Hyper parameter tuning

Learning Objectives:

- Master the process of hyperparameter tuning
- Describe softmax classification for multiple classes
- Apply batch normalization to make your neural network more robust
- Build a neural network in TensorFlow and train it on a TensorFlow dataset
- Describe the purpose and operation of GradientTape
- Use tf.Variable to modify the state of a variable
- Apply TensorFlow decorators to speed up code

Hyper parameter tuning

1 Tuning process
2 Using an appropriate scale to pick hyperparameters
3 Hyperparameters tuning in practice: Pandas vs Caviar
4 Normalizing activations in a network
5 Fitting Batch Norm into a neural network
6 Why does Batch Norm work?
7 Softmax regression
8 Deep Learning frameworks
9 TensorFlow

FPT Education

FPT UNIVERSITY

Hyperparameter tuning

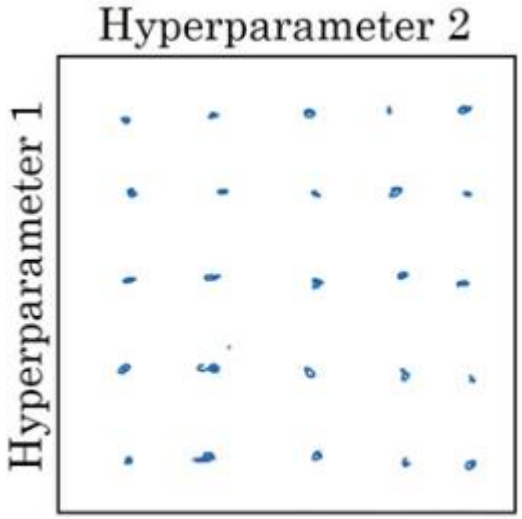
Tuning process

Hyperparameters

- Tuning these hyperparameters effectively can lead to a massive improvement in your position on the leaderboard. Following are a few common hyperparameters we frequently work with in a deep neural network:
  - Learning rate – α
  - Momentum – β
  - Adam's hyperparameter – β_1, β_2, ε
  - Number of hidden layers
  - Number of hidden units for different layers
  - Learning rate decay
  - Mini-batch size
- Learning rate usually proves to be the most important among the above. This is followed by the number of hidden units, momentum, mini-batch size, the number of hidden layers, and then the learning rate decay.

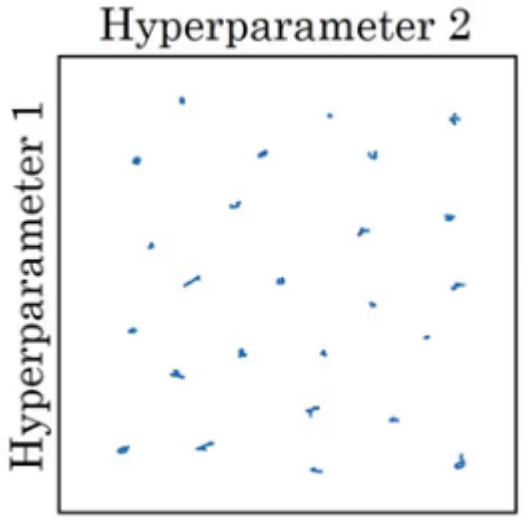
Hyperparameters

- Now, suppose we have two hyperparameters. We sample the points in a grid and then systematically explore these values. Consider a five-by-five grid:



Hyperparameters

- We check all 25 values and pick whichever hyperparameter works best. Instead of using these grids, we can try random values as well. Why? Because we do not know which hyperparameter value might turn out to be important, and in a grid we only define particular values.



- The major takeaway from this sub-section is to use random sampling and adequate search.

