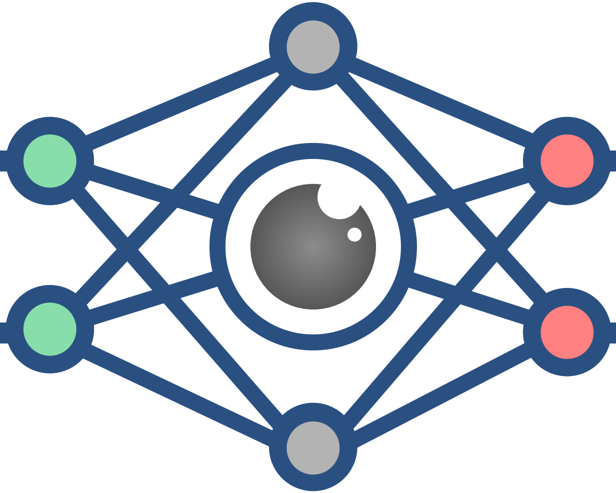
CS3485

Deep Learning for Computer Vision



Lec 3: The Multilayer Perceptron and Intro to Deep Learning

Announcements

- Lab 1:
  - Due on Thursday!
  - Let me know if you haven’t found any partner yet. For this specific lab, I’ll allow for individual submissions.
- Quiz 1:
  - This Thursday, make sure to check our schedule for next quizzes!

(Tentative) Lecture Roadmap

Basics of Deep Learning

- Intro to Computer Vision
- Linear Classifiers and Perceptron
- Multilayer Perceptron
- Optimization in Deep Learning
- Pytorch I – MLPs
- Pytorch II – Images and Regularization
- Convolutional Neural Networks
- Data Augmentation and Deep CNNs
- Transfer Learning and Residual Nets
- Inception Net and what CNNs learn
- Adversarial Examples and Self-supervision

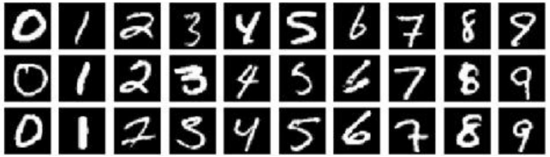
Deep Learning and Computer Vision in Practice

- Intro to MLOps
- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- The Attention Mechanism
- Transformers and ChatGPT
- Image Generation by Prompt
- Misc. Topics

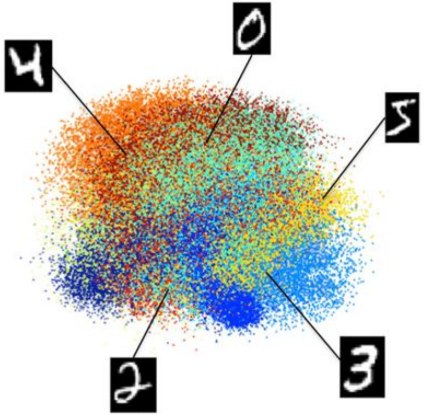
Limitations of the Perceptron Model

- Last time we saw that the Perceptron Model is useful to **model linear classifiers** and the Perceptron Algorithm is efficient at **learning the parameters** of the model.
- They, however, have two main limitations in terms of its applicability to most realistic datasets (like MNIST, on the right):
    a. The model can only perform **binary classification**, i.e., handle datasets with two classes,
    b. The algorithm expects the data to be **linearly separable**.
- Today we'll improve the Perceptron Model, so it can tackle both of the above issues.
- This new model, **the Multilayer Perceptron**, will form the basis for Deep Learning algorithms.

MNIST Dataset

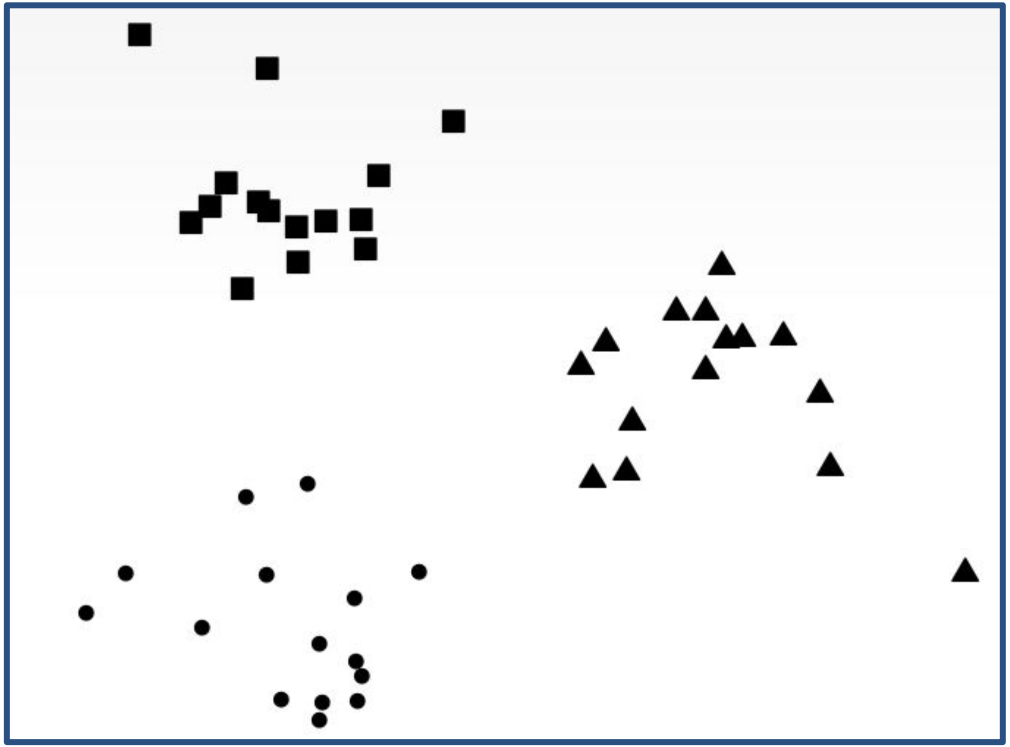


MNIST Dataset Projected in 2D



Multiclass Problems

- For the purpose of this course, assume that the Perceptron model our best solution* for **binary** classification problems.
- Now consider the problem of **classifying the points on the right into 3 classes**: squares, triangles and circles?
- How can we use only the Perceptron to do it?

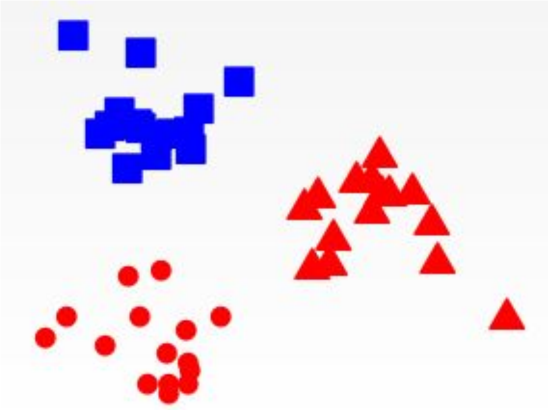


* In fact, there are many better algorithms for binary classification using linear classifiers than the perceptron, such as Support Vector Machines.

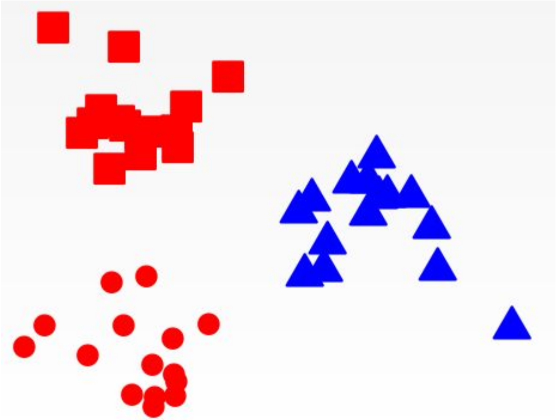
Multiclass Problems

- We can use three classifiers!

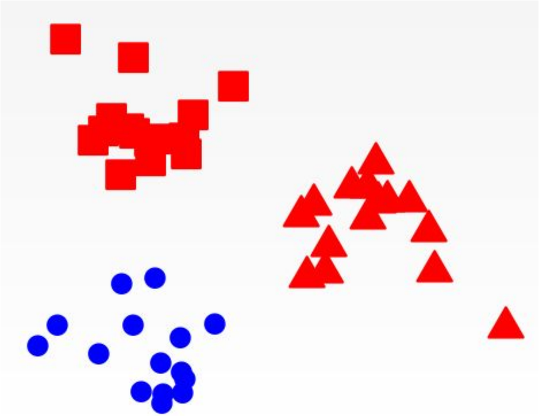
1. Is this a square?



2. Is this a triangle?



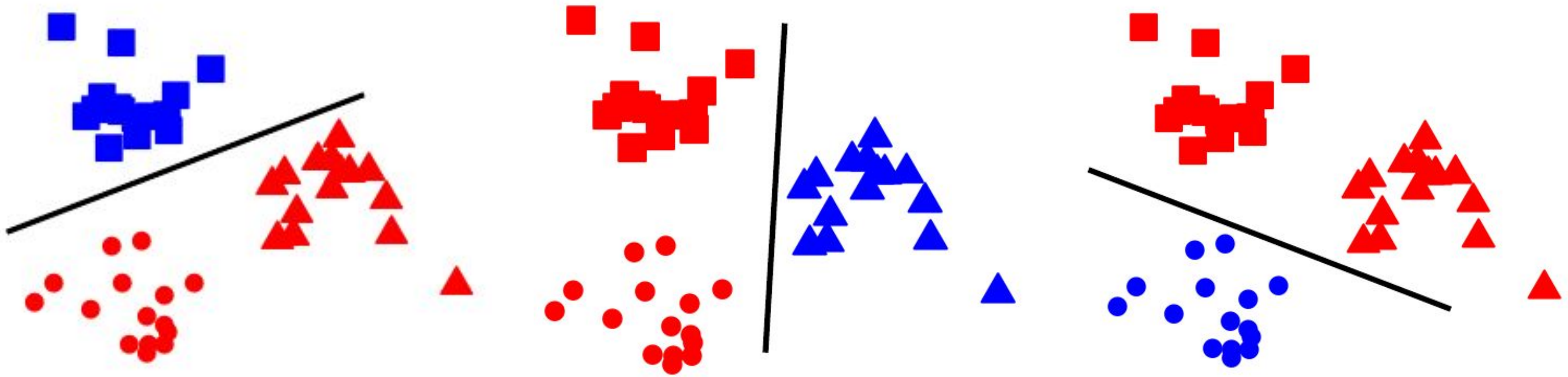
3. Is this a circle?



