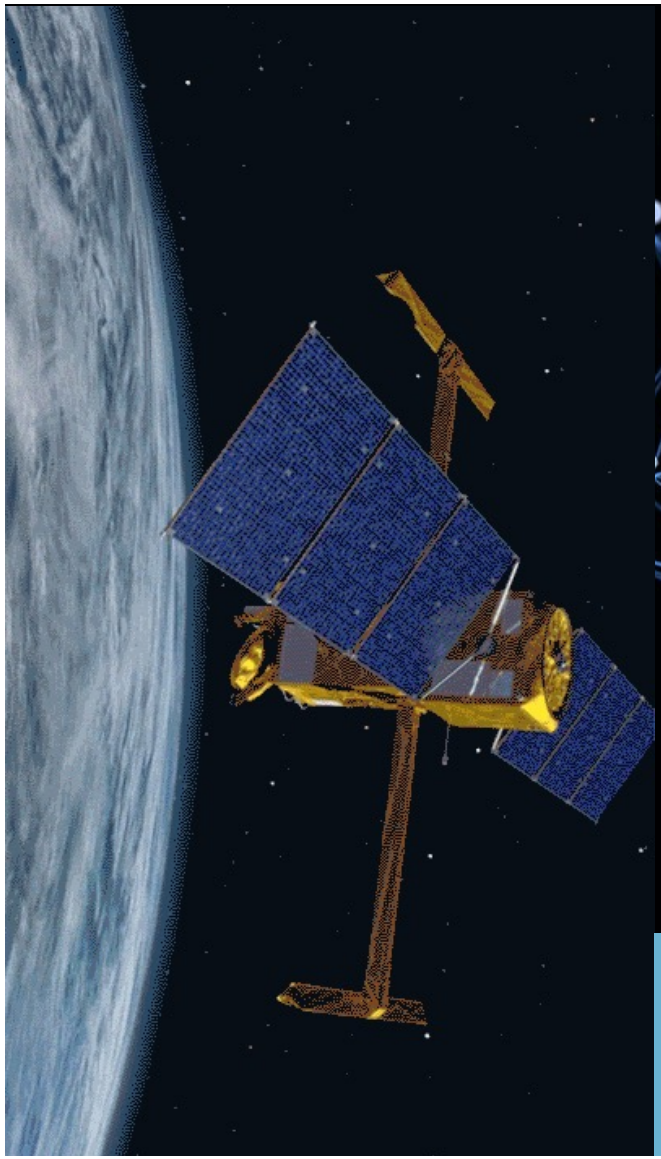
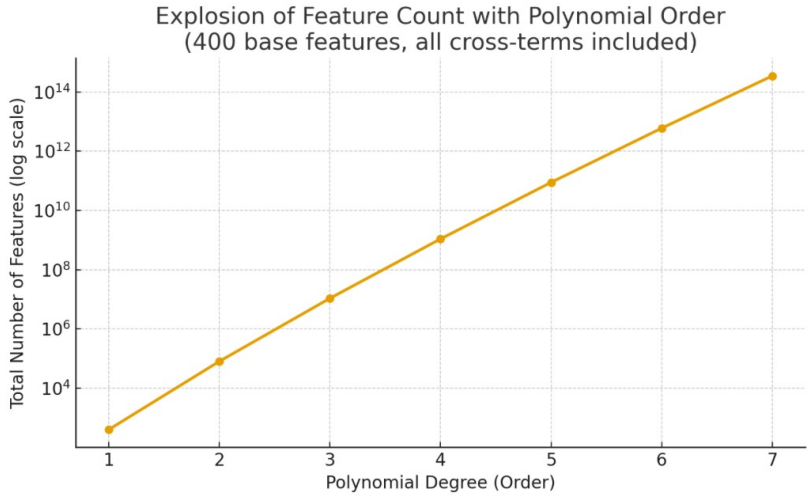


Lecture 10
Artificial Neural Networks

Why yet another algorithm?

- **Linear** numerical and logistic regression often insufficiently complex
- **Polynomials** become too expensive, especially due to cross-terms

Degree	Number of Features
1	401
2	80601
3	10827401
4	1093567501
5	88578967581
6	5993843472981

