EEN1083

Data analysis and machine learning I

Ali Intizar

DCU Ollscoil Chathair Bhaile Átha Cliath Dublin City University

Neural Networks

DCU Ollscoil Chathair Bhaile Átha Cliath Dublin City University

Outline

1. History of AI, connectionism, and neural networks
2. Single neuron models
3. Multiple outputs
4. Multi-layer perceptrons

History of AI, connectionism, and neural networks

1950 and 60s

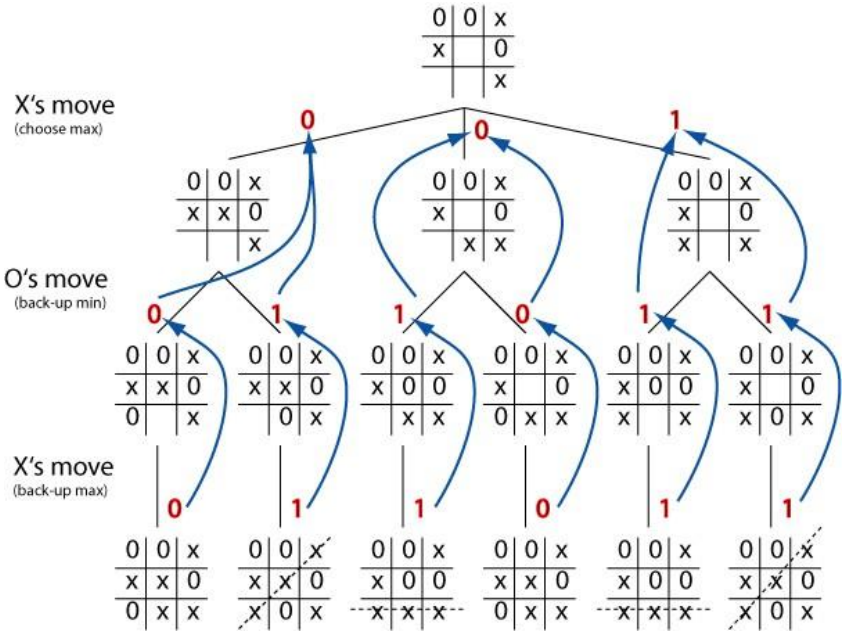
- Alan Turing
- First AI conference 1956
- Great expectations!
- Lots of funding from agencies like DARPA
- Many people at the time predicted AI would be solved in 20 years.

Achievements of the 50s and 60s

Early work in **reasoning as search**

- Graph search
- Backtracking
- Heuristics

Appreciation of the combinatorial explosion in complexity

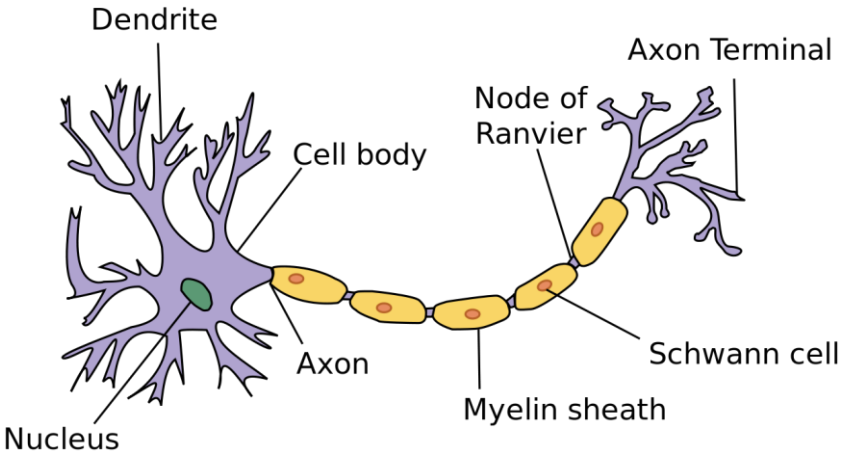


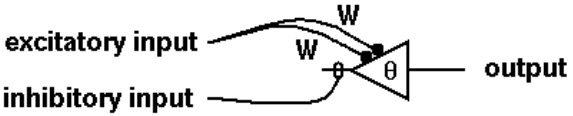
Achievements of the 50s and 60s

Neural networks

McCullough and Pitts model (1943) of an artificial neuron

- Fixed preset weights
- No learning algorithm





Achievements of the 50s and 60s

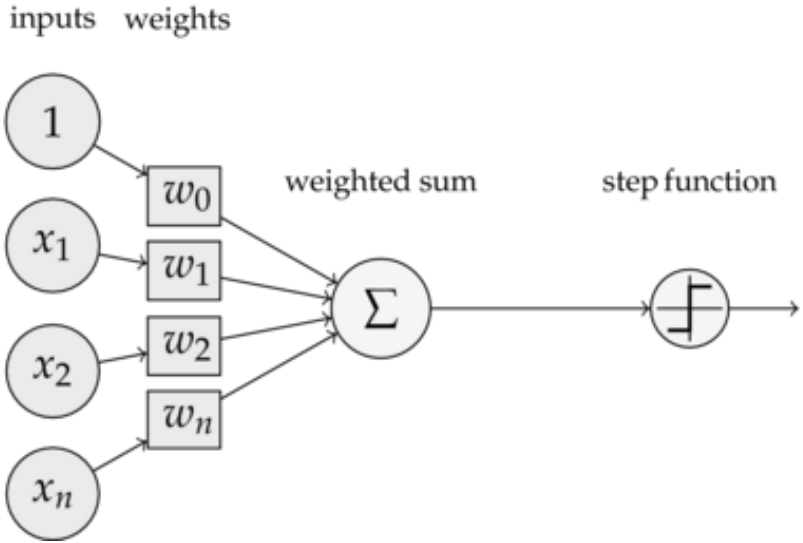
Rosenblatt's Perceptron (1958)

- Generalized model
- Simple algorithm for learning weights!

Generated a lot of excitement.

Wild claims were made of what perceptrons could learn.

Beginning of **connectionism**



Aside: the perceptron learning algorithm

Prediction function

Linear prediction function:

$$f(\mathbf{x}) = H(\mathbf{w}^T \mathbf{x}),$$

where H is the heaviside step function $H(x) = \mathbb{1}[x > 0]$.

Learning algorithm

For each example $(\mathbf{x}_i, y_i)$ update the weights with:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha(\hat{y}_i - y_i)\mathbf{x}_i$$

Rosenblatt's perceptron learning algorithm is equivalent to stochastic (sub)-gradient descent using the hinge loss with no L_2 regularizer. Strong connection with Linear SVM.

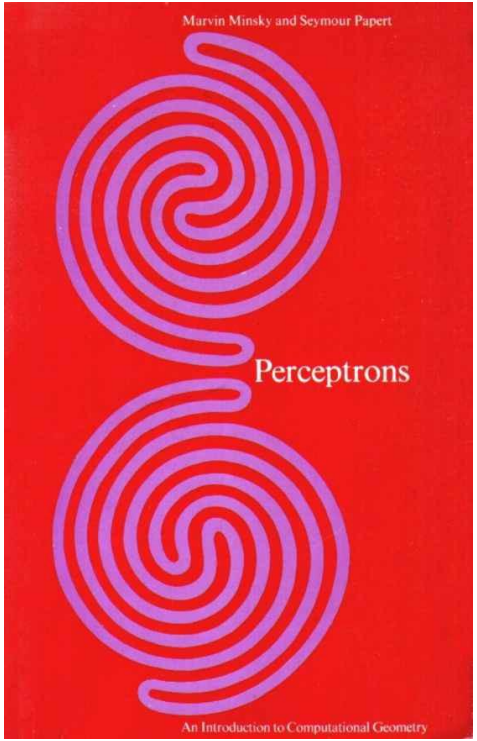
Achievements of the 50s and 60s

Marvin Minsky wrote ***Perceptrons*** in 1969

Showed that some simple logical operations could not be performed by simple perceptrons.

A single perceptron cannot implement an XOR gate.

Interest in connectionism and neural networks stagnated for over a decade



1970s

Some early work into **“expert systems”**

- Medical diagnosis system **Mycin** (1970s, Stanford, infectious diseases). Appropriate therapy in 69% cases.