Multiclass Problems

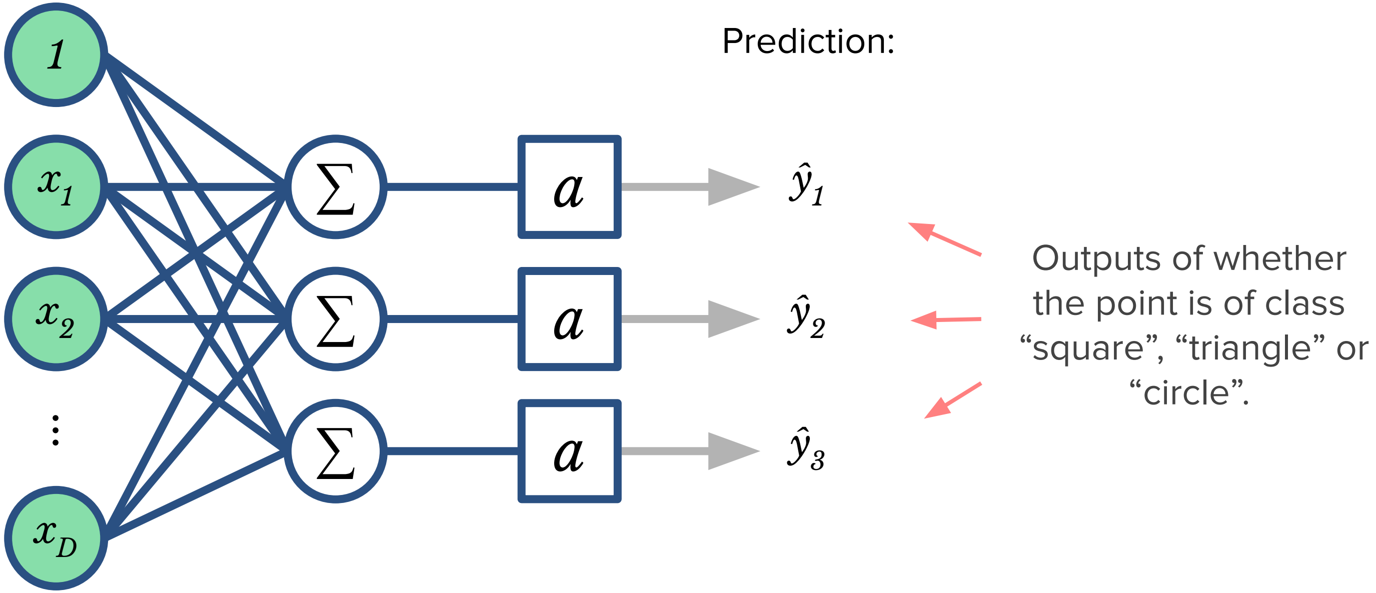
- We can use three classifiers!

1. **Is this a square?**
2. **Is this a triangle?**
3. **Is this a circle?**



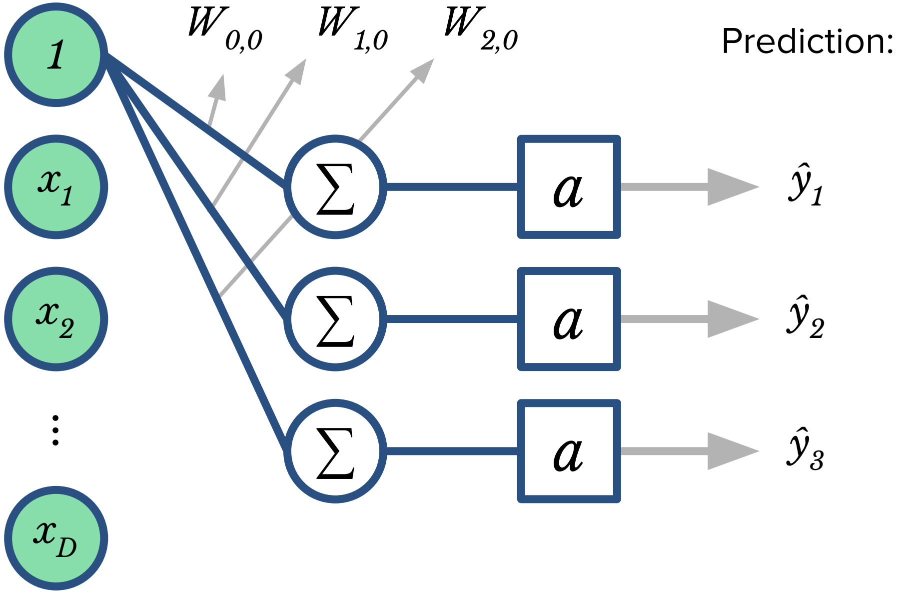
Multiclass perceptron

- We have a perceptron (or **neuron/unit**) for each class, so we can represent them as:



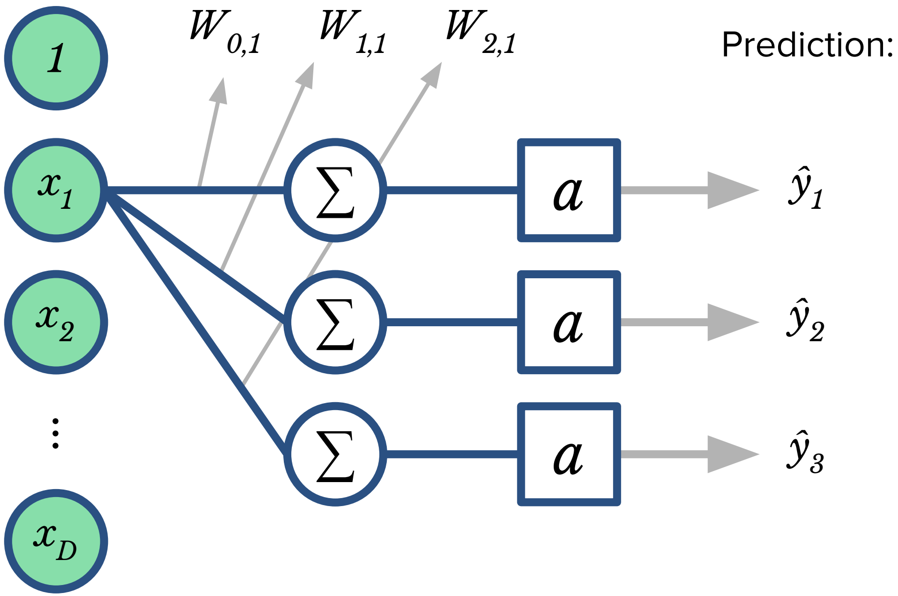
Multiclass perceptron

- Now each weight set of each neuron will be an entry of a $3 \times (D+1)$ **matrix of weights** W.



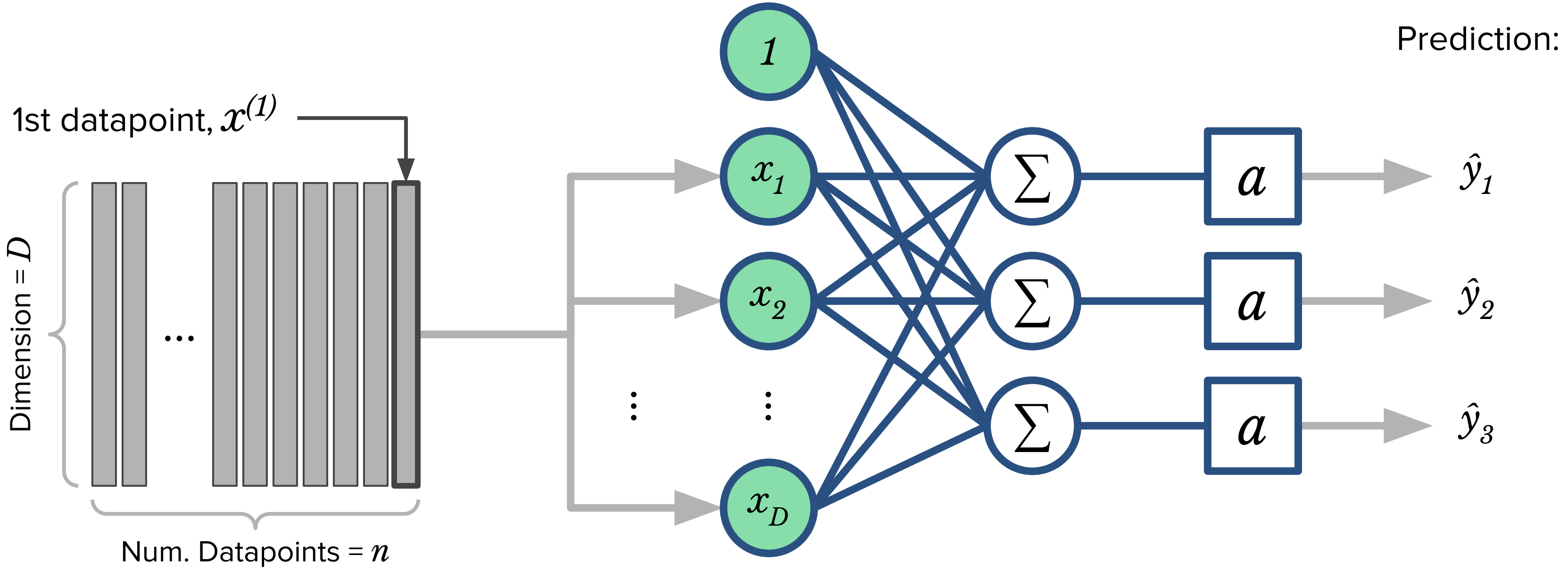
Multiclass perceptron

- Now each weight set of each neuron will be an entry of a $3 \times (D+1)$ **matrix of weights** W.