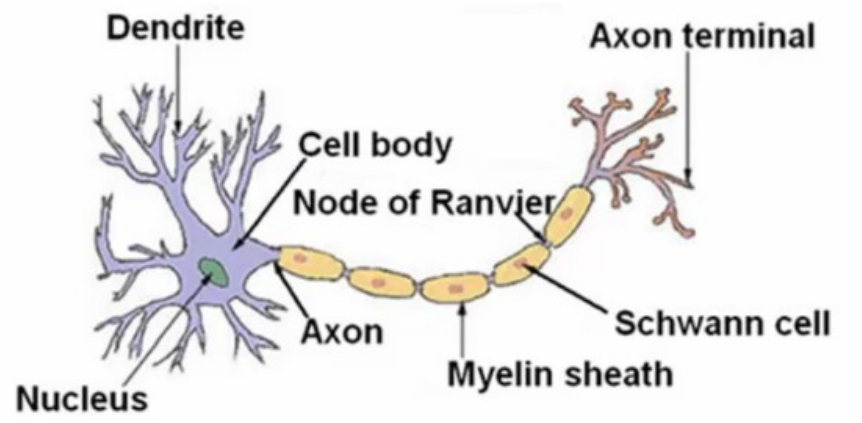


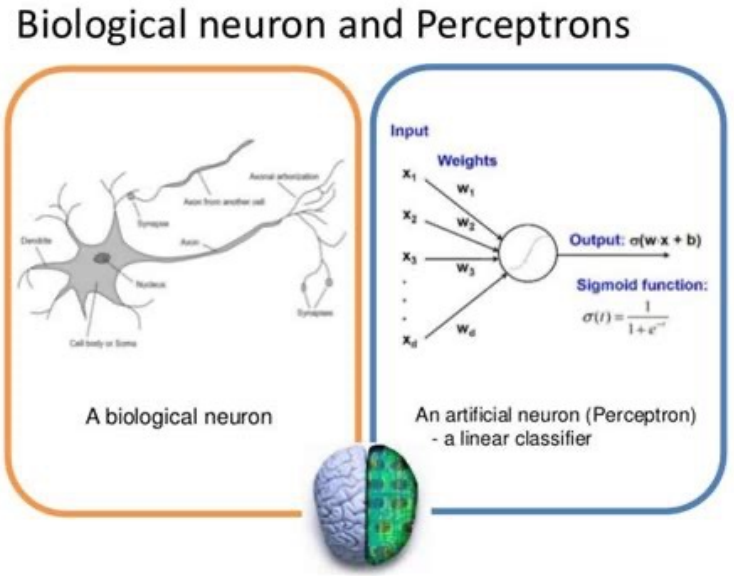
- Need more *efficient way to introduce high degree of non-linearity*: **neural networks**

Model Representation


Inspired by the human brain

Perceptron or Logistic Unit

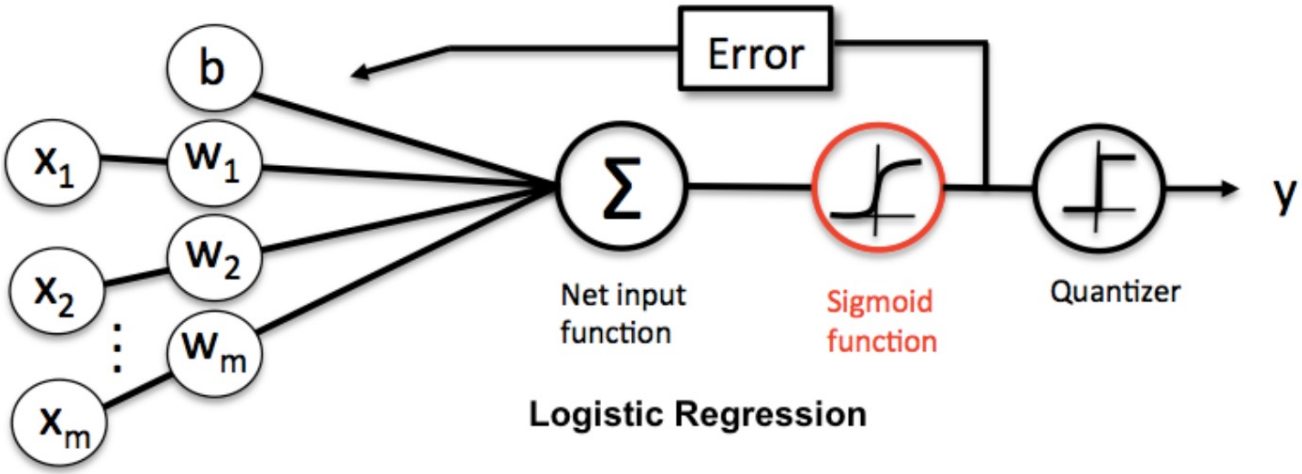




Feature	Biological Neuron	Perceptron / Artificial Node
Inputs / connections	Dendrites receive signals from many other neurons	**Inputs** (x_i), each multiplied by a **weight** (θ_i)
Summation / integration	Cell body (soma) integrates inputs i.t.o. electrical potential.	**Weighted sum** ($\sum_i x_i\,\theta_i + \theta_0$) Called **logit**
Threshold / activation	If input exceeds threshold, neuron 'fires'	**Activation function** (step, sigmoid, ReLU, …)
Output / propagation	Axon carries signal to other neurons	Node's output sent to subsequent nodes
Layers / network	Many neurons interconnected in networks	Layers of perceptrons / nodes

Logistic Regression as Single Neuron/Perceptron

- Binary logistic regression:



- 1 input dendrite per feature, 1 sigmoid soma, 1 output axion.

Subtlety. One-Hot-Encoding of Target

- Consider 3 labels 0, 1, 2 and 4 measurement samples
- Instead of target vector $y = [2, 0, 1, 2]$ we define one-hot-encoded:

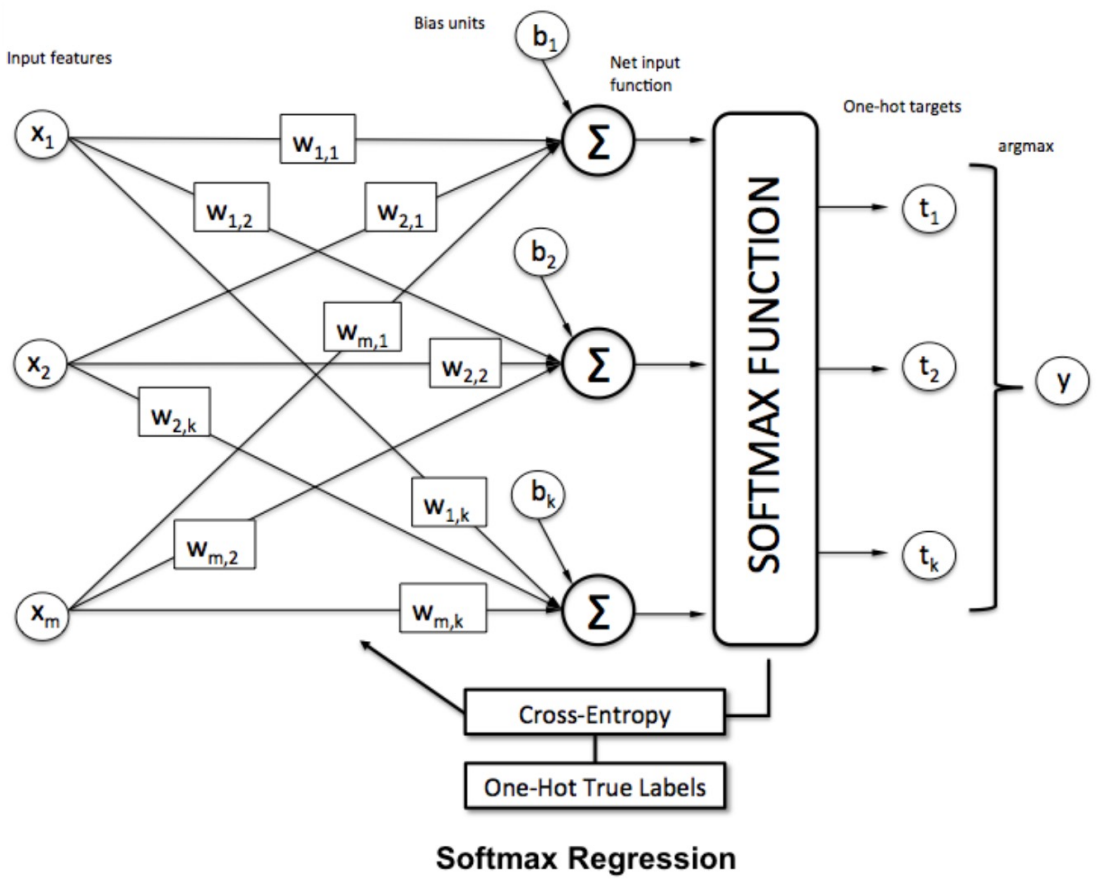
- $y = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

- Each row is sample
- Each column for a class
- y_k denotes column/class k

```
1   # Example data
2   ys = np.array([2, 0, 1, 2])
3   m = len(ys)
4   numlabels = 3
5
6   # One-hot encoding
7   Y = np.zeros((m, numlabels))
8   Y[np.arange(m), ys] = 1
9
10  print("Labels (y):", ys)
11  print("One-hot encoded (Y):\n", Y)
```

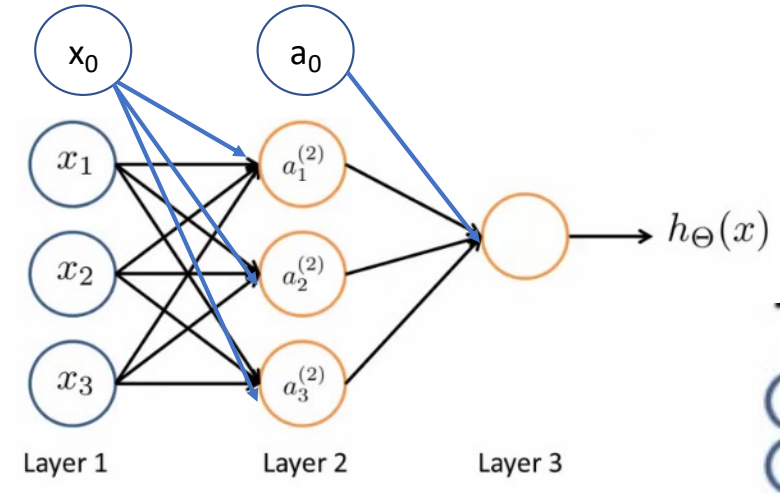
Logistic Regression as Single Neuron/Perceptron

- Multi-class with sigmoid, or **softmax**
- For $k = 1, \ldots, K$ classes
- $z_k = \sum_k \sum_j \theta_{j,k}\, x_j$
- Sigmoid: $\mathrm{y_k} = g(z_k)$ or
- Softmax: $\mathrm{y_k} = \frac{\exp(z_k)}{\sum_i \exp(z_i)}$ are true probabilities
- *K* output axions