First **AI winter 1974-1980**:

- Unrealistic expectations
- Computational intractability
- Lack of computing power
- Perceptrons book and death of connectionism
- Lack of funding

1980s

Rise of **expert systems**

- Hand coded rules (if-this-then-that)
- “Knowledge engineering”
- Expert systems became some of the first truly successful forms of artificial intelligence (AI) software
- XCON: rule-based system to assist in ordering computer components
  - Saved DEC 40 million dollars annually

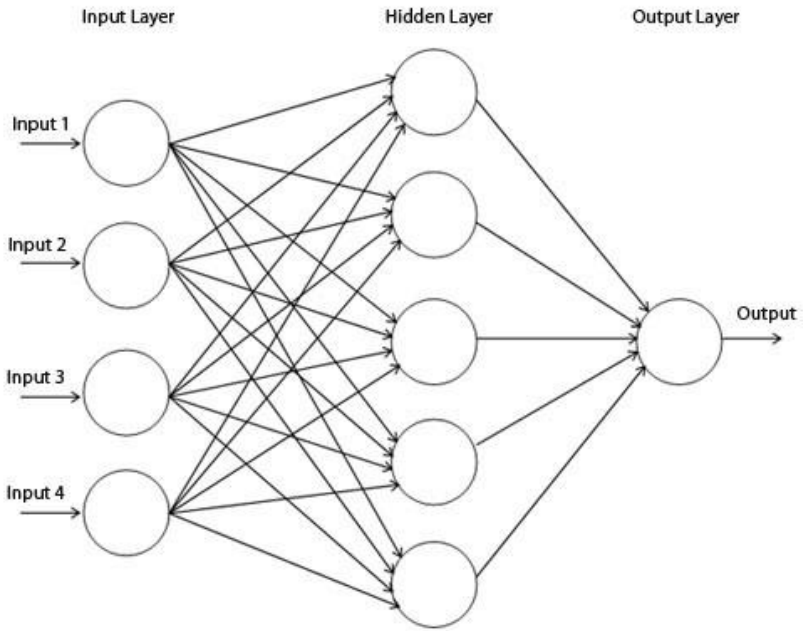
More powerful reasoning as search systems

Better chess playing programs (DeepThought) defeat masters 1989

1980s

Revival of connectionism and neural networks

- Idea of stacking together perceptrons had been around a while
  - Multi-layered perceptron, neural network
- **Backpropagation**: practical algorithm for learning the weights
  - 1982: Werbos
  - 1986: Rumelhart, Hinton, Williams
  - Basically (stochastic) gradient descent using the chain rule



Late 80s: Second AI winter

Unrealistic expectations in expert systems

People still couldn’t fit multi-layer perceptrons well

- Computational issues
- Slow convergence
- Overfitting

Many problems remained: especially in **perception** (vision, speech recognition, NLP).

Funding cut

1993-2009

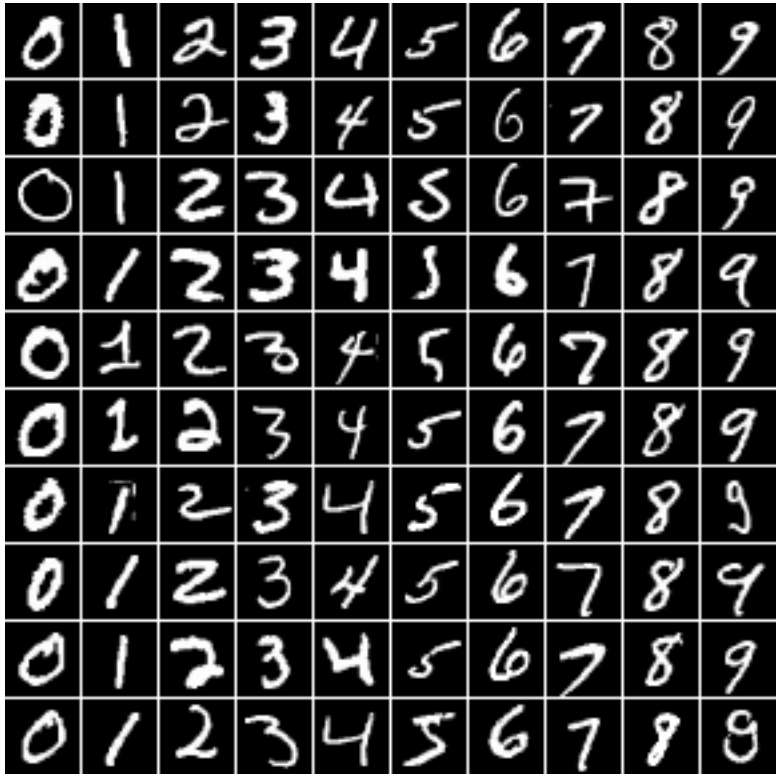
Rise in interest in **statistical machine learning**: learning rules from data

Invention of **support vector machines**

Renewed interest in neural networks

Early successful **convolutional neural networks** (Yann LeCun)

Progress made in computer vision and audio using feature engineering and machine learning techniques



Modern AI

Resurgence of interest in neural networks circa. 2007

- New algorithms capable of training **deep neural networks**
- Rebranding of neural nets as deep learning
- Significantly increased computational power (GPUs)
- Larger annotated datasets (big data)
- Several key innovations making neural nets easier to train and less prone to overfitting.

Rapid progress in computer vision, audio speech processing, natural language processing, machine translation, reinforcement learning, ...

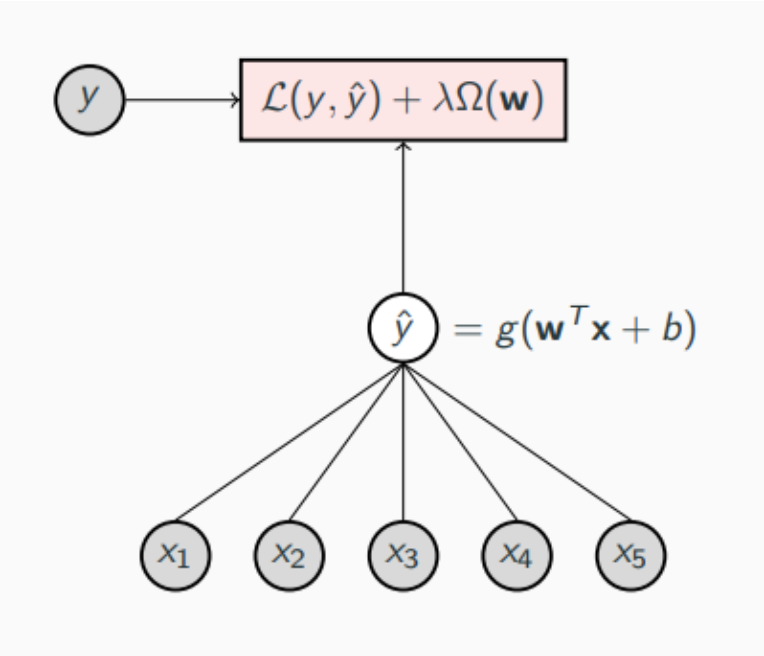
Single neuron models

The single neuron model

A linear model

Parts:

- Input features $\mathbf{x} = \{x_1, \ldots, x_D\} \in \mathrm{R}^D$
- Unit (activation function or non-linearity): $g : \mathrm{R} \to \mathrm{R}$
- Loss function $L(y, \hat{y})$
- Regularization $\Omega(\mathbf{w})$