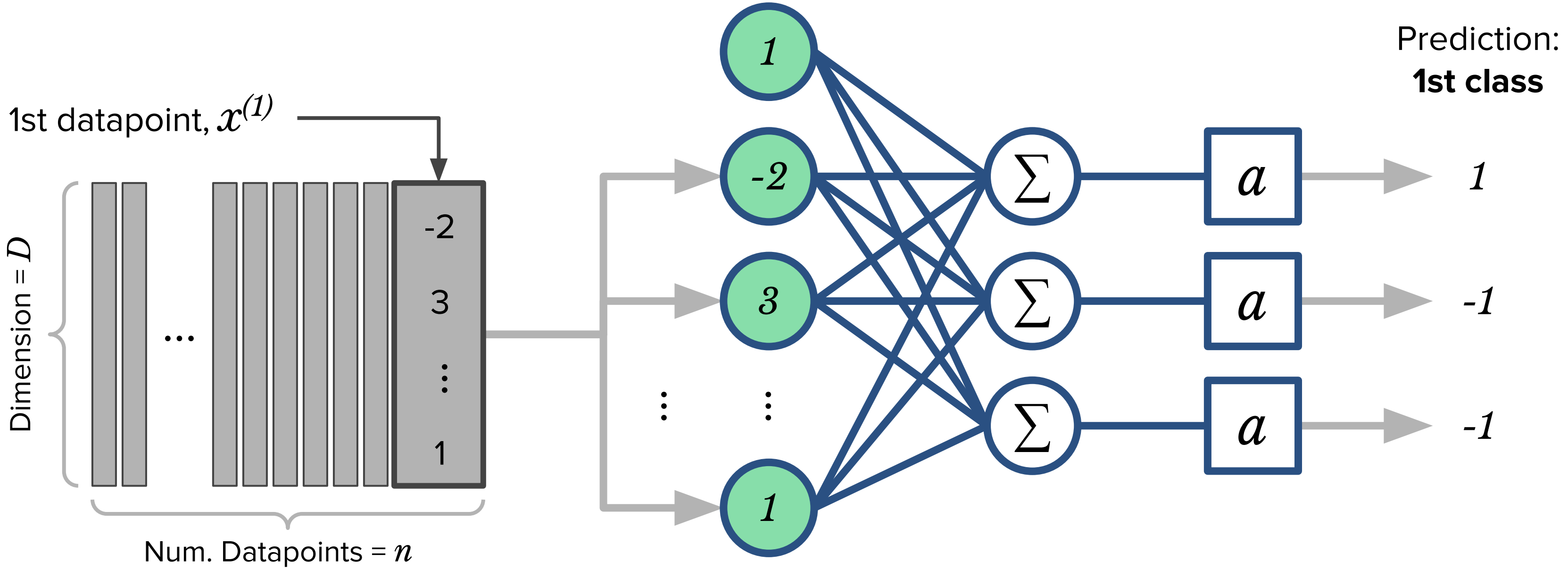
Multiclass perceptron

- Assuming we trained the three perceptrons, we can do a forward pass on them:



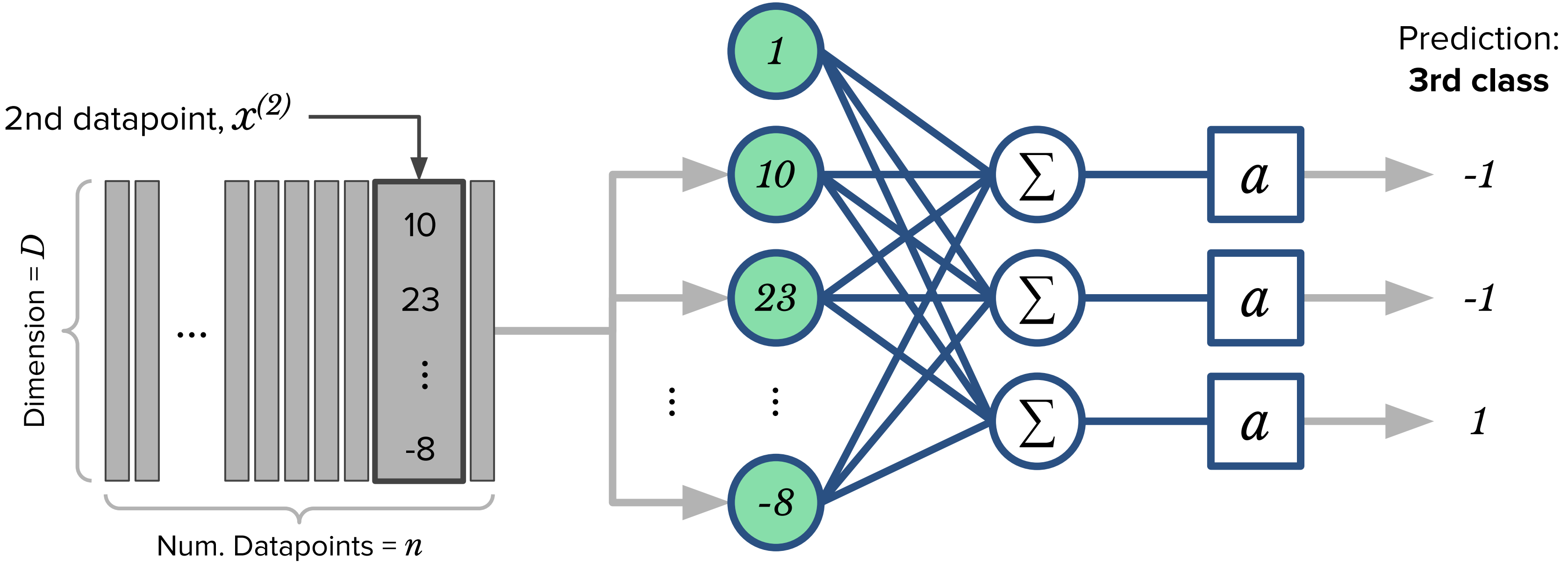
Multiclass perceptron

- Assuming we trained the three perceptrons, we can do a forward pass on them:



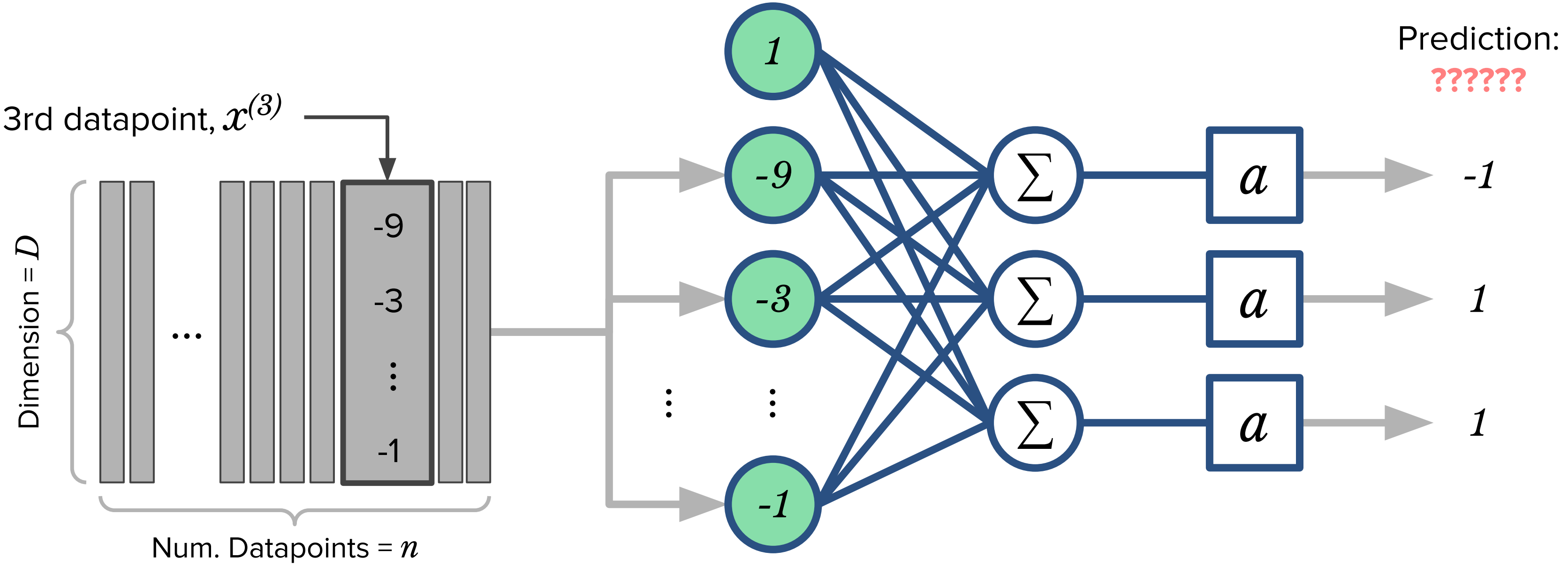
Multiclass perceptron

- Assuming we trained the three perceptrons, we can do a forward pass on them:



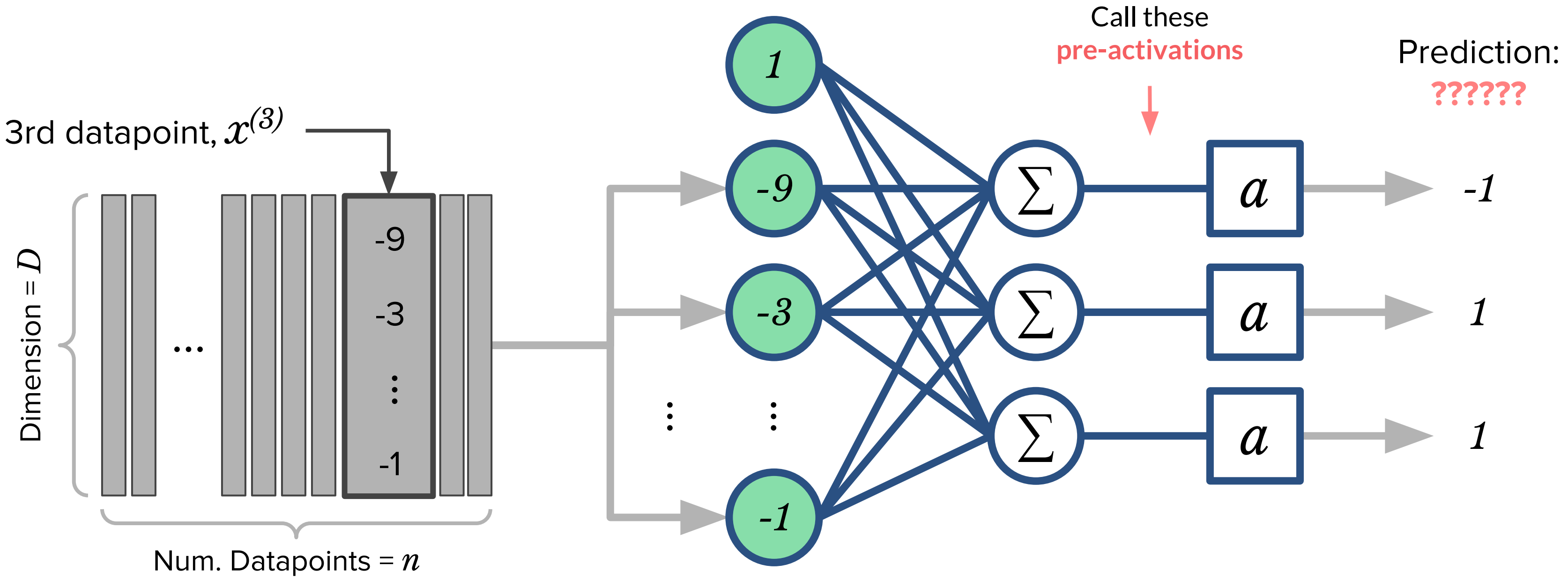
Multiclass perceptron

- What do we do when there is ambiguity?