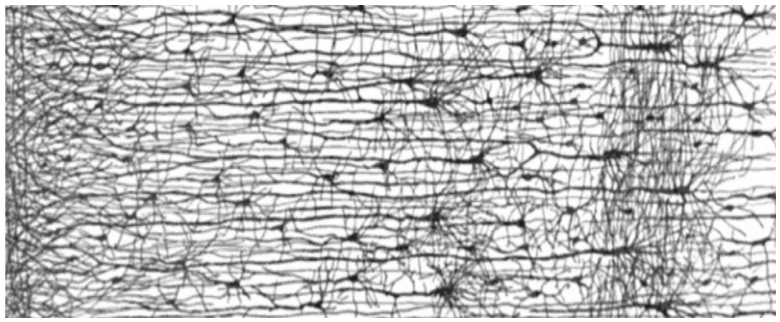


Artificial Neural Network

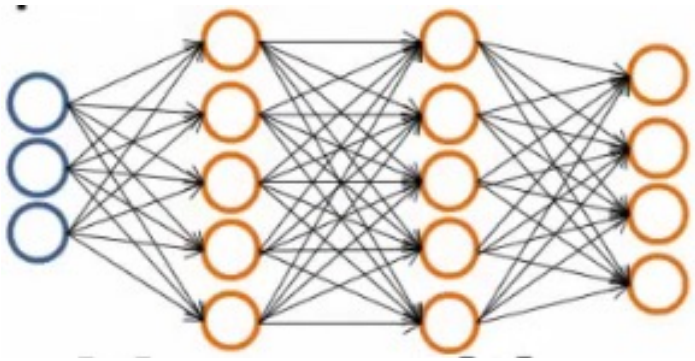
- Combine multiple logistic units:



Binary classification: 0 or 1

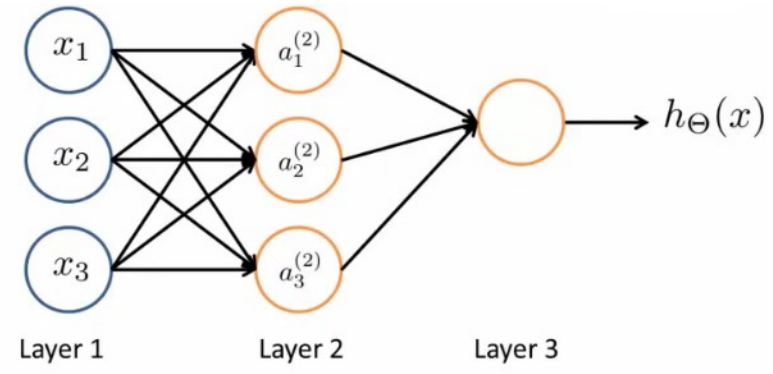


Multiclass: 0, 1, 2, 4, etc (probability)



$h_\Theta(x) \in \mathbb{R}^4$

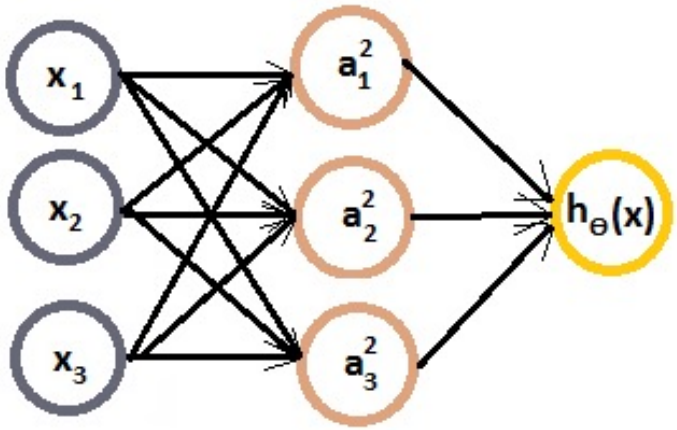
Nomenclature



- Input, output, and ***hidden*** layers
- Shallow or deep ANN
- Each node applies ***activation function*** $a_i^{(l)}$ (e.g., sigmoid)
- Superscript *l* denotes layer nr, subscript *i* for each node in that layer
- One node for each feature in input layer with unit activation: $a_i^{(1)} = x_i$
- One node for each class in output layer: $a_i^{(3)}$

Algebra of Binary ANN Classification

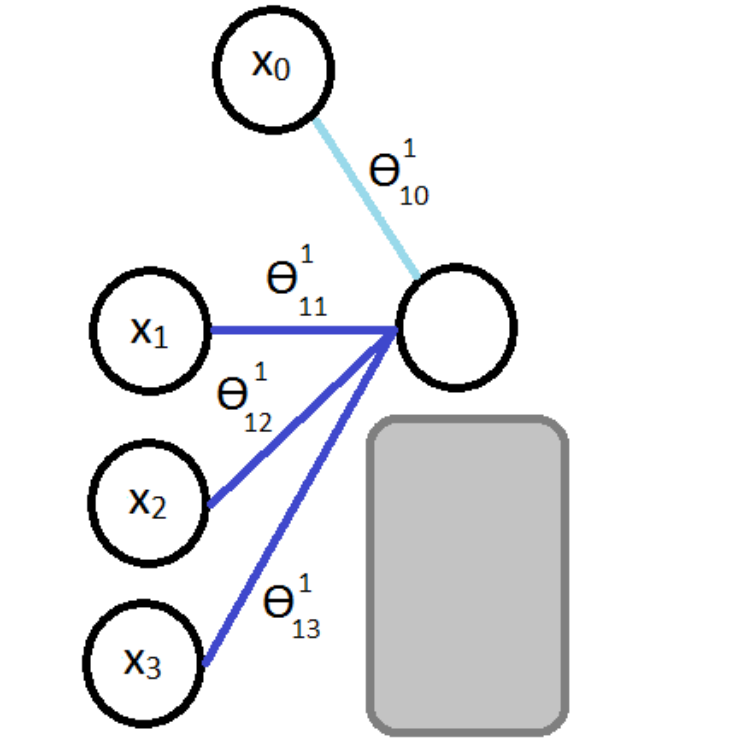
(bias not shown)