Linear regression as a single neuron

Activation function: identity

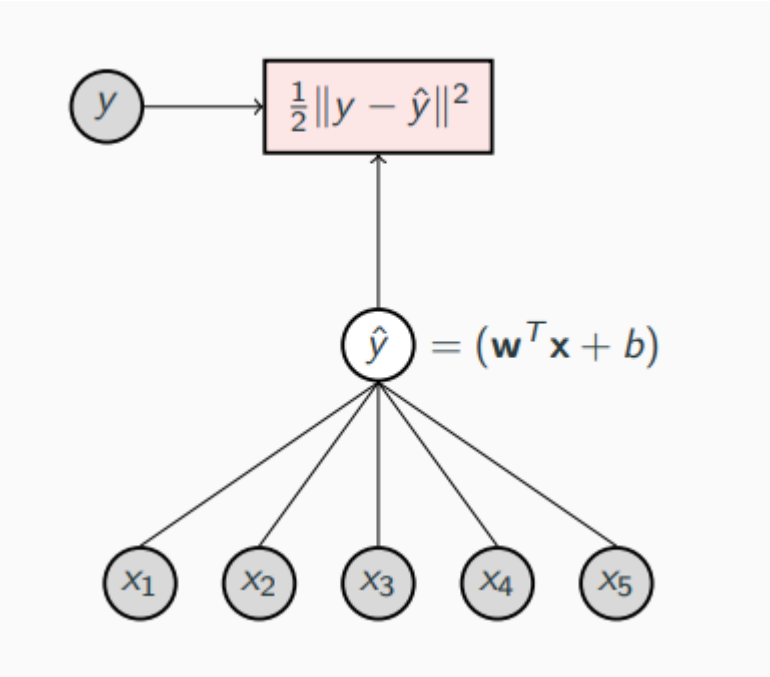
$g(x) = x$

Loss function: Euclidean

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}\|y - \hat{y}\|^2$$

Regularization: None

$\Omega(\mathbf{w}) = 0$



Ridge regression as a single neuron

Activation function: identity

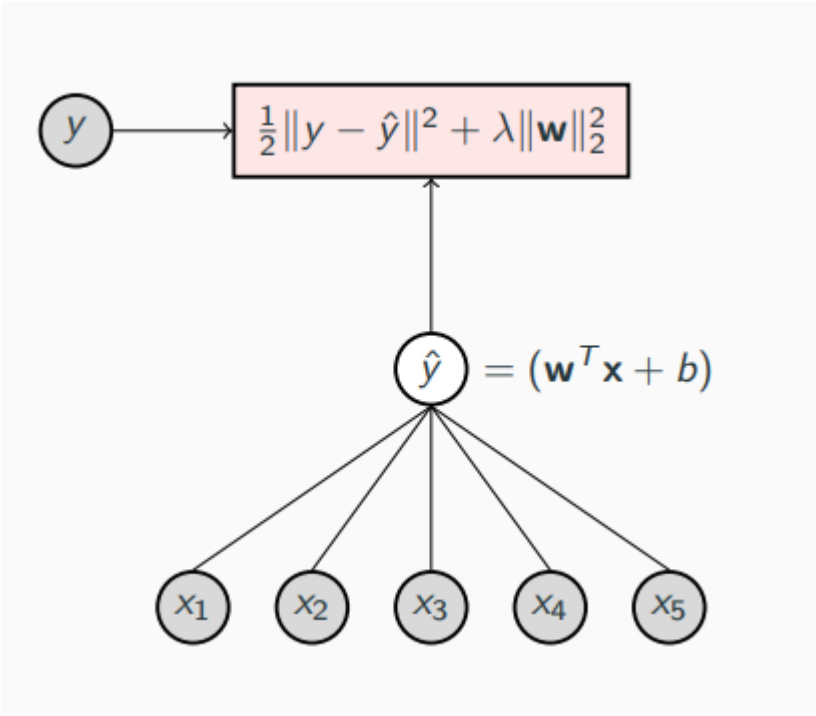
$g(x) = x$

Loss function: Euclidean

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}\|y - \hat{y}\|^2$$

Regularization: L_2

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2$$



Lasso as a neuron

Activation function: identity

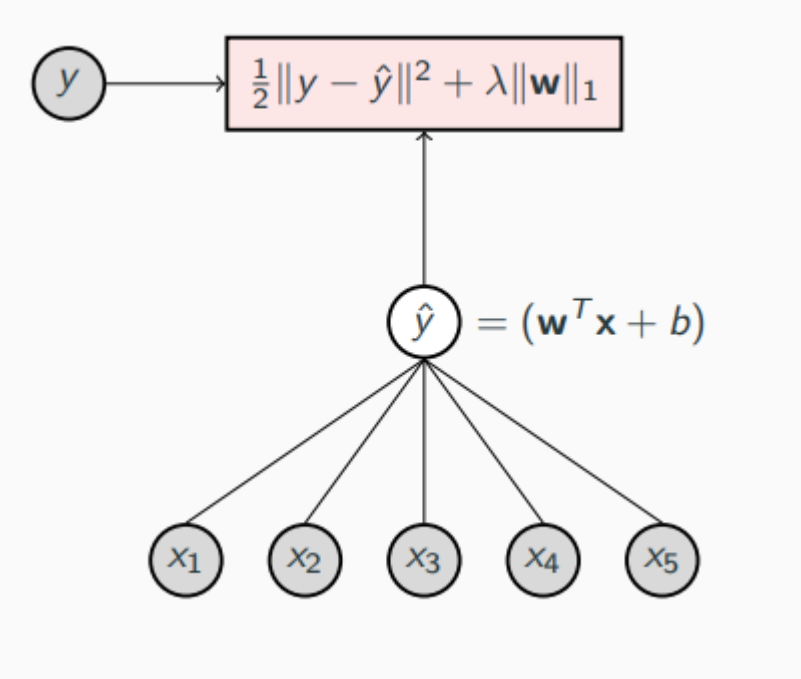
$g(x) = x$

Loss function: Euclidean

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}\|y - \hat{y}\|^2$$

Regularization: L_1

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_1$$



Logistic regression as a single neuron

Activation function: sigmoid

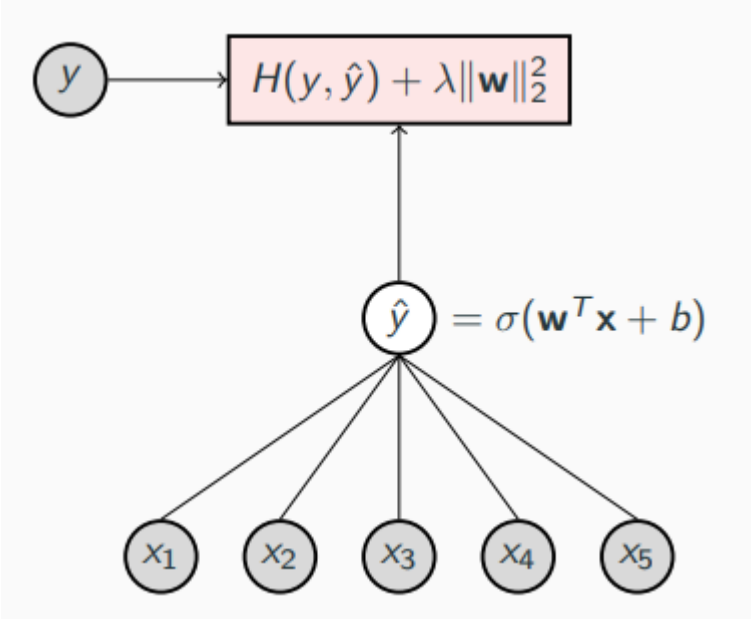
$g(x) = \sigma(x)$

Loss function: cross entropy

$$\begin{aligned}\mathcal{L}(y, \hat{y}) &= H(y, \hat{y}) \\ &= -y \log \hat{y} \\ &\quad - (1 - y) \log(1 - \hat{y})\end{aligned}$$