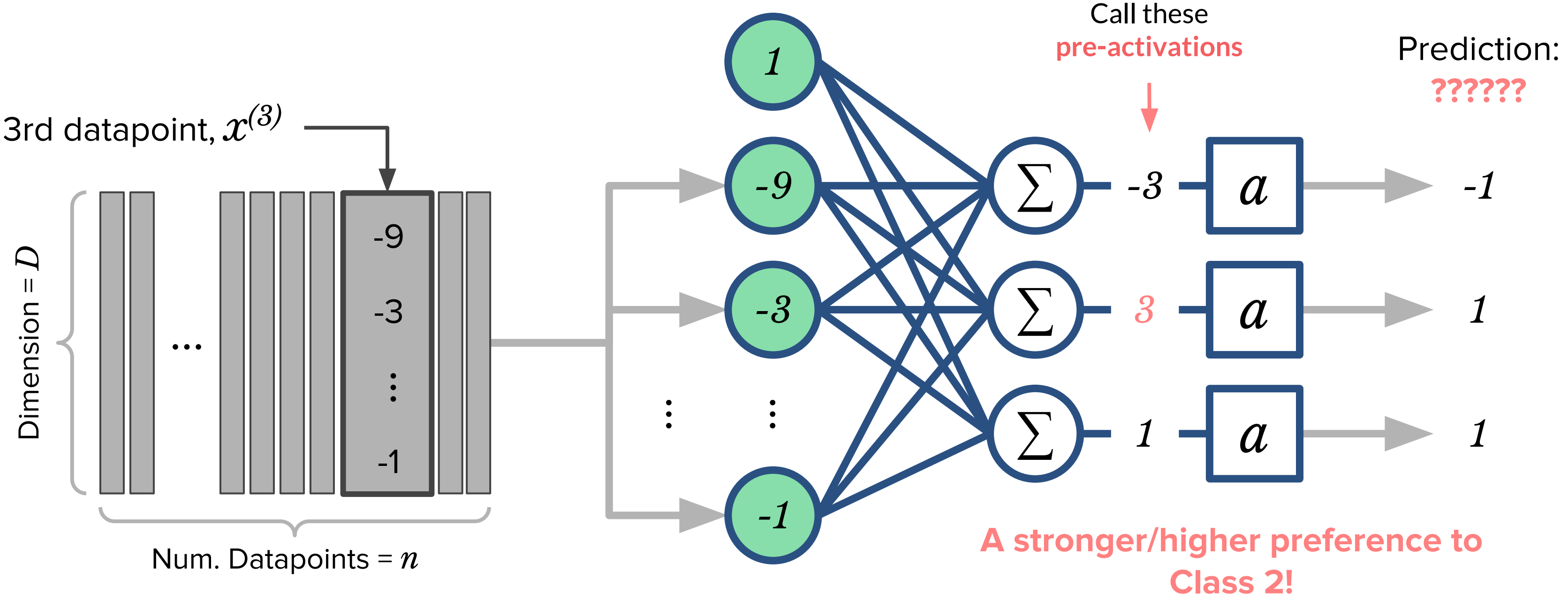
Multiclass perceptron

- What do we do when there is ambiguity?



Multiclass perceptron

- Via pre-activations (also called **logits**), we can check the class that the point "prefers" the most.



The Softmax Function

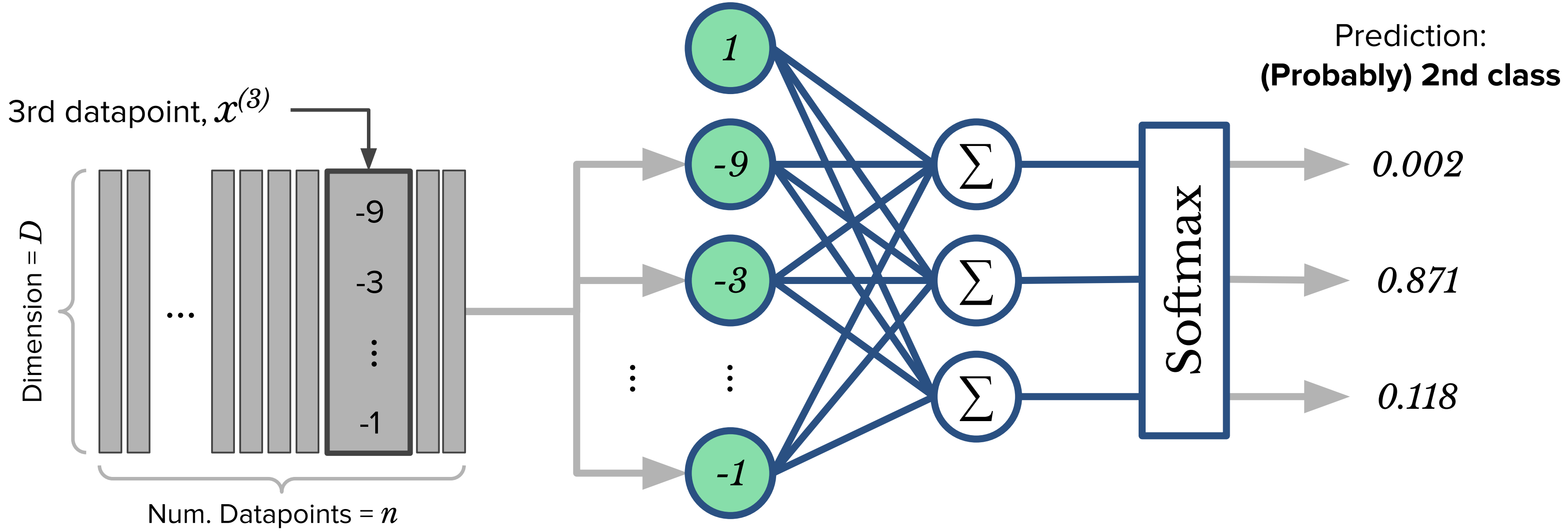
- Using **pre-activations** seems like a great way to avoid ambiguities, but they are set up is not ideal to compare them to the real classes (*as we'll see later*).
- A better approach would be to transform them into a **probability distribution** (i.e. make them all positive and summing to 1).
- Call the pre-activations $z = [z_1, z_2, \ldots, z_K]$. One way to turn z into a distribution is via the **softmax** function:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{c=1}^{K} \exp(z_c)}$$

  - The $\exp()$ function turns the z's into positive numbers.
  - The normalization makes the resulting numbers sum to 1.
- *Example*: $z = [-3, 3, 1] \rightarrow \text{softmax}(z) = [0.002, 0.878, 0.118]$
- In practice, we replace the perceptrons' activation functions by a softmax operation.

Multiclass perceptron with softmax

- With softmax the softmax operation, we have now the following model:



One-hot Enconding

- If the input of the model is called x, its output can be computed using the **multiclass perceptron formula**:

$$\hat{y} = \mathrm{softmax}(Wx)$$

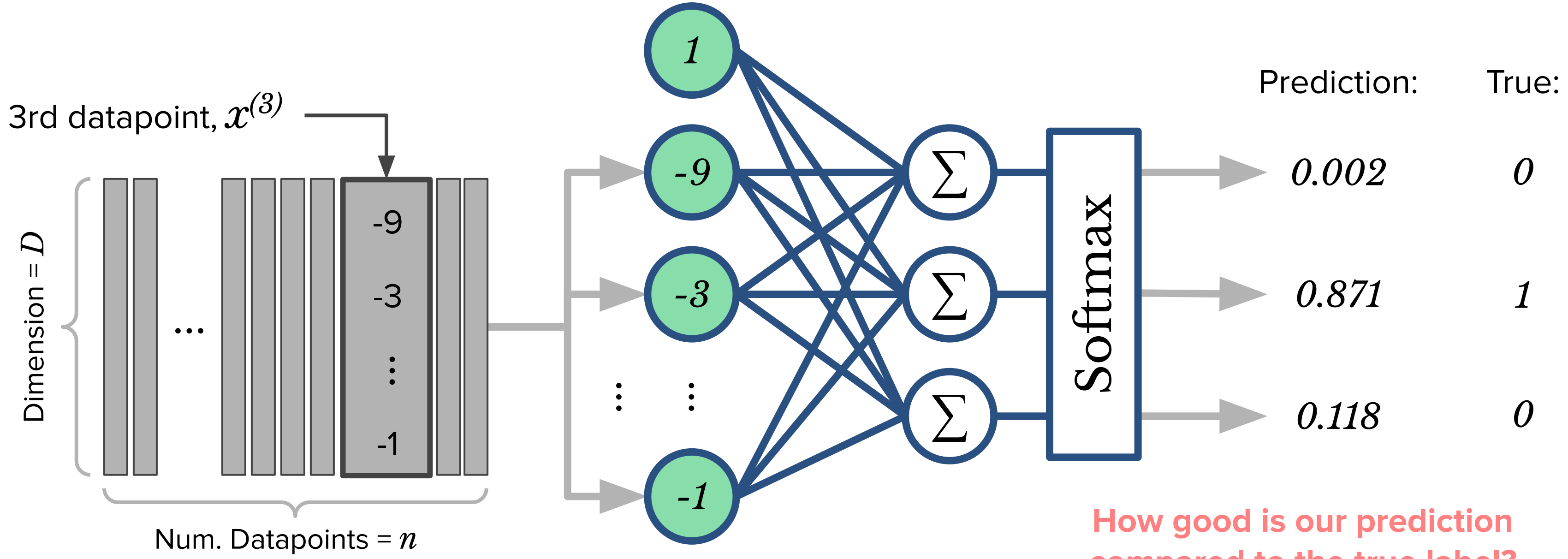
- We need to assess how good is the output of our model (say, $\hat{y}$ = $[0.22, 87.89, 11.89]$) for a point x in relation to the true label of that point y (which can be "circle").
- In Neural Networks, we use a **one-hot encoding** of the true labels, so we can compare them to our predictions.
- If we have K classes, the one-hot encoding of the labels will be vectors of K dimensions, **mostly made of zeros except at the dimension corresponding to the label value, where it is one.**
- *Example*: in our dataset, we may have

y_1 = "Triangle" $\rightarrow y_1 = [1, 0, 0]$
y_2 = "Square" $\rightarrow y_2 = [0, 0, 1]$
y_3 = "Circle" $\rightarrow y_3 = [0, 1, 0]$
...
y_n = "Square" $\rightarrow y_n = [0, 0, 1]$

- These encodings are also distributions (made of positive numbers that sum to 1)!