FPT UNIVERSITY

Using an appropriate scale to pick hyperparameters

Picking hyperparameters at random

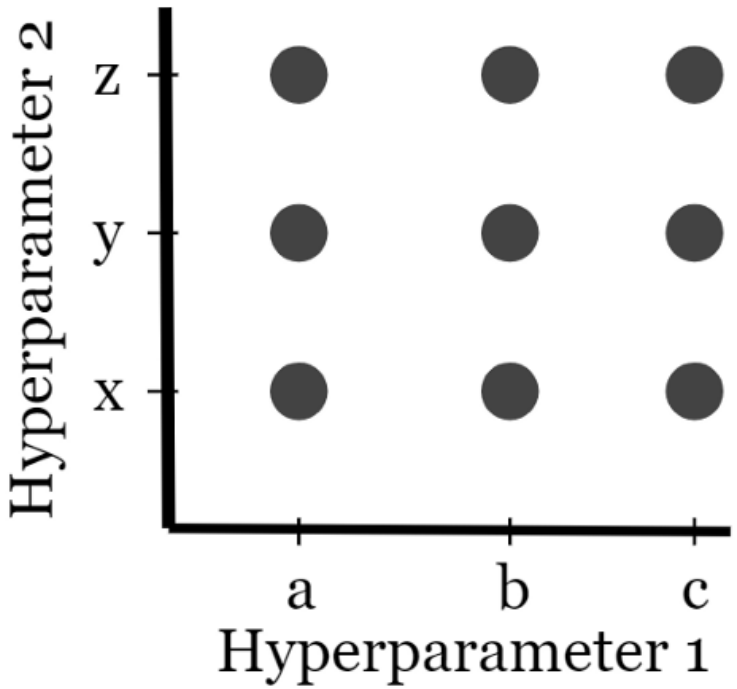
- While sampling at random can be a useful way to search for hyperparameters, it's not always the best approach.
- For example, if you're trying to select the number of hidden units for a given layer, sampling uniformly at random over a range of values may be appropriate.

Picking hyperparameters at random

Grid Search

Pseudocode

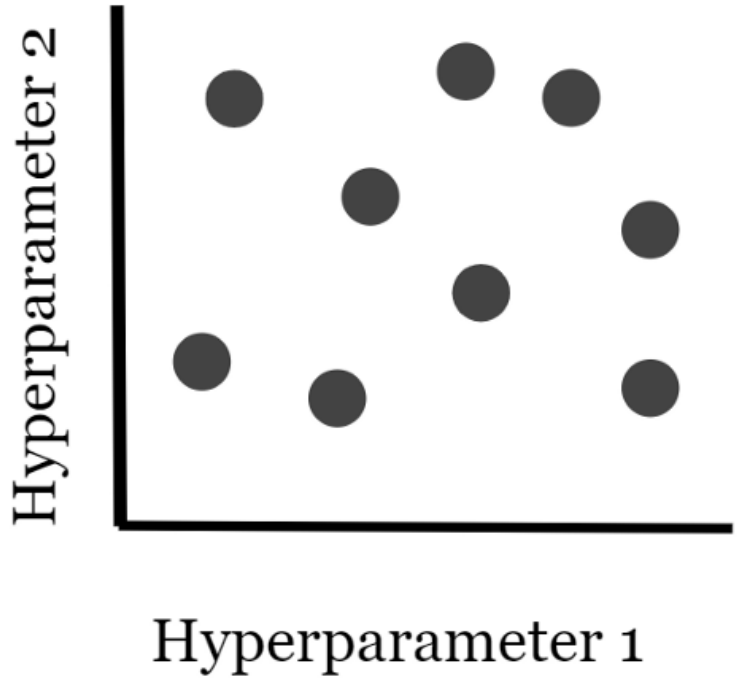
```
Hyperparameter_One = [a, b, c]
Hyperparameter_Two = [x, y, z]
```



Random Search

Pseudocode

```
Hyperparameter_One = random.num(range)
Hyperparameter_Two = random.num(range)
```



Appropriate scale for hyperparameters

- Randomly selecting values from a uniform distribution might not be ideal for certain hyperparameters like learning rate (alpha) or beta in exponentially weighted averages.
- For parameters like alpha, a more effective approach is to sample on a logarithmic scale, prioritizing the lower end where small changes can significantly impact results.
  - To sample on a logarithmic scale in Python, you can use the formula alpha = 10 to the power of r, where r is a random number between -4 and 0.