Regularization: None or L_2

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2$$



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

Linear SVM as a single neuron

Activation function: identity

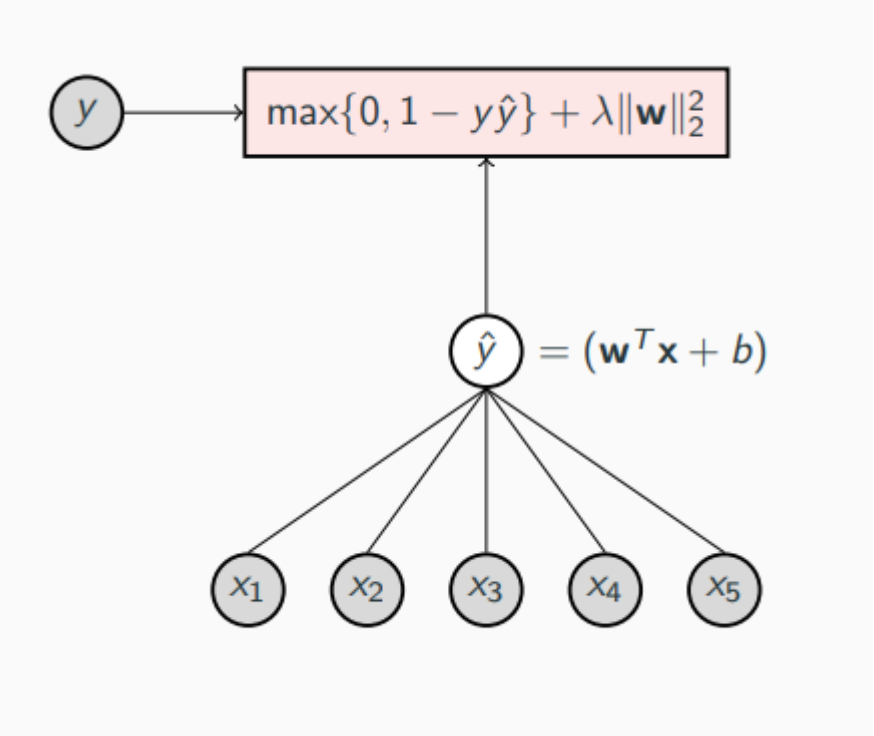
$g(x) = x$

Loss function: hinge

$\mathcal{L}(y, \hat{y}) = \max\{0, 1 - y\hat{y}\}$

Regularization: L_2

$\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2$



Summary of single neuron models

Algorithm	y	$f(x)$	$\mathcal{L}(y, \hat{y})$	$\Omega(\mathbf{w})$
Linear reg.	$\mathbb{R}$	$\mathbf{w}^T\mathbf{x} + b$	$\frac{1}{2}\|y - \hat{y}\|^2$	0
Ridge reg.	$\mathbb{R}$	$\mathbf{w}^T\mathbf{x} + b$	$\frac{1}{2}\|y - \hat{y}\|_2^2$	$\|\mathbf{w}\|_2^2$
Lasso	$\mathbb{R}$	$\mathbf{w}^T\mathbf{x} + b$	$\frac{1}{2}\|y - \hat{y}\|^2$	$\|\mathbf{w}\|_1$
Logistic reg.	$\{0, 1\}$	$\sigma(\mathbf{w}^T\mathbf{x} + b)$	$-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$	$\|\mathbf{w}\|_2^2$
Linear SVM	$\{-1, 1\}$	$\mathbf{w}^T\mathbf{x} + b$	$\max\{0, 1 - y\hat{y}\}$	$\|\mathbf{w}\|_2^2$

Properties of these models

Interpretable:

- Large weights mean features are important to prediction.
- Small weights mean features are unimportant.

Usually (strongly) convex:

- Single unique minimum solution.
- No local minima in loss function.
- Optimal solution can be found using gradient descent.

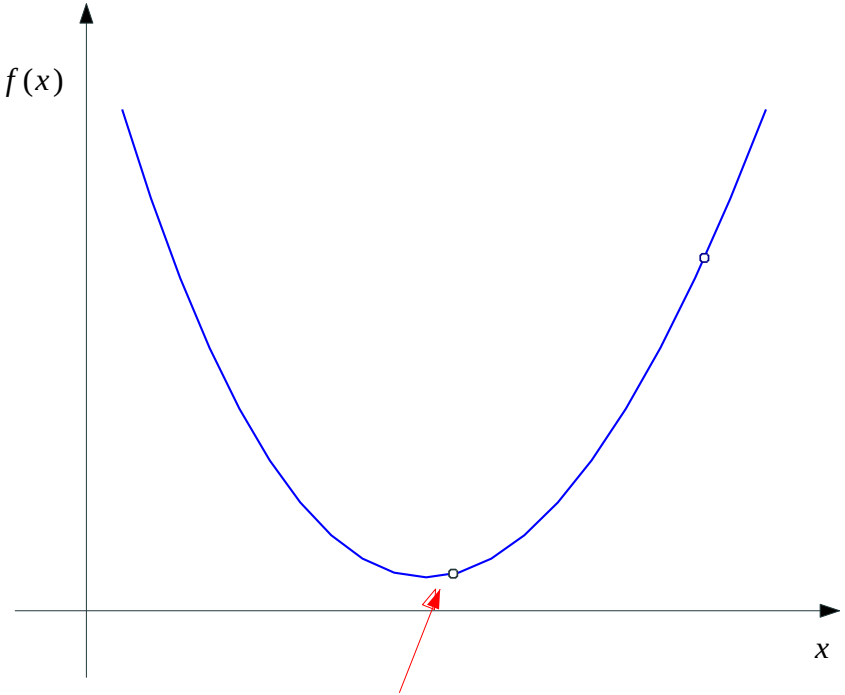
Fast predictions:

- Prediction just involves multiplying weights by features and adding up.

Convex functions

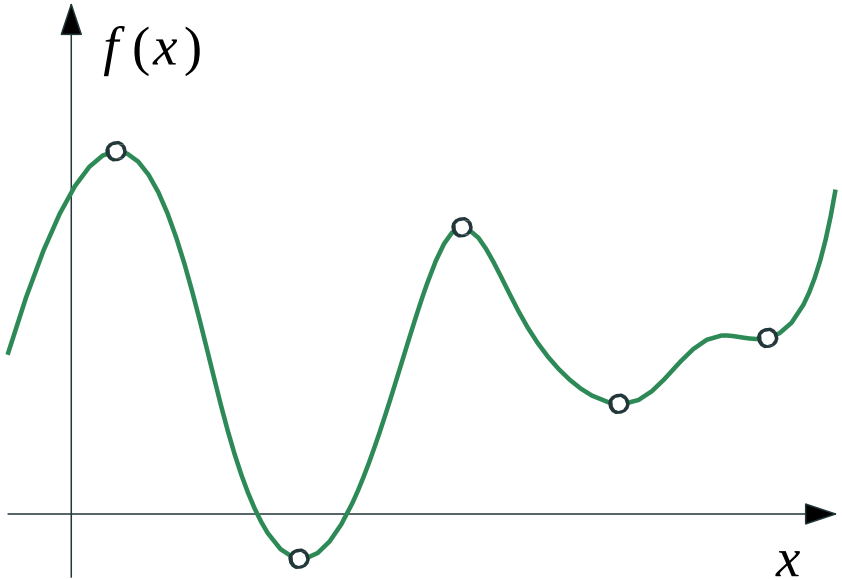
A function f is convex if,

Local minimum is global minimum



Non-convex functions

- Lots of local minima
- Saddle points
- Plateaus
- More difficult to optimize



Loss functions for multi-layer perceptrons are non-convex.

Fitting single neuron models

Most linear models have fast specialized solvers available. E.g:

- For linear regression, can just solve the normal equations $X^T X \mathbf{w} = X^T \mathbf{y}$ or compute $\mathbf{w} = (X^T X)^{-1} X^T \mathbf{y}$
- For ridge regression solve $(X^T X + \lambda I)\mathbf{w} = X^T \mathbf{y}$

All of the single neuron models so far can also be solved using gradient descent and stochastic gradient descent (SGD).

Although this is usually slower, SGD works for a broader class of models.