Multiclass perceptron with softmax

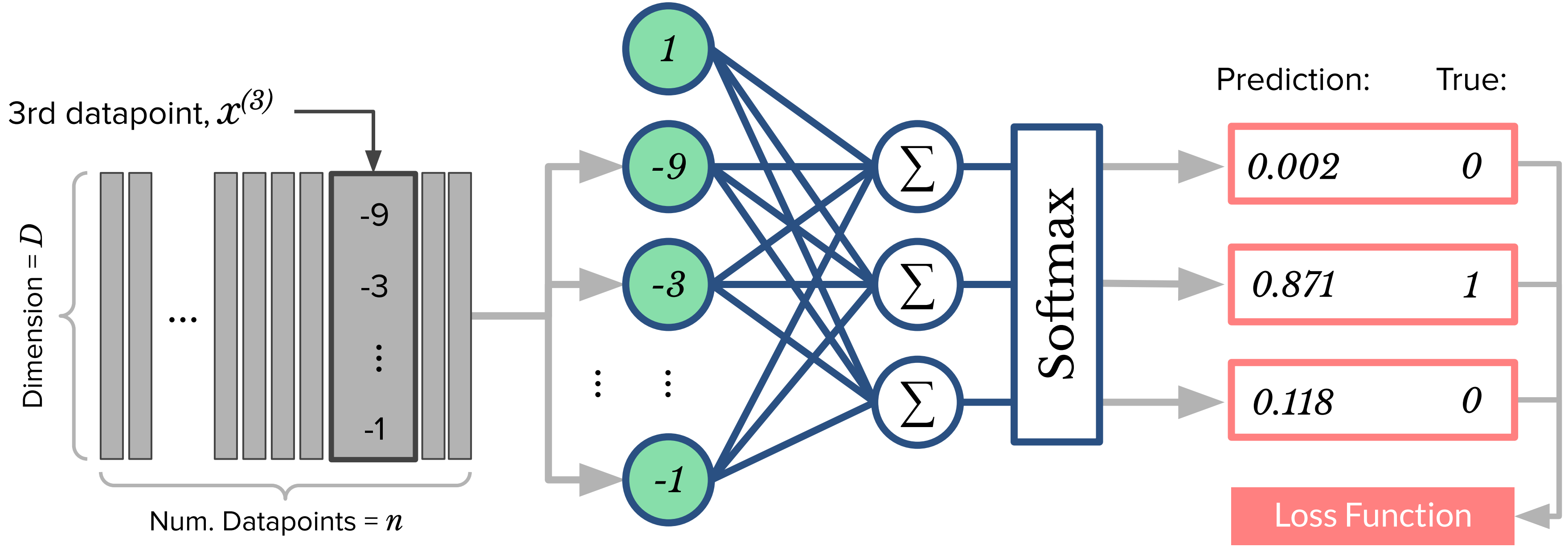
- Say the third point's true label is "circle", which is encoded as [$0, 1, 0$]:



How good is our prediction compared to the true label?

Multiclass perceptron with softmax

- We can now compare our prediction to the true label using a **loss function**.



Loss functions

- We need a set of weights that achieves the **best classification performance**.
- To that end, we can then use **loss (or cost) functions** $l(\hat{y}, y)$, that compares how similar our prediction $\hat{y}$ is to the true one-hot encoded label y.
- One of the most used loss functions* for multiclass problems is the **cross-entropy loss**:

$$l(\hat{y}, y) = -y^\top \log(\hat{y}) = -\sum_{j=1}^{K} y_j \log(\hat{y}_j)$$

- If we have a labeled dataset of size n, we can consider the **average loss** $L(\theta)$ on it:

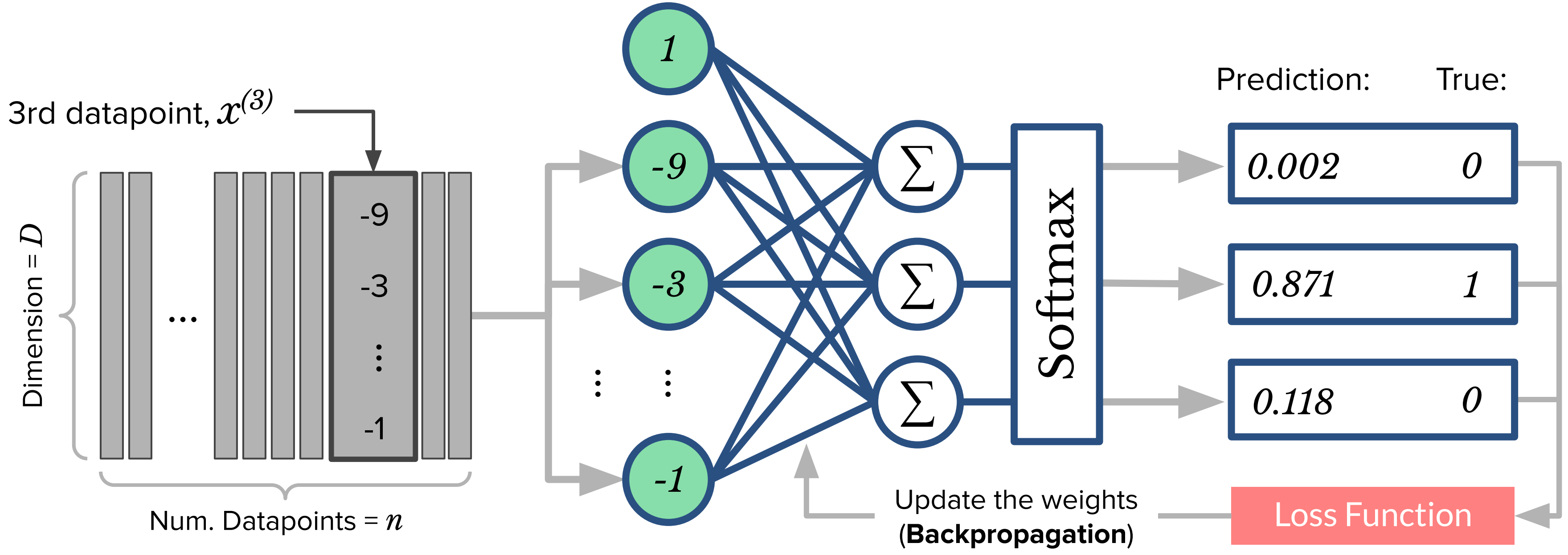
$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} l(\hat{y}^{(i)}, y^{(i)})$$

where θ represents all the parameters (weights, in the case of the perceptron or in Deep Learning generally speaking) used to compute $\hat{y}$ from x (*more on it next time*).

* There are many other losses used in Deep learning, and we'll see them as we go.

Multiclass perceptron with softmax

- We use the loss value to update the weights and improve the classifier (*more next time*).



Exercise (*In pairs*)

- How many weights are there to learn if we have input data in D dimensions and K classes in the output of our multiclass perceptron?
- Compute the cross-entropy loss for the following pairs true label/prediction (assume you only have these three data points in your dataset):
  - True labels:

$$y_1 = [1, 0, 0],\ y_2 = [0, 1, 0],\ y_3 = [0, 0, 1]$$

  - Predictions:

$$\hat{y}_1 = [1, 0, 0],\ \hat{y}_2 = [0.1, 0.8, 0.1],\ \hat{y}_3 = [1/3, 1/3, 1/3]$$

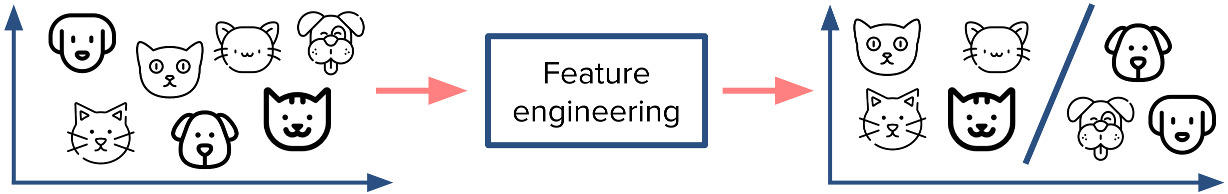
- Repeat the above exercise using the **Squared Error** loss*:

$$l(\hat{y}, y) = \|y - \hat{y}\|_2^2 = \sum_{j=1}^{K} (y_j - \hat{y}_j)^2$$

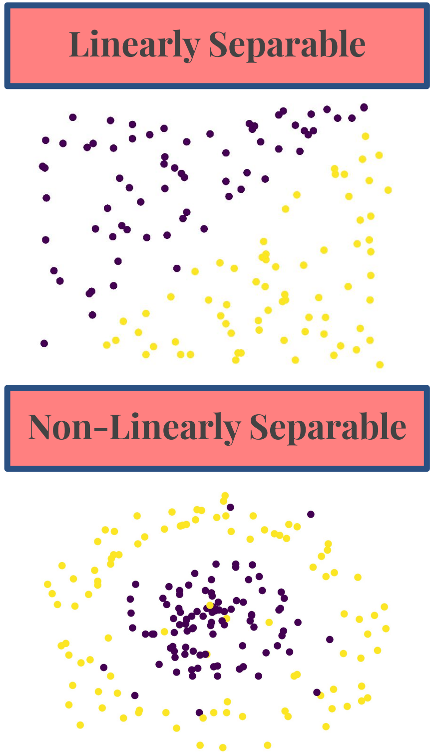
*The dataset's Average Loss $L(\theta)$ when using the squared error loss is the famous **Mean Squared Error (MSE) loss**.