- Fitting parameters, or **weights**, are now *arrays*: $\Theta_{ji}^{(l)}$ where *i* is node in layer *l* and *j* is node in layer *l*+1
- In this example: $\boldsymbol{\Theta}^{(1)}$ is 3×4 and $\boldsymbol{\Theta}^{(2)}$ is 1×4 vector, for binary output

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

Algebra of Binary ANN Classification



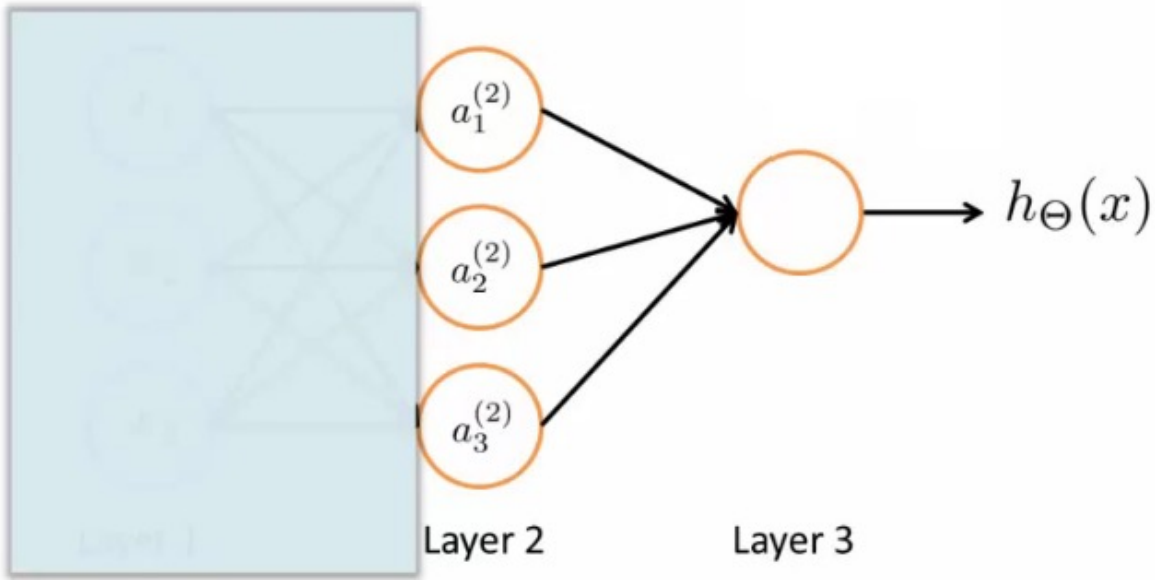
$g(\Theta_{10}^{1}x_0 + \Theta_{11}^{1}x_1 + \Theta_{12}^{1}x_2 + \Theta_{13}^{1}x_3)$

- Fitting parameters, or **weights**, are now *arrays*: $\Theta_{ji}^{(l)}$ where *i* is node in layer *l* and *j* is node in layer *l*+1
- In this example: $\boldsymbol{\Theta}^{(1)}$ is 3×4 and $\boldsymbol{\Theta}^{(2)}$ is 1×4 vector, for binary output

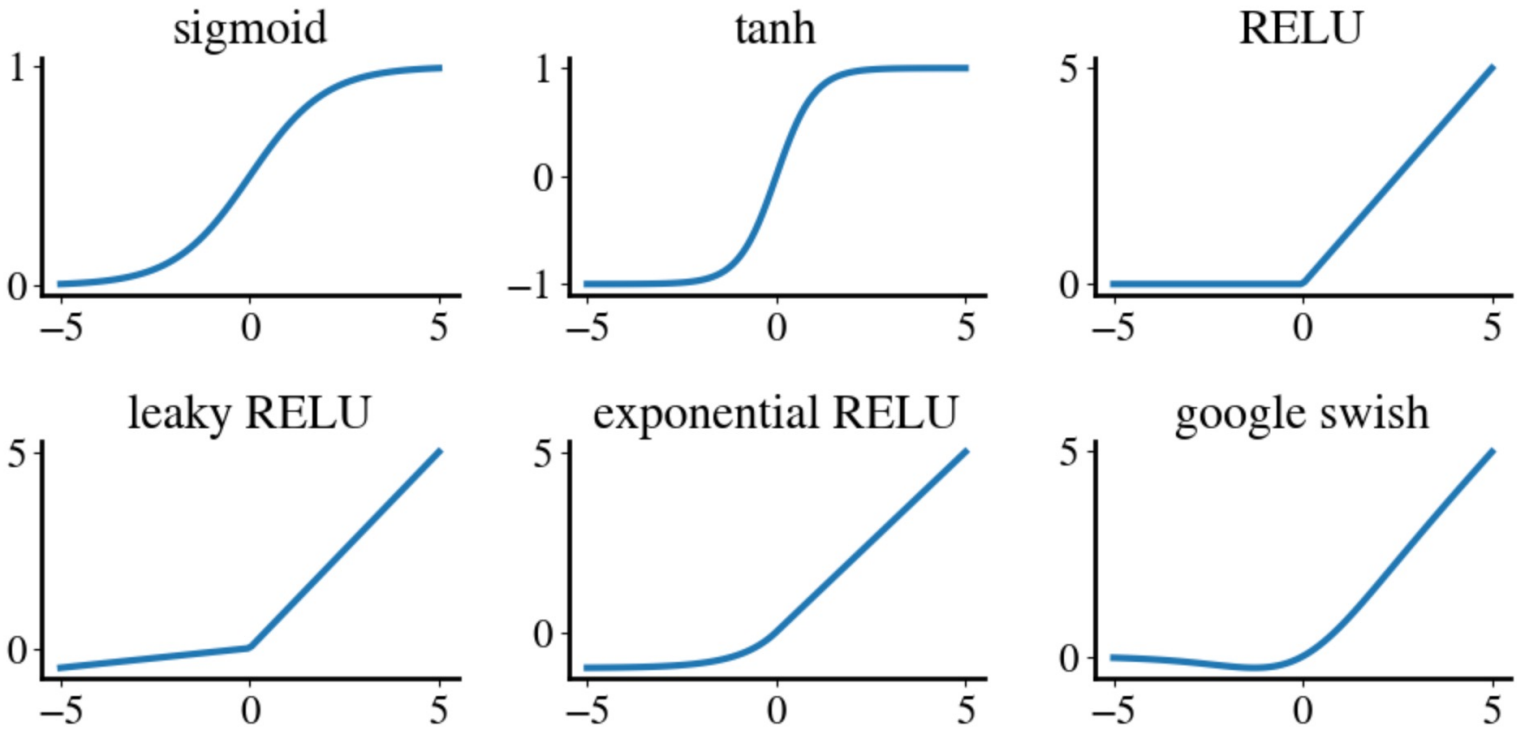
- Also, $\boldsymbol{z}^{(2)} = \boldsymbol{\Theta}^{(1)} \cdot \boldsymbol{x} = \boldsymbol{\Theta}^{(1)} \cdot \boldsymbol{a}^{(1)}$
- So $\boldsymbol{a}^{(2)} = g(\boldsymbol{z}^{(2)})$