Gradient descent

The problem:

$$\underset{\mathbf{w},b}{\text{minimize}} \quad \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y_i,\hat{y}_i) + \lambda\Omega(\mathbf{w})$$

with

$$\hat{y}_i = f(\mathbf{x}_i) = g(\mathbf{w}^T\mathbf{x}_i + b)$$

Gradient descent

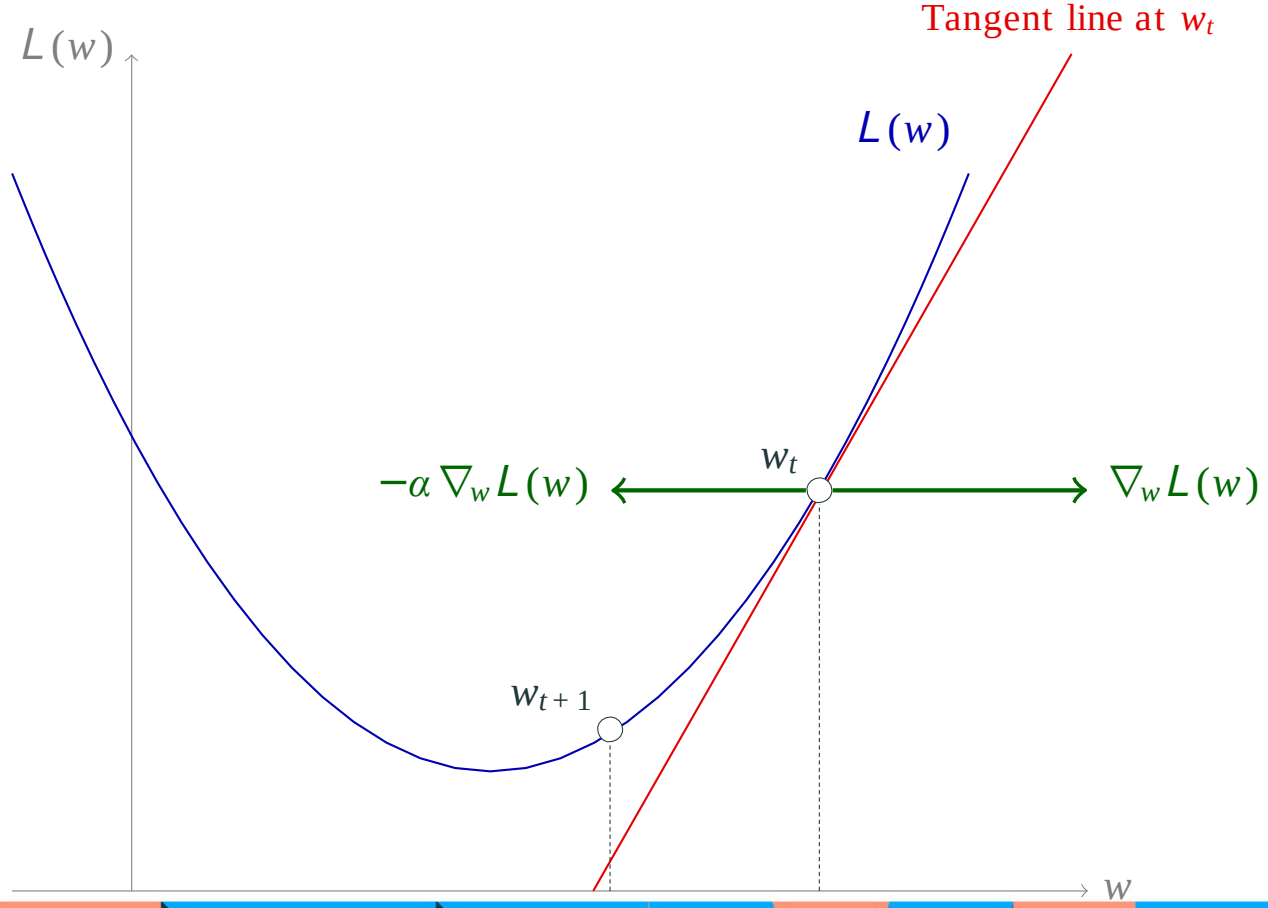
Algorithm 1 Gradient descent for single neuron models

```
while not converged and epoch < max_epochs do
    for i = 1 ... N do
        ŷ_i ← g(w^T x_i + b)
    w ← w − α [ (1/N) Σ_{i=1}^{N} ∇_w L(y_i, ŷ_i) ]        ▷ Data loss
    w ← w − α [ λ ∇_w Ω(w) ]                                ▷ Regularization
    b ← b − α [ (1/N) Σ_{i=1}^{N} ∇_b L(y_i, ŷ_i) ]        ▷ Bias
    Update learning rate α
    epoch ← epoch +1
```

Convergence:

- When the loss stops changing significantly.
- When the parameters $\mathbf{w}, b$ stop changing significantly.

Gradient descent



Properties of gradient descent

The learning rate α:

- Gradient descent will converge on convex functions as long as the learning rate α keeps getting smaller over time.
- Gradient descent may be very slow if α is too small.
- Gradient descent may diverge if α is too large.

Gradient descent requires a full pass over the dataset each time it updates the parameters (weights and biases). Slow for large datasets!

Stochastic gradient descent

Idea: Instead of computing the gradient of the loss for the full dataset, estimate it using a **single randomly chosen example** and then take step in that direction.

Algorithm 2 Stochastic gradient descent for single neuron models

```
while not converged and iter < max_iters do
    pick a random training example x_i, y_i
    ŷ_i ← g(w^T x_i + b)
    w ← w − α [∇_w L(y_i, ŷ_i)]                ▷ Data loss
    w ← w − α [(λ/N) ∇_w Ω(w)]                 ▷ Regularization
    b ← b − α [∇_b L(y_i, ŷ_i)]                ▷ Bias
    Update learning rate α
    iter ← iter +1
```

Properties of SGD

Usually much faster than batch gradient descent, since we update the weights every iteration instead of every epoch.

Can operate on large datasets, since we only need to process a single example at a time.

Important to shuffle training data to ensure the gradient estimates are unbiased (i.i.d.)

But: gradient updates are noisy, since it is only an estimate of the full gradient. Convergence may be slower as you near the optimum solution.

Mini-batch SGD

Combine the benefits of full gradient descent and SGD.

At each step, estimate the total gradient using the average gradient on a small **mini batch** of training examples (e.g. 10-100 examples).

Advantage

- Can operate on large datasets like SGD
- Less noisy gradients and better convergence properties than SGD
- Can take advantage of parallelism in modern GPU/CPU hardware.

Implementing with autodiff

You can use automatic differentiation (autodiff) so you don't need to calculate formulas for gradients by hand!

Most modern deep learning libraries (e.g. PyTorch, TensorFlow) support autodiff.

Just need to specify the prediction function, loss function, and regularization.

Implementing with PyTorch

```python
import torch

# generate some synthetic training data
N, D= 100, 1
x = torch.rand(N, D) * 5
y_true = 2.6 + 5.3 * x
y_obs = y_true + 0.8 * torch.randn(N, D)

# model parameters
w= torch.randn(D, requires_grad=True)
b = torch.randn(1, requires_grad=True)

# prediction function
def predict(X):
    return torch.unsqueeze(torch.matmul(X, w) + b, 1)

# mean squared error loss
def mse_loss(y_pred, y_true):
    return torch.mean((y_pred - y_true) ** 2)

# ridge regression loss
def regularized_loss(y_pred, y_true, reg_coeff=0):
    return mse_loss(y_pred, y_true) + reg_coeff * (w**2).sum()

# helper function to zero gradients
def zero_gradients(*params):
    for p in params:
        p.grad.zero_()
```

```python
# training hyperparameters
nb_epochs = 500
lr = 0.01
regularization_coeff = 1e-4

# training loop
losses = []

for epoch in range(nb_epochs):

    # make prediction
    y_pred = predict(x)

    # compute the loss
    loss = regularized_loss(y_pred, y_obs, regularization_coeff)

    # reverse mode autodiff
    loss.backward()

    # gradient descent (don't track gradients)
    with torch.no_grad():
        w-= lr * w.grad
        b -= lr * b.grad

    # set gradients to zero so they don't accumulate
    zero_gradients(w, b)

    # save loss for plotting
    losses.append(loss.item())
```

Multiple outputs

Multiple outputs with neurons

Sometimes you want to predict more than a scalar value:

- Multi-class classification $y_i \in \{1, \ldots, K\}$.
- Multiple regression $\mathbf{y}_i \in \mathrm{R}^K$.

The simplest way to handle multiple outputs is to train K independent models.

- **Regression**: train K independent models for each of the K-dimensions of the output variable $\mathbf{y}_i$
- **Classification**: encode y_i using a one-hot encoding $\mathbf{y}_i \in \{0, 1\}^K$. Now train K classifiers using for each of the K-dimensions of the output variable $\mathbf{y}_i$. This is called one-vs-rest (or OVR) classification.