Learning Representations

- Multiclass Perceptrons are still not performant when the classes are non-linearly separable (as on the right).
- Traditionally, this problem would be solved by **handcrafting a feature transformation** that would make the data separable:

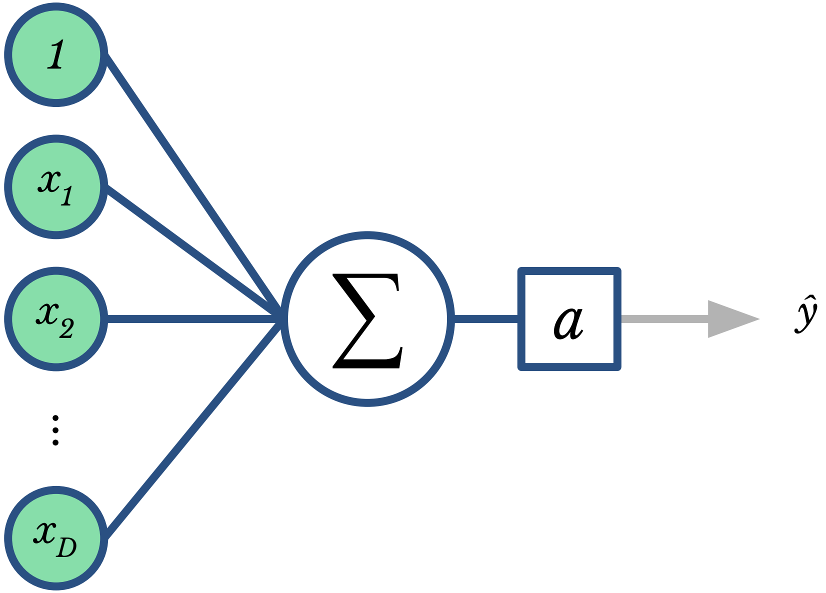


- *Example*: a feature that easily separates cats and dogs is the "*ear pointiness*". We could **represent** cats and dogs by that.
- In Deep Learning, we aim at **learning these representations** from a labeled dataset.
- To achieve that goal, we have to "evolve" our perceptrons.



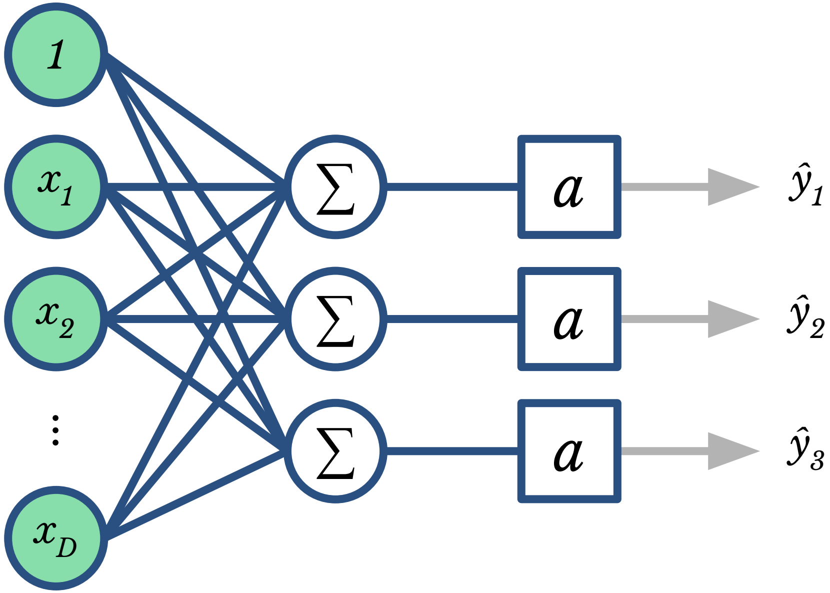
Perceptron Evolution

- The first perceptron had *1* neuron and only handled binary linearly separable problems.



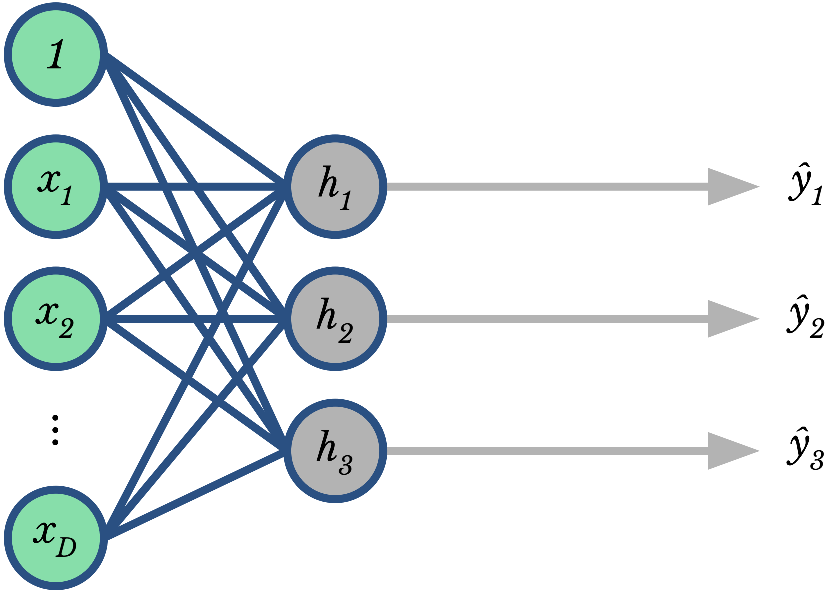
Perceptron Evolution

- Staking up some neurons, we were able to “solve” multiclass problems.



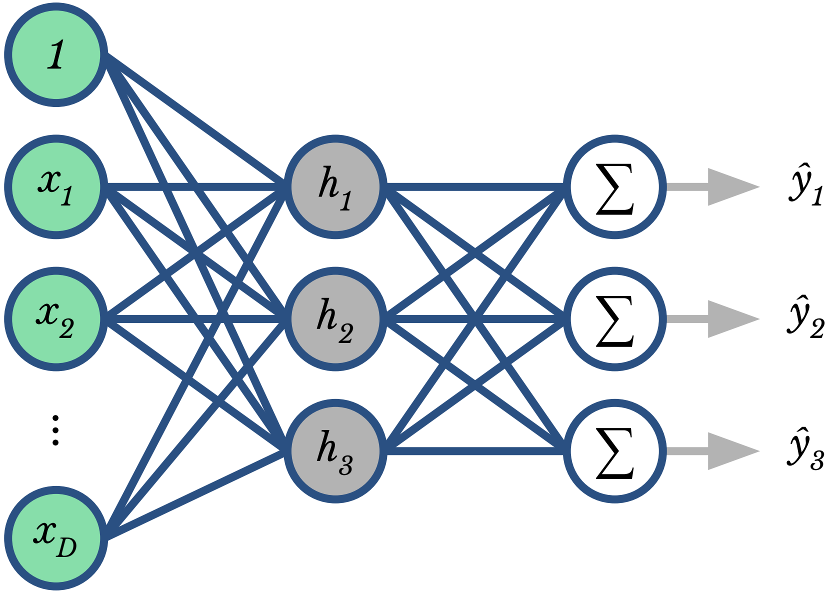
Perceptron Evolution

- We could, however, consider the outputs of the neurons as a “transformed” input …



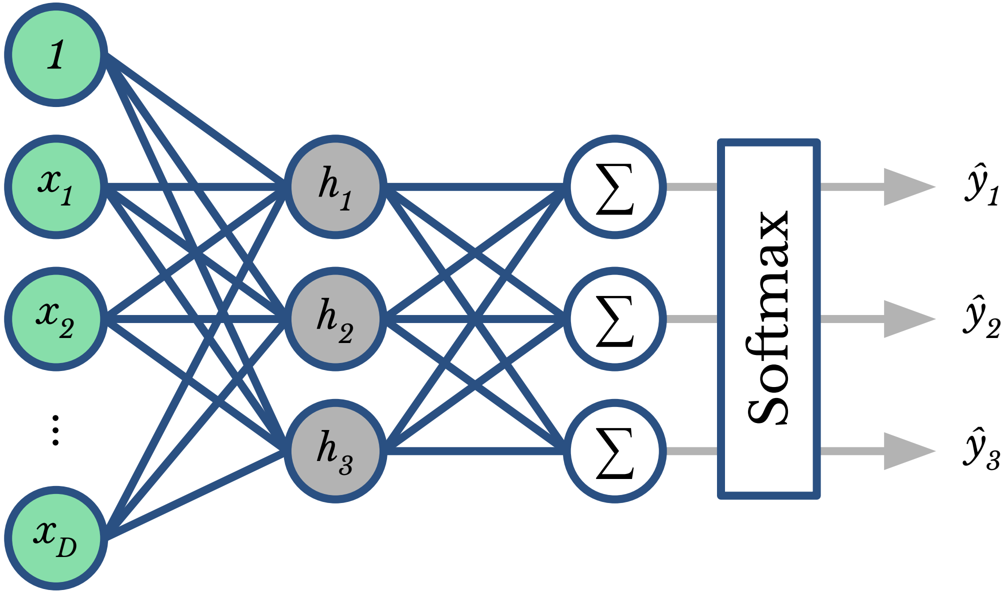
Perceptron Evolution

- … and the pass this new data into new sums, like what we did to x, …



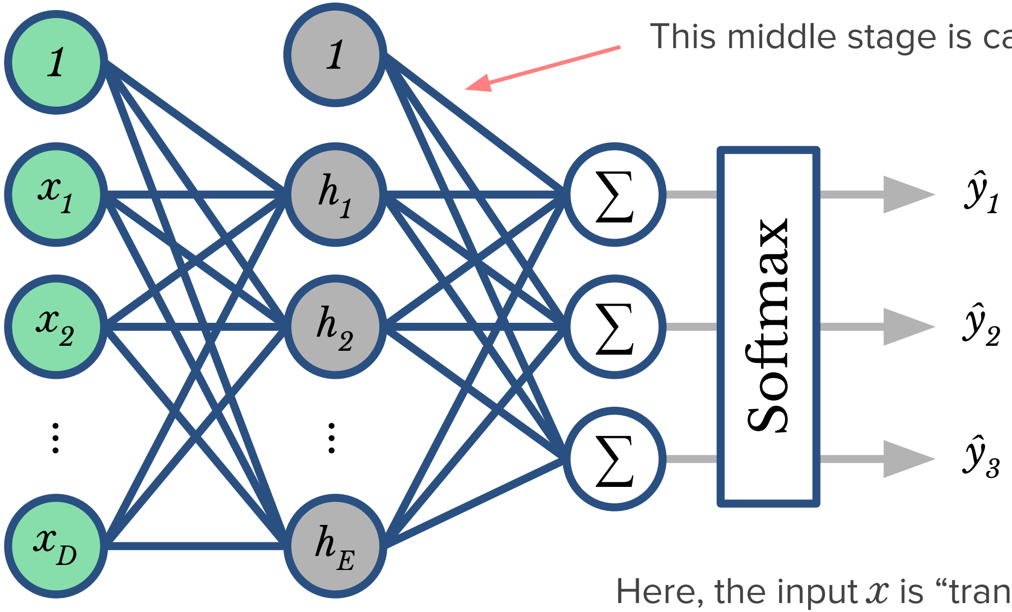
Perceptron Evolution

- … and only then make the results of the sums go through the softmax function.