Intuition

- Note: last step is identical to logistic regression, but i.t.o. $\boldsymbol{a}^{(2)}$ 'features'
- First step creates many new non-linear features
- Model automatically learns which non-linear features are useful



Different activation functions for hidden layers

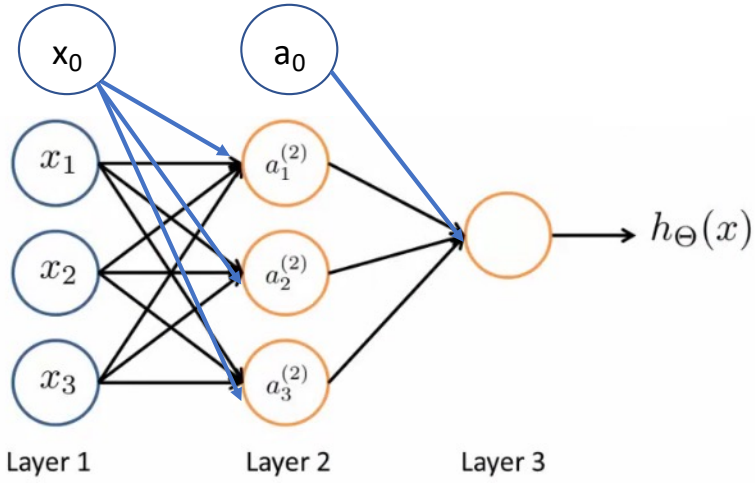


Forward Propagation

Forward propagation

How to make **binary** ANN prediction for a single measurement?

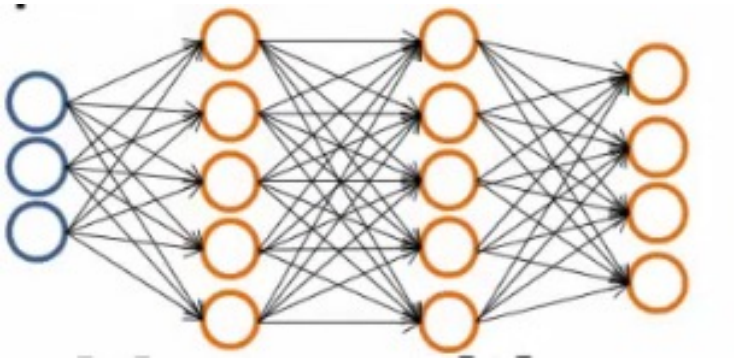
- Define feature array ***x*** as always and add bias. For 1 measurement, 3 features, this is a 4-element vector (one row of full feature array) $a^{(1)}$.
- For 3×4 array of weights $\boldsymbol{\Theta}^{(1)}$, compute $\boldsymbol{z}^{(2)} = \boldsymbol{\Theta}^{(1)} \cdot \boldsymbol{a}^{(1)}$ for 3 nodes.
- Apply 3 (sigmoid) activations $\boldsymbol{a}^{(2)} = g(\boldsymbol{z}^{(2)})$
- Add bias to $\boldsymbol{a}^{(2)}$ (now 4 values)
- $\boldsymbol{z}^{(3)} = \boldsymbol{\Theta}^{(2)} \cdot \boldsymbol{a}^{(2)}$ (with 4 weights $\Theta^{(2)}$)
- $\boldsymbol{a}^{(3)} = g(\boldsymbol{z}^{(3)})$ is final prediction/hypothesis
- $h_\Theta(x) \sim g(\boldsymbol{\Theta}^2 \cdot g(\boldsymbol{\Theta}^{(1)} \cdot x))$



Forward propagation

How to make **multiclass** (4) ANN prediction for a single measurement?