OVR classification

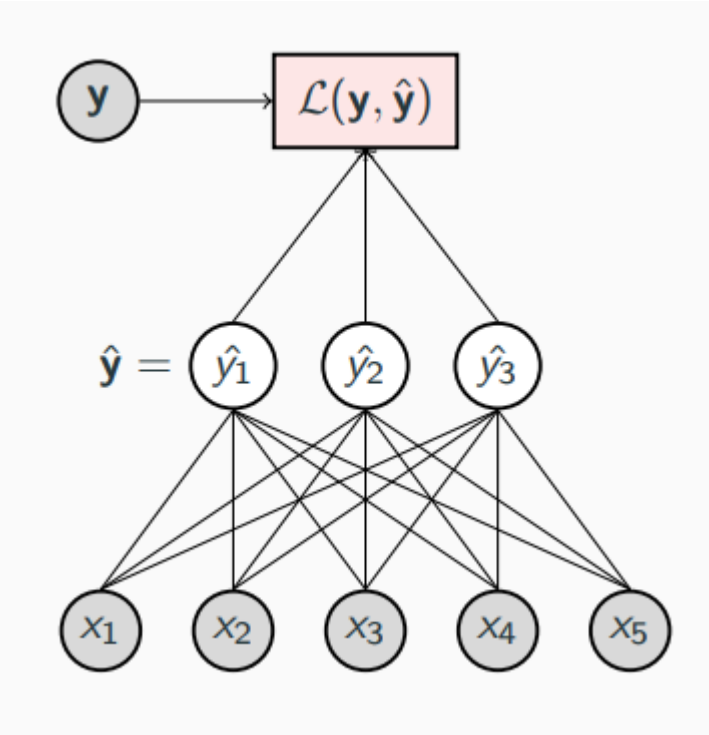
3 class classifier

3× num parameters

E.g. $\mathbf{y}_i = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$

Loss: $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{j=1}^{3} \mathcal{L}(y_j, \hat{y}_j)$

Regularization applied to all parameters.



Open world assumption

Note that in this setup not only allows for multi-class classification, but also **multi-label classification**. Each training example can have zero, one, or more labels.

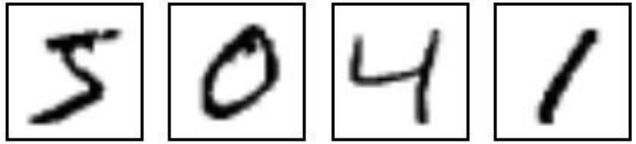
$$\text{E.g. } \mathbf{y}_i = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \text{ or } \mathbf{y}_i = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \text{ or } \mathbf{y}_i = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

Classifiers like this use the **open world** assumption. The probability of each target class is **independent** of the others. The prediction may give high probability (or score) for one, several, or none of the classes.

Closed world assumption

In some cases, you want the prediction to be **one and only one** of K classes. The target classes are **mutually exclusive**.

Example: handwritten digit classification
Predict digit $y \in \{0, \ldots 9\}$ from image pixels.



The target is one and only one digit (never multiple, and never none).

Multi-layer perceptrons

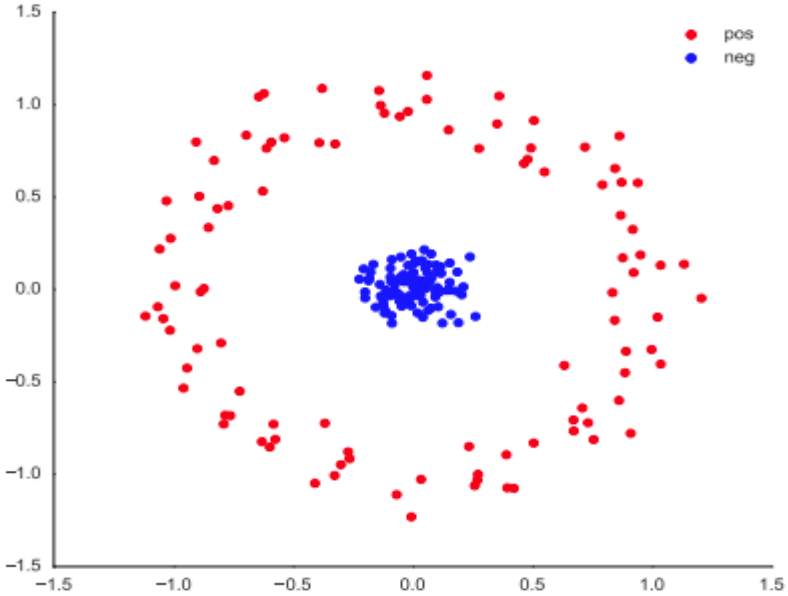
Motivation

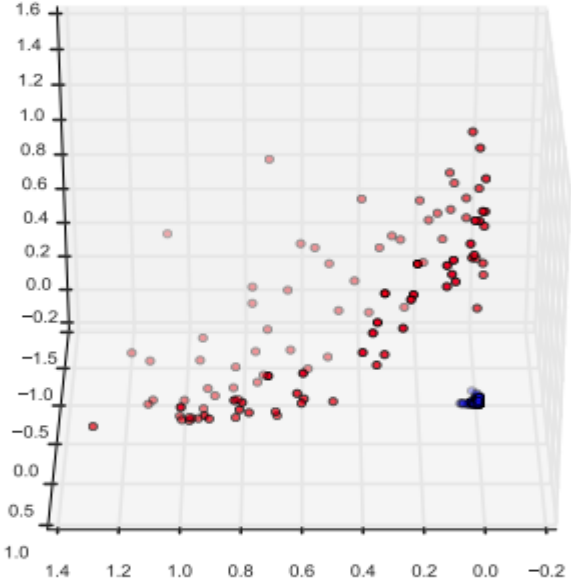
All of the single layer neural network we've seen so far are only able to produce **linear** decision boundaries.

Many interesting problems are not linearly separable.

We could design a feature mapping function $\varphi(\mathbf{x}) : \mathbb{R}^{D_1} \rightarrow \mathbb{R}^{D_2}$ that maps features to a higher-dimensional space where data is more likely to be linearly separable, and then apply our linear classifiers to the new features.

Mapping functions





$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^2 & \sqrt{2}x_1x_2 & x_2^2 \end{bmatrix}^T$$

Feature engineering

Designing such feature functions by hand is called **feature engineering**.

Many such feature maps have been proposed:

- Computer vision: SIFT, HOG, LBP, etc.
- Speech recognition: MFCC.
- Text: N-grams, LSI and LDA features (unsupervised learning).

Designing effective features is often **very difficult**!

It would be nice if we could learn the features from the data...

Adding a hidden layer

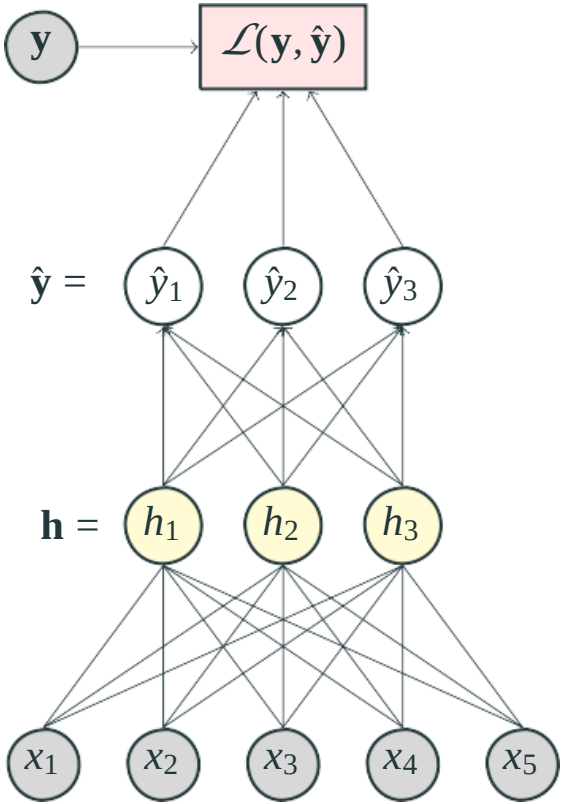
A **multi-layer perceptron** (MLP)

Input layer: $\{x_1, \ldots, x_D\}$

Hidden layer: $\{h_1, \ldots, h_M\}$

Output layer: $\{\hat{y}_1, \ldots, \hat{y}_K\}$

Loss: $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$



Adding a hidden layer

Now have **two** sets of weights and biases:

- W_1 and $\mathbf{b}_1$: input to hidden,
- W_2 and $\mathbf{b}_2$: hidden to output.