Perceptron Evolution

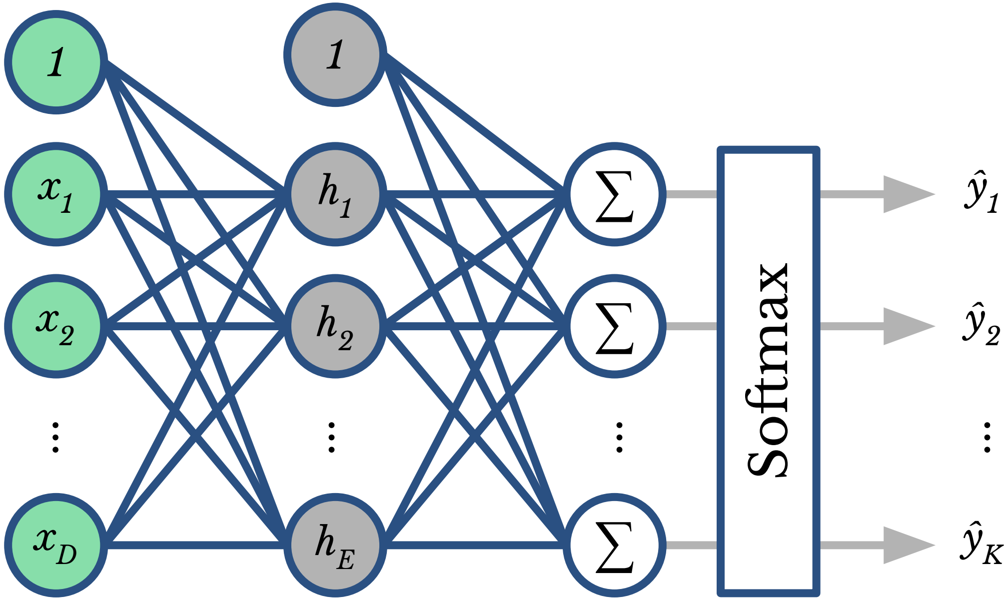
- We don't need to have only 3 neurons in the middle, and we can also add a bias term.



Here, the input x is "transformed into" a better representation for our classification problem

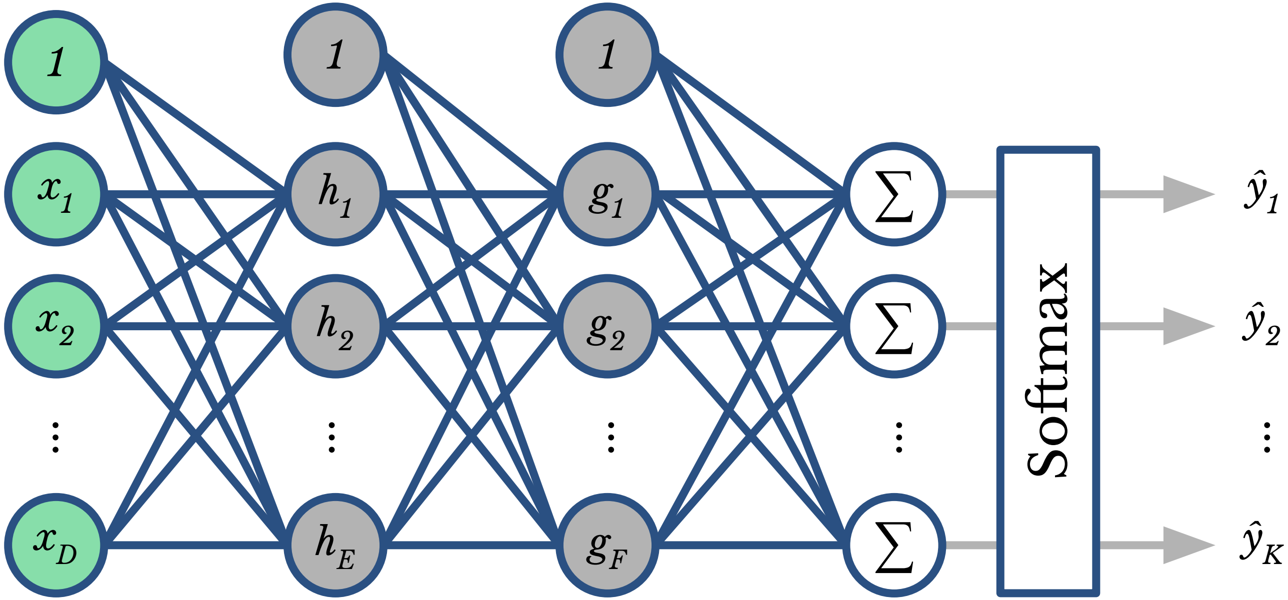
Perceptron Evolution

- This same idea can be expanded for the cases when we have any number of classes.



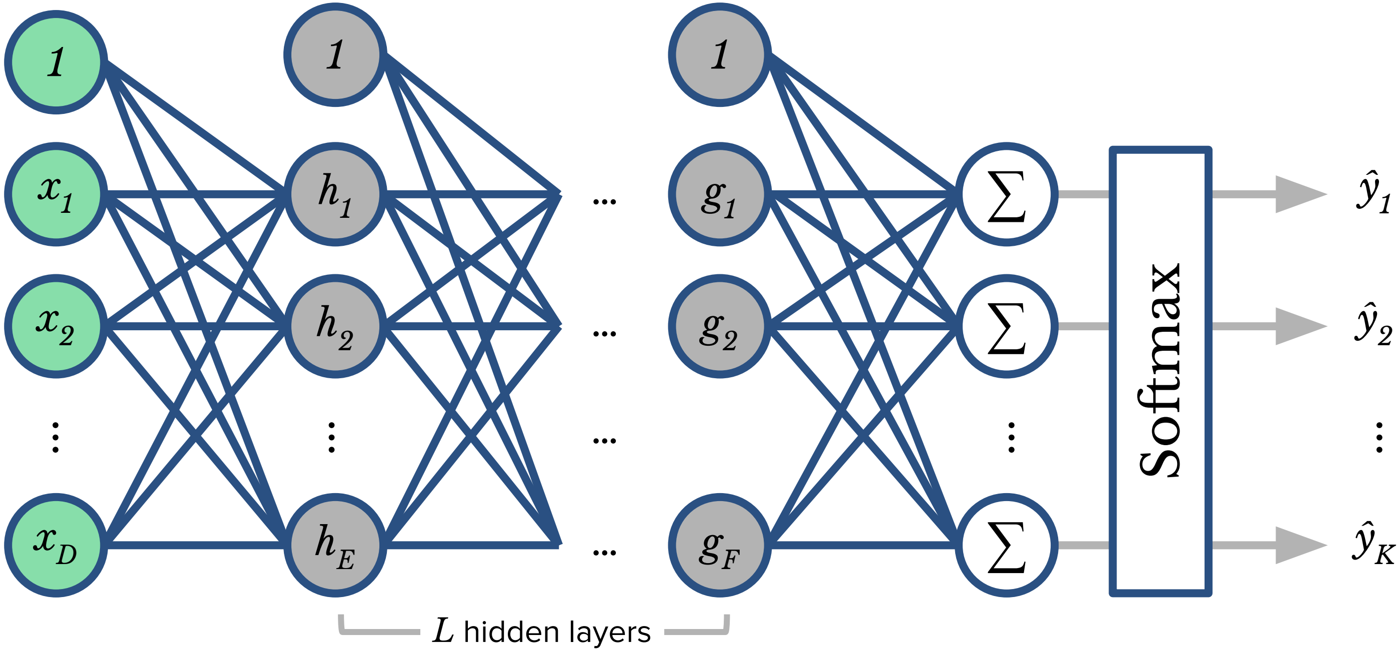
Perceptron Evolution

- Nobody will stop us from adding another hidden layer.



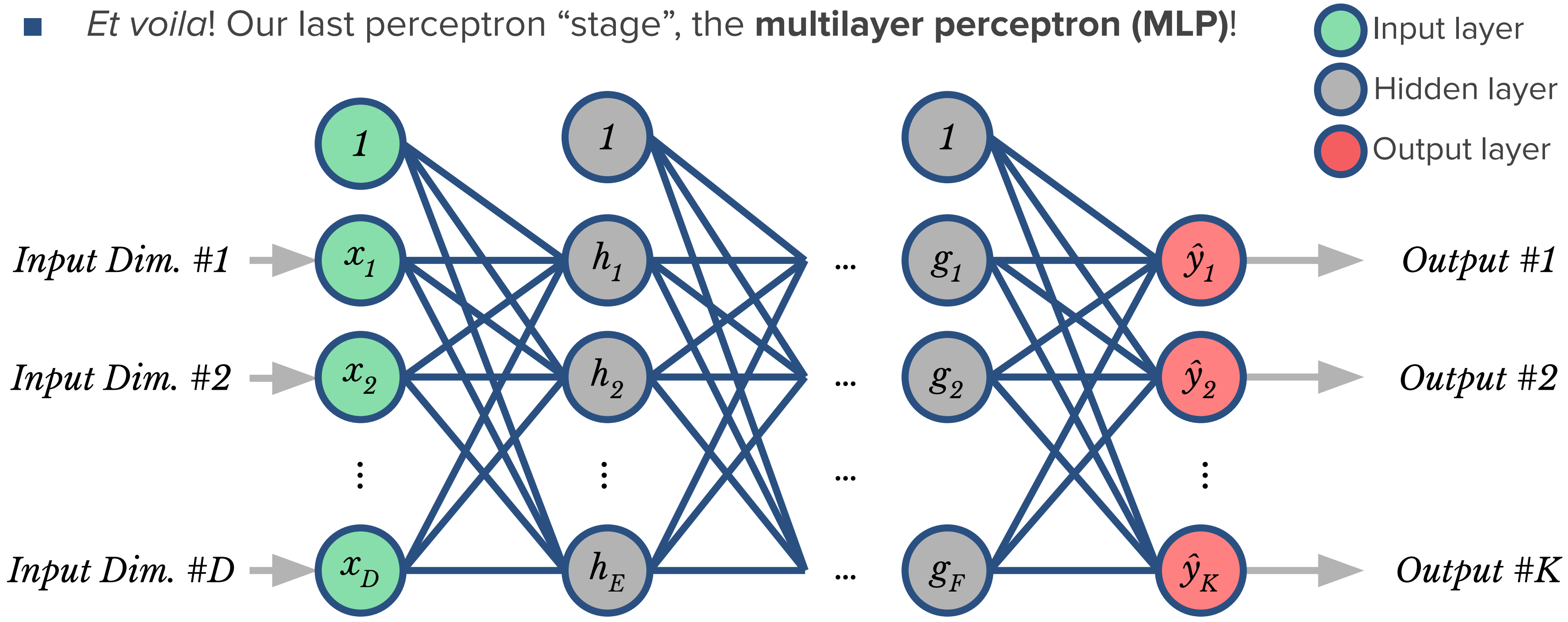
Perceptron Evolution

- Or in fact, as many hidden layers as we need.