- Define feature array ***x*** as always and add bias. For 1 measurement, 3 features, this is a 4-element vector (one row of full feature array) $a^{(1)}$.
- For 3×4 array of weights $\boldsymbol{\Theta}^{(1)}$, compute $\boldsymbol{z}^{(2)} = \boldsymbol{\Theta}^{(1)} \cdot \boldsymbol{a}^{(1)}$ for 3 nodes.
- Apply 3 (sigmoid) activations $\boldsymbol{a}^{(2)} = g(\boldsymbol{z}^{(2)})$
- Add bias to $\boldsymbol{a}^{(2)}$ (now 4 values)
- $\boldsymbol{z}^{(3)} = \boldsymbol{\Theta}^{(2)} \cdot \boldsymbol{a}^{(2)}$ **(with 4×4 weights $\Theta^{(2)}$)**
- $\boldsymbol{a}^{(3)} = g(\boldsymbol{z}^{(3)})$ is final prediction/hypothesis
- $0 < \boldsymbol{a}^{(3)} < 1$ has 4 values/probabilities



Forward Propagation - Code

```
4   def nn_forward(theta, input_layer_size, hidden_layer_size, num_labels, X):
5       """Forward pass through a 2-layer neural network.
6          Shapes chosen so that no .T transposes are needed."""
7       m, n = X.shape
8
9       # Reshape Thetas for left-to-right matrix flow
10      Theta1_dims = (input_layer_size + 1, hidden_layer_size)
11      Theta1_len  = np.prod(Theta1_dims)
12      Theta2_dims = (hidden_layer_size + 1, num_labels)
13
14      Theta1 = theta[:Theta1_len].reshape(Theta1_dims)
15      Theta2 = theta[Theta1_len:].reshape(Theta2_dims)
16
17      # Forward pass
18      X1 = np.c_[np.ones((m, 1)), X]
19      Z2 = X1 @ Theta1
20      A2 = sigmoid(Z2)
21      A2 = np.c_[np.ones((m, 1)), A2]
22      Z3 = A2 @ Theta2
23      H3 = sigmoid(Z3)
24
25      return H3, Theta1, Theta2
```

Note: hypothesis / output is probability for *each* class!

Probabilities -> Integer Class Labels

```python
def nn_predict(theta, input_layer_size, hidden_layer_size, num_labels, X):
    """
    Predict labels for input data X using trained network parameters.
    Uses nn_forward() for the forward pass.
    """
    # Forward pass (reuse your function)
    H3, _, _ = nn_forward(theta, input_layer_size, hidden_layer_size, num_labels, X)

    # Pick the class with highest activation for each example
    pred = np.argmax(H3, axis=1)
    return pred

```

Pick class with highest probability

ANN Algorithm

Same approach as always

- Objective: find fitting parameters / weights $\boldsymbol{\theta} = [\boldsymbol{\Theta}^{(1)}[:]; \boldsymbol{\Theta}^{(2)}[:]]$
- Need ‘error’ or **cost function** to compare ANN prediction to truth
- Need ***gradient* of cost function** for gradient descent, accelerated methods

Cost function for ANN

- Remember logistic cost function (with Ridge regularization):

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

- Cost function (**cross-entropy**) for ANN is simply:

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

- With extra sum over different classes:

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

Cross-Entropy Cost + Ridge Regularization

```
4   def nn_cost(theta, input_layer_size, hidden_layer_size, num_labels, X, y, lambda_r):
5       """Compute the cost J(θ) for a 2-layer neural network."""
6       m = X.shape[0]
7       H3, Theta1, Theta2 = nn_forward(theta, input_layer_size, hidden_layer_size, num_labels, X)
8
9       # One-hot encode labels
10      Y = np.zeros((m, num_labels))
11      Y[np.arange(m), y] = 1
12
13      # --- Core cost (cross-entropy) ---
14      eps = 1e-12
15      J = -np.sum(Y * np.log(H3 + eps) + (1 - Y) * np.log(1 - H3 + eps)) / m
16
17      # --- Regularization (ignore bias columns) ---
18      # Strip the first column (bias weights)
19      reg = (np.sum(Theta1[1:, :]**2) + np.sum(Theta2[1:, :]**2)) * lambda_r / (2*m)
20      J += reg
21
22      return J
23
```

Backward Propagation

Gradients of ANN cost function

- Remember for logistic regression: $\frac{\partial J(\theta)}{\partial \theta} = -\frac{1}{m}\sum_{i=1}^{m}(h(x_i;\theta) - y^i)\, x_i$
- But for ANN, the output is $h_\Theta(x_i) = g(\Theta^{(2)} \cdot a^2)$, and $a^2 = g(\Theta^{(1)} \cdot x_i)$

- To obtain $\frac{\partial J(\Theta^{(1)},\Theta^{(2)})}{\partial \Theta^{(1)}}$ and $\frac{\partial J(\Theta^{(1)},\Theta^{(2)})}{\partial \Theta^{(2)}}$ requires partial derivatives/chain rule
- Note that $\frac{\partial g(z)}{\partial z} = g(z)(1 - g(z))$