Appropriate scale for hyperparameters

- For hyperparameters like beta, explore the range of values for 1 minus beta (0.1 to 0.001) by sampling uniformly at random.
- This approach allocates more resources to the range where small changes in beta have a significant impact.
- While selecting the appropriate scale is crucial, note that even suboptimal scaling decisions can yield acceptable results.
- A coarse-to-fine search is also effective, focusing on the most useful hyperparameter values in later iterations.

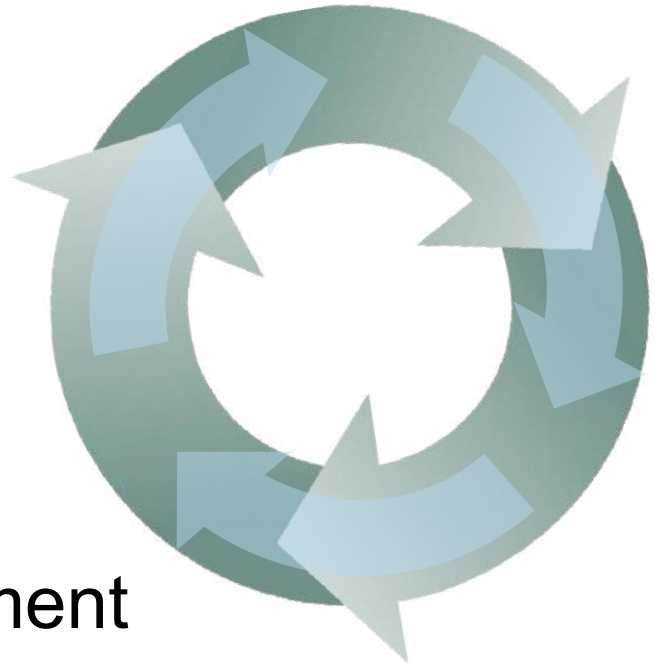
FPT UNIVERSITY

Hyperparameters tuning in practice Pandas vs Caviar

Hyperparameters tuning in practice Pandas vs Caviar

- For efficient hyperparameter search in deep learning, consider that intuitions may not universally transfer across application areas.
- Cross-fertilization between domains is valuable, and exploring research papers from different areas can provide inspiration.
- It's advisable to retest or reevaluate hyperparameters periodically, ensuring that the best settings remain optimal amid changes in data, algorithm development, or server upgrades.

Re-test hyperparameters occasionally



- NLP, Vision, Speech, Ads, logistics, ….

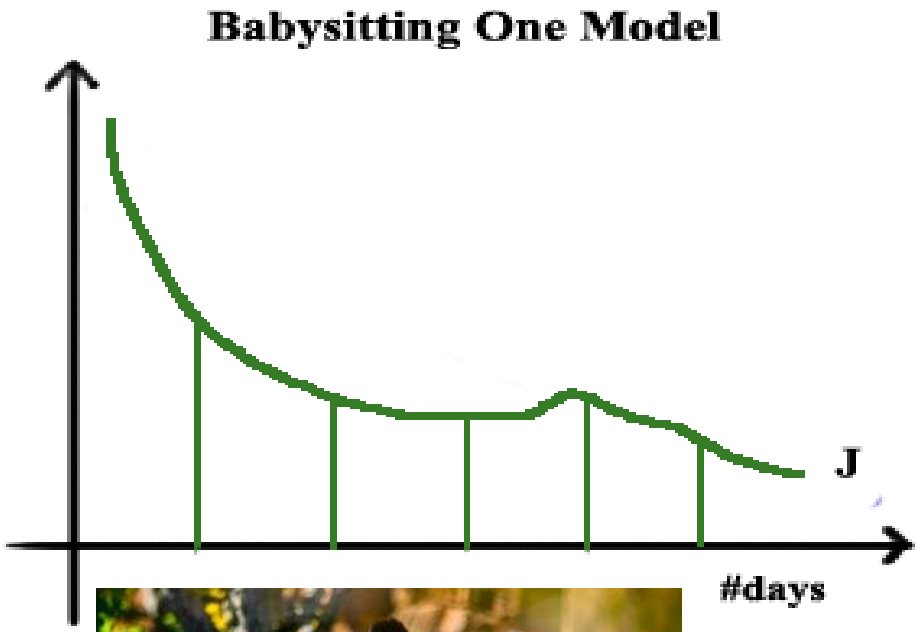
- Intuitions do get stale. Re-evaluate occasionally.