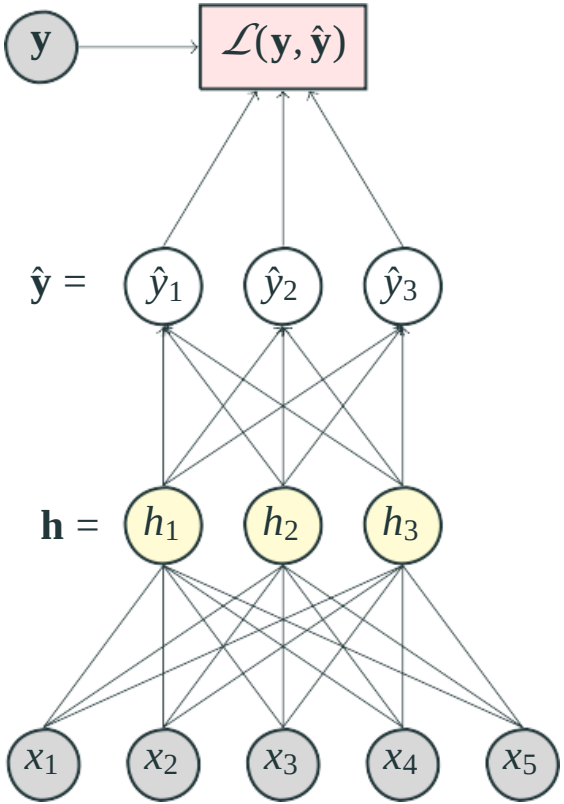
Input to hidden:

$$\mathbf{h} = g_1(W_1\mathbf{x} + \mathbf{b}_1)$$

Hidden to output:

$$\hat{\mathbf{y}} = g_2(W_2\mathbf{h} + \mathbf{b}_2)$$

g_1 and g_2 are **activation functions**. E.g. sigmoid.

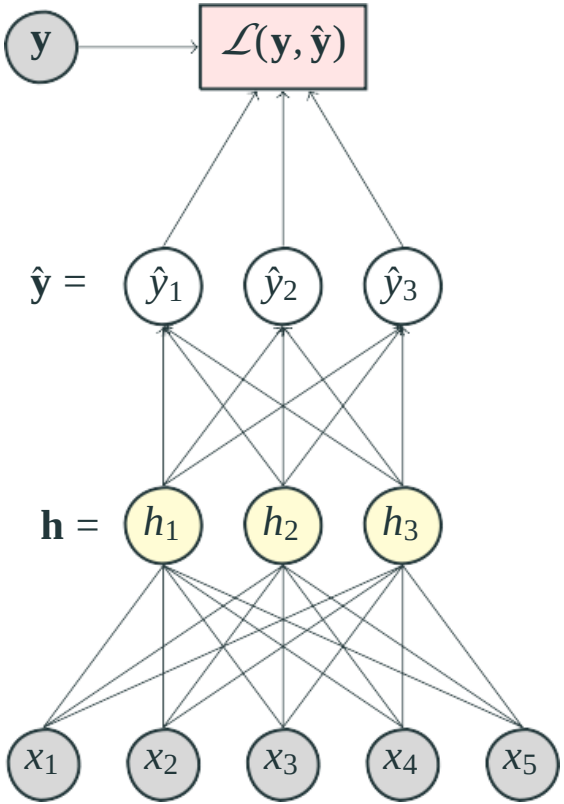


Adding a hidden layer

Decision function:

$$\begin{aligned}\hat{\mathbf{y}} &= f(\mathbf{x}) \\ &= g_2(W_2\mathbf{h} + \mathbf{b}_2) \\ &= g_2(W_2 g_1(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)\end{aligned}$$

If g_1 is a non-linear activation function (like a sigmoid), then $f(\mathbf{x})$ is a non-linear function of $\mathbf{x}$!

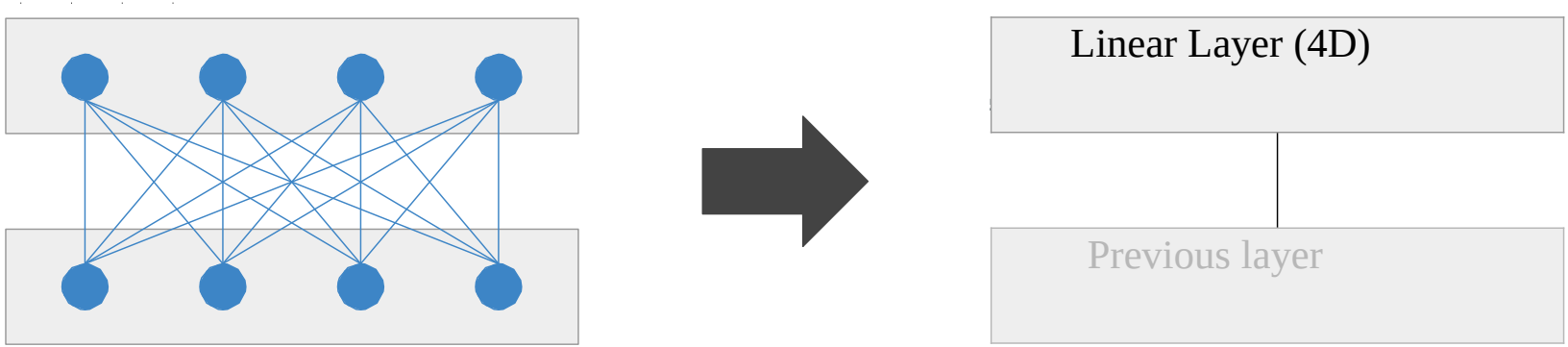


The layer abstraction

Instead of thinking about individual neurons, create an abstraction of a whole layer of neurons.

E.g. a linear layer. Each output is a linear combination of its inputs (plus a bias). The linear layer corresponds to a matrix multiplication by the weights:

$$\mathbf{x}_{t+1} = W\,\mathbf{x}_t + \mathbf{b}$$



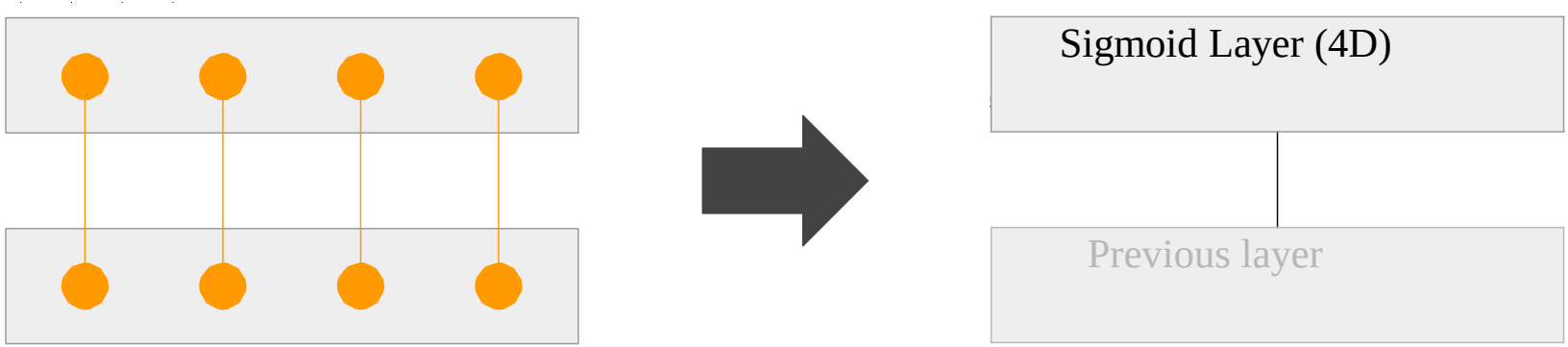
The layer abstraction: sigmoid layer

Can do the same for the non-linearities. E.g. the sigmoid layer does an elementwize sigmoid operation.