Multilayer Perceptron

- *Et voila*! Our last perceptron "stage", the **multilayer perceptron (MLP)**!



Mathematically Speaking...

- The previous illustrations are useful for intuition, but we need to **describe the MLP mathematically**, so we can find the best weights for them (*more on that next time*).
- Let's first recall the simple multiclass perceptron. Calling its weight matrix W_0, we have:

$$\hat{y} = \mathrm{softmax}(W_0 x)$$

- When we added the first hidden layer, we changed the above formula to:

$$\hat{y} = \mathrm{softmax}(W_1 a(W_0 x))$$

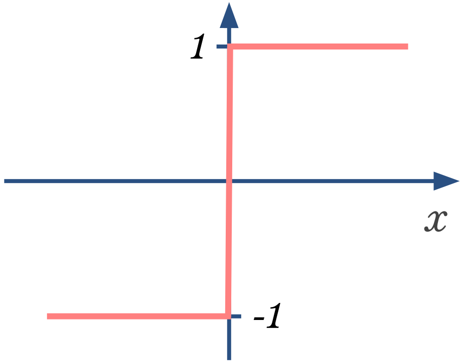
where a is the activation function applied to each of the outputs of the previous layer, i.e., if $W_0 x = [10, 20, 5]$, $a(W_0 x) = [a(10), a(20), a(5)]$.

- With L hidden layers now, we have the following expression MLP's output:

$$\hat{y} = \mathrm{softmax}(W_L a(W_{L-1} \cdots a(W_0 x) \cdots))$$

Other Activation Functions

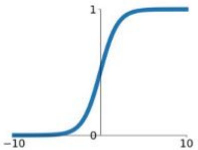
- The activation function we've used is the $\text{sign}(x)$ function:



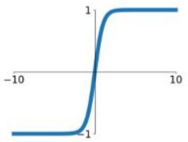
- It is, however, not great for backpropagation, as we shall see in next class, and it does not perform well in practice.

- Two more suitable activations are the sigmoid and the hyperbolic tangent (tanh), as they are **smooth** versions of the the sign function:

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

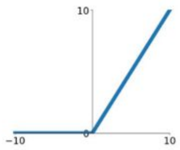


tanh
$\tanh(x)$



- The most widely used activation is the Rectified Linear Unit (ReLU):

ReLU
$\max(0, x)$



which, despite not smooth, works well in practice.

Hands-on Neural Networks

- How good is this new network to solve non-linearly separable classification problems?
- We'll see this in practice using a function from **Python's `sklearn` library**!
- First, we create* our dataset, called "blobs":

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=400, centers=4,
                  cluster_std =2, random_state=10)
```

- Then, we split* our dataset in training and test using the function "`train_test_split`" from `sklearn`:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X , y,
                                    test_size =0.4, random_state=2)
```

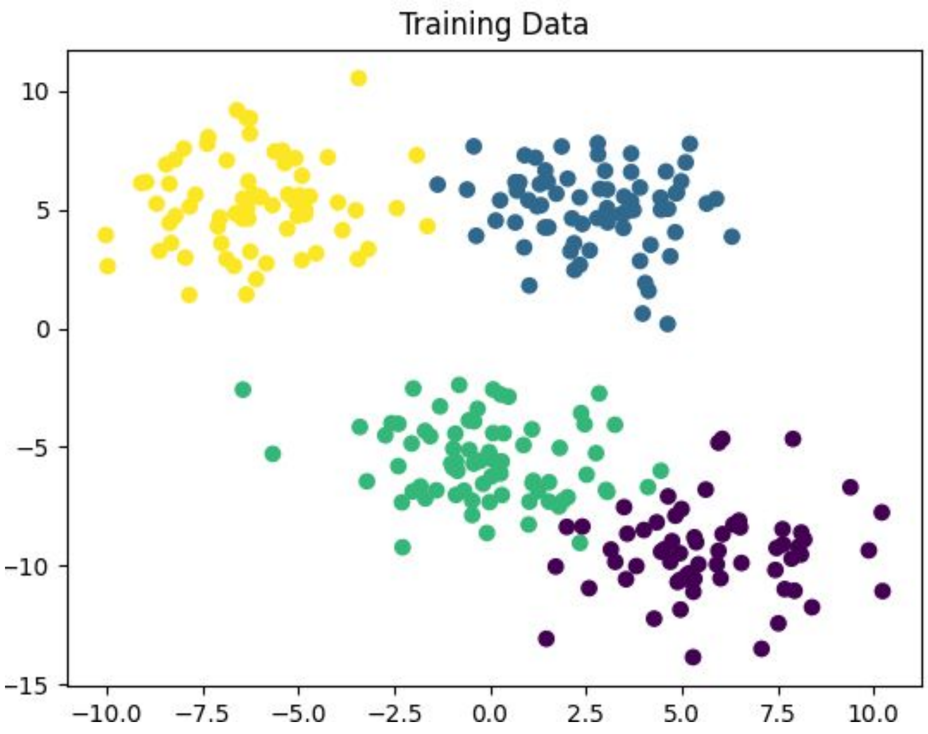
* In both functions we set the random states to those values, so you can reproduce the exact same results as in here.

Hands-on Neural Networks

- As **it is usually a good practice**, we then visualize the training data:

```
import matplotlib.pyplot as plt
plt.scatter(X_train[:, 0],
            X_train[:, 1],
            c=y_train)
plt.title("Training Data")
plt.show()
```

- Notice that, despite the classes being somewhat well defined, this is **not** a linearly separable dataset.
- Therefore, we should use a multilayer perceptron here.



Hands-on Neural Networks

- Now we can train a Multilayer Perceptron Classifier using `MLPClassifier`:

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=5, random_state=10)
clf.fit(X_train, y_train) # This command trains the MLP on the training data
```

- In this example, we only have **one hidden layer with 5 neurons***.

```
np.set_printoptions(suppress=True,
                    precision=2)
print(clf.predict_proba(X_test[0:3, :]))
print(clf.predict(X_test[0:3, :]))
print(y_test[0:3])
print(clf.score(X_test, y_test))
```

```
[[0.84 0.04 0.13 0.  ]
 [0.07 0.26 0.03 0.64]
 [0.22 0.22 0.26 0.29]]
[0 3 3]
[0 3 1]
0.7416666666666667
```

- Not bad, as a random classification would have around 1/4 = 25% accuracy.

* Here are the functions we are using: `predict_proba` gives the softmax response to each dataset, `predict` gives the predicted class for the listed points accord to `predict_proba`, and `score` outputs the overall accuracy of the classifier.

Hands-on Neural Networks

- Let’s see if we can improve this result by changing the number of neurons:

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=10, random_state=10)
clf.fit(X_train, y_train)
```

- Now we have **one hidden layer with 10 neurons**. Let’s see how it performs:

```
np.set_printoptions(suppress=True,
                    precision=2)
print(clf.predict_proba(X_test[0:3, :]))
print(clf.predict(X_test[0:3, :]))
print(y_test[0:3])
print(clf.score(X_test, y_test))
```

```
[[0.97 0.   0.03 0.  ]
 [0.   0.01 0.   0.99]
 [0.02 0.97 0.01 0.  ]]
[0 3 1]
[0 3 1]
0.9416666666666667
```

- Much better! Notice how the predictions’ probabilities are more “certain” of what class these points should belong to!

Hands-on Neural Networks

- Now, let's more layers, instead of just more neurons in one layer:

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(20, 10), random_state=10)
clf.fit(X_train, y_train)
```

- Now we have **two hidden layers with 20 and 10 neurons**, resp. So, it performs like:

```
np.set_printoptions(suppress=True,
                    precision=2)
print(clf.predict_proba(X_test[0:3, :]))
print(clf.predict(X_test[0:3, :]))
print(y_test[0:3])
print(clf.score(X_test, y_test))
```

```
[[1.   0.   0.   0.  ]
 [0.   0.01 0.01 0.99]
 [0.   0.99 0.   0.  ]]
[0 3 1]
[0 3 1]
0.9833333333333333
```

- Almost perfect! And the softmax probabilities are almost sure about their predictions!
- But, how long did "`fit`" take to run?

Hands-on Neural Networks

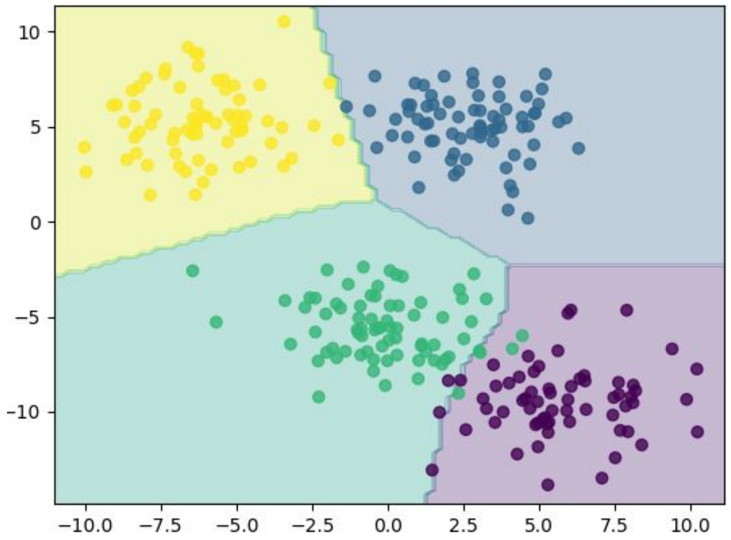
- We can also visualize the decision boundaries using the function:

```
def plot_decision_boundaries(X, y, model_class, **model_params):
    model = model_class(**model_params, random_state=10)
    model.fit(X, y)

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    h = .01 * np.mean([x_max - x_min, y_max - y_min])
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3)
    plt.scatter(X[:, 0], X[:, 1], c=y, alpha=0.8)
    plt.show()
```

Hands-on Neural Networks

- Now we can visualize the training data with the decision boundary:

```
plot_decision_boundaries(X_train , y_train, MLPClassifier,
                         hidden_layer_sizes =(20, 10))
```



Exercise (*In pairs*)

Click here to open code in Colab

- How many weights we need to learn if or MLP has 3 layers with 10, 5 and 20 neurons, respectively, and we have points of dimension 100, belonging to 3 different classes?
- Run the Multilayer Perceptron on the following datasets

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05)
```

```
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=200, noise=0.05)
```

- Try different numbers of MLP layers and neurons.
- Compute the score of each classification.
- Visualize the decision boundaries in each case using `plot_decision_boundaries`.