- Start from output layer, partial derivatives layer-by-layer to input layer
- **Backward propagation** or '**back prop**'

```python
def nn_gradient(theta, input_layer_size, hidden_layer_size, num_labels, X, y, lambda_r):
    m = X.shape[0]

    # Forward pass
    H3, Theta1, Theta2 = nn_forward(theta, input_layer_size, hidden_layer_size, num_labels, X)

    # Recompute intermediates needed for backprop
    X1 = np.c_[np.ones((m, 1)), X]                  # (m, input+1)
    Z2 = X1 @ Theta1                                # (m, hidden)
    A2_no_bias = sigmoid(Z2)                        # (m, hidden)
    A2 = np.c_[np.ones((m, 1)), A2_no_bias]         # (m, hidden+1)

    # One-hot encode labels
    Y = np.zeros((m, num_labels))
    Y[np.arange(m), y] = 1

    # Error in output layer
    d3 = H3 - Y                                     # (m, num_labels)

    # Error in hidden layer (excluding bias)
    d2 = (d3 @ Theta2[1:, :].T) * (A2_no_bias * (1 - A2_no_bias))  # (m, hidden)

    # Unregularized gradients
    Theta2_grad = (A2.T @ d3) / m                   # (hidden+1, num_labels)
    Theta1_grad = (X1.T @ d2) / m                   # (input+1, hidden)

    # Regularization (ignore bias row 0)
    Theta1_grad[1:, :] += (lambda_r / m) * Theta1[1:, :]
    Theta2_grad[1:, :] += (lambda_r / m) * Theta2[1:, :]

    # Unroll gradients
    grad = np.concatenate([Theta1_grad.ravel(), Theta2_grad.ravel()])
    return grad
```

Same terms as forward propagation
Needed to compute derivatives

Same one-hot-encode

Chain rule derivatives
of error contributions
of each layer

Derivative of Ridge term

Return one long vector of
all gradients

Final detail

- Unlike for logistic and numerical regression, we cannot start with initial guess of 0 for all $\boldsymbol{\theta} = [\boldsymbol{\Theta}^{(1)}[:];\ \boldsymbol{\Theta}^{(2)}[:]]$;
- If all initial guesses are the same, the values for all the nodes in each layer are the same (even after back-prop) and ANN will perform like regular logistic regression
- Solution: small random values as first guess

```
epsilon_init = 0.12

# New shape convention: (input+1, hidden)
Theta1 = np.random.rand(input_layer_size + 1, hidden_layer_size) * 2 * epsilon_init - epsilon_init

# (hidden+1, output)
Theta2 = np.random.rand(hidden_layer_size + 1, num_labels) * 2 * epsilon_init - epsilon_init

# Flatten and concatenate
initial_theta = np.concatenate([Theta1.ravel(order='F'), Theta2.ravel(order='F')])
```

Putting all together

How to train ANN?

- Same as every numerical & logistic regression before!
- Use any function to minimize cost function, e.g. conjugate gradient:

```
[15]  1  %%time
      2  activation_function_choice = 1
      3  all_theta_CG = optimize.fmin_cg(cost_nn, initial_theta,fprime=cost_gradient_nn, ...
      4                                  args=(input_layer_size,hidden_layer_size,num_labels,X,y,lambda_r),gtol=0.0001)
```

```
Optimization terminated successfully.
         Current function value: 0.002624
         Iterations: 158
         Function evaluations: 473
         Gradient evaluations: 473
CPU times: user 50.3 s, sys: 19.2 s, total: 1min 9s
Wall time: 40.3 s
```

Or our own Gradient Descent!

```
%%time
if do_gradient_descent:
  nrsteps = 100000

  alpha = 1
  lambda_r = 0
  errors = np.zeros(nrsteps)
  theta = initial_theta

  i=0
  converged = False
  diverged = False
  tolerance = 5e-4
  while i < nrsteps-1 and not converged and not diverged:

    errors[i] = cost_nn(theta,input_layer_size,hidden_layer_size,num_labels,X,y,lambda_r)
    theta -=  alpha * cost_gradient_nn(theta,input_layer_size,hidden_layer_size,num_labels,X,y,lambda_r)
    if i>100:
      improvement = np.abs(errors[i]/errors[i-1])
      if 1-improvement < tolerance:
        converged = True
        print(i, "Converged with error ", round(errors[i],2),improvement, 'thetas:', np.round(theta[:10],2), '...',np.round(theta[-10:],2))
      elif improvement > 1:
        print(i, improvement, "Errors are diverging:", errors[i], ">", errors[i-1])
        diverged = True
    i+=1
```

Conclusions

- ANN combine multiple logistic units to make non-linear predictions
- Input layer has one node per feature, plus bias
- Output layer has one node per class (no need for one-vs-all)
- Multiple ‘hidden’ layers can have any nr of nodes
- Each hidden node takes a different linear combination of outputs from previous layer and applies non-linear activation function
- Effectively, this creates/learns new complex features, and deletes others

Conclusions

- Predictions from **forward propagation**:
- $h_\Theta(x) = g(\Theta^4 \cdot g\left(\Theta^3 \cdot g\left(\Theta^2 \cdot g(\Theta^1 \cdot x)\right)\right))$
- Cost gradients from **backward propagation** (chain rule, partial derivatives)

- One cost function and gradient are defined, training ANN is identical to numerical and logistic regression
- Subtlety 1: all $\Theta^{[l]}$ need to be rolled into one huge vector
- Subtlety 2: fitting parameters/weights should not be initialized to all zeros