$$\mathbf{x}_{t+1} = g(\mathbf{x}_t)$$

with

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



The layer abstraction: ReLU layer

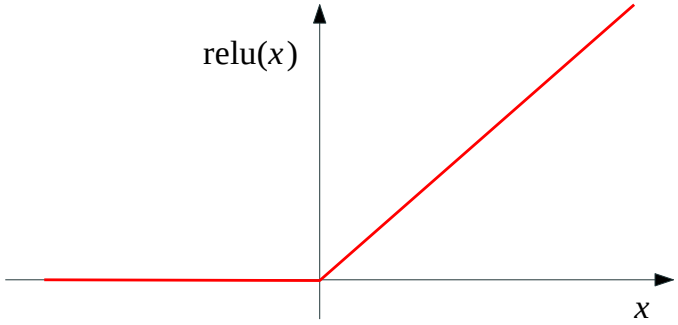
Another important non-linearity used in neural networks is the **rectified linear unit (ReLU)**:

$$\mathbf{x}_{t+1} = \mathrm{relu}(\mathbf{x}_t)$$

with

$$\mathrm{relu}(x) = \max\{0, x\}$$

again applied elementwise.



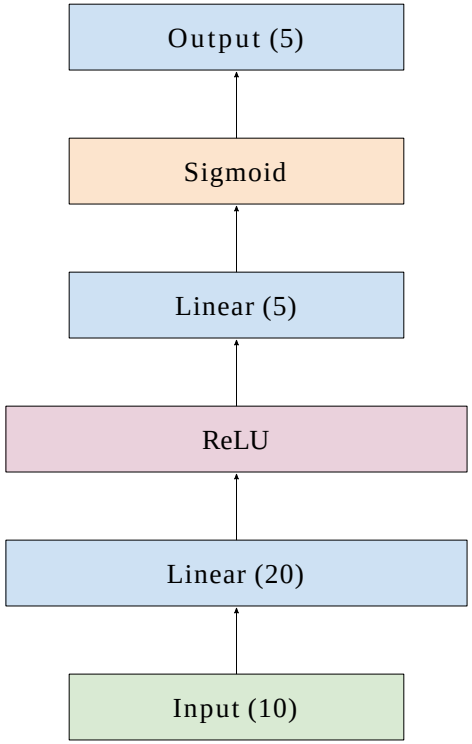
Composing layers into networks

Once we have defined these abstractions, we can compose these *bricks* together into more complex networks.

But how do we compute the gradients wrt the parameters in such networks?

We can do the same as before: backpropagate error from the output (loss) back through the layers and calculate the gradient wrt the parameters as we go.

Each layer in our network needs to also know how to propagate error backward.

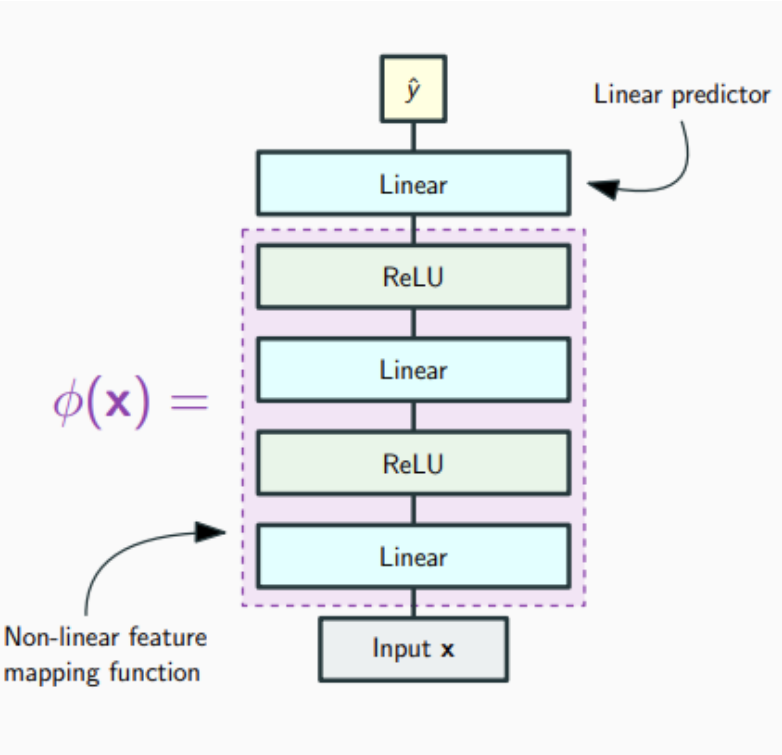


Representation learning

Hidden layers as feature extractors

The output of the hidden layers can be thought of a trainable non-linear feature mapping function $\varphi(\mathbf{x}) = g(W\mathbf{x} + \mathbf{b})$.

In this sense, an MLP **learns a representation** of the data that is more linearly separable than the original representation.



Deep learning

You can stack multiple hidden layers to compute successively more **abstract layers of representation**.

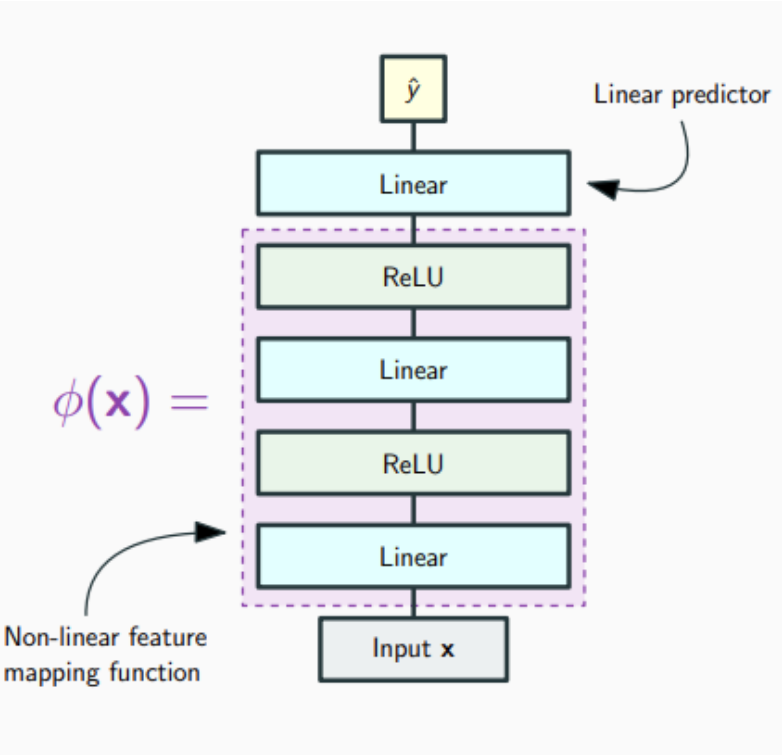
This is the idea behind **deep learning**.

The number of parameters increases when you

- increase number of hidden units per layer,
- increase number of hidden layers.

Deep networks are prone to **overfit** without:

- Strong regularization, and
- Lots of training data.



Further reading

- Hastie et al. The elements of statistical learning, Chapter 11. Section 11.3 onwards
  - **Note**: material in this chapter is a little out of date
  - Older editions of the book say that backprop is not usually the method of choice for optimizing deep networks, and that conjugate gradient is more common. No longer true, SGD and variants are most commonly used methods now.
  - Book uses sigmoid non-linearities in hidden layers: sigmoids have very slow convergence.
- Michael Nielsen. Neural networks and deep learning. Chapter 2
  - http://neuralnetworksanddeeplearning.com/index.html
  - Note: Non-modular approach to explaining backprop.

Resources

- Nando de Frietas on modular backpropagation https://www.youtube.com/watch?v=-YRB0eFxeQA