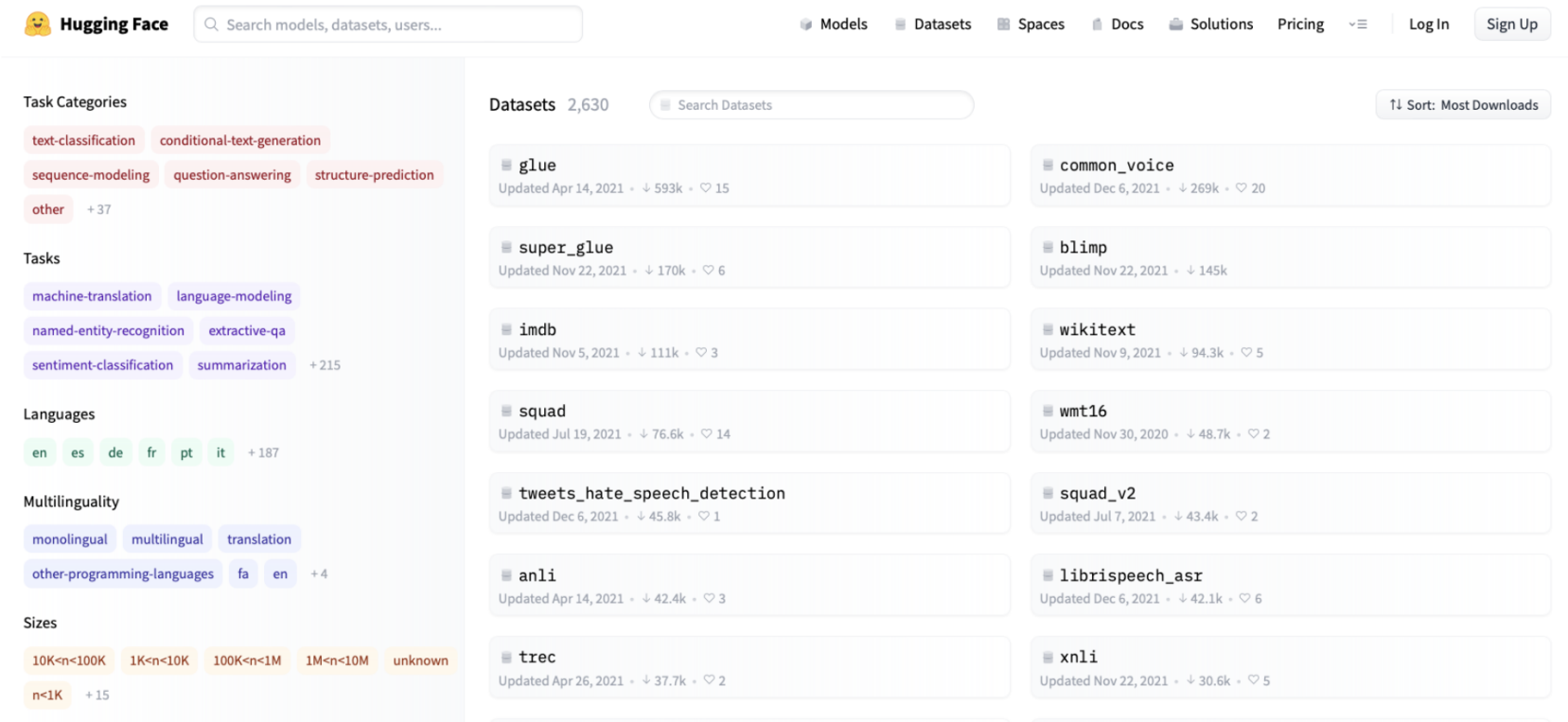
Evaluate on test set

Training notes

- Feedforward neural networks can be implemented easily in PyTorch
  - Different optimizers available: SGD, Adam, etc.
  - Different loss functions available: CrossEntropy, MSELoss, etc.
  - Different activation functions available: ReLU, Sigmoid, etc.
  - Different regularization techniques available: Dropout, L2, etc.
  - Different learning rate schedules available: Step, Cosine, etc.
- ... use the standard tricks to get good performance

Where can I find text classification data

NLP benchmarks



See: `https://huggingface.co/datasets`

Text Classification: we have looked at

1. The Text Classification Problem
2. Example Applications
3. Models (Logistic Regression, Feedforward Neural Networks)

Next Time: Word Representations

Questions?

Some slides are based on slides from the Stanford CS231n course by Fei-Fei Li et al.