Hyperparameters tuning in practice Pandas vs Caviar

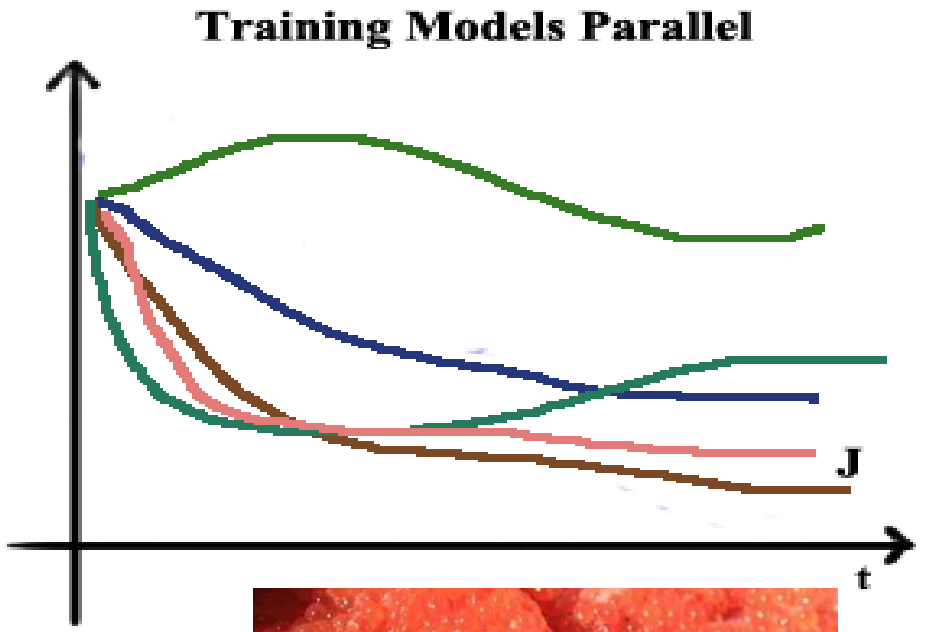
- For hyperparameter tuning, there are two main approaches: the "panda approach," where users carefully adjust a single model's learning rate, and the "caviar strategy," involving parallel training of multiple models with diverse hyperparameters to quickly identify the best-performing one.
- The choice between these approaches depends on available computational resources.
- Additionally, certain techniques can enhance neural network robustness to hyperparameter choices, simplifying the search process and accelerating training in applicable cases.

Babysitting one model

Training many models in parallel



Panda



Caviar

FPT UNIVERSITY

Batch Normalization

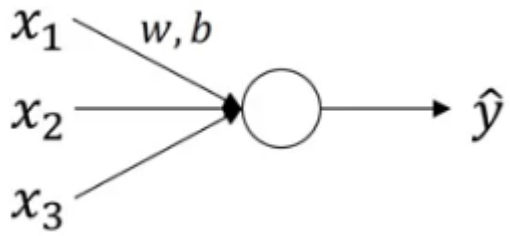
Normalizing activations in a network

Normalizing inputs to speed up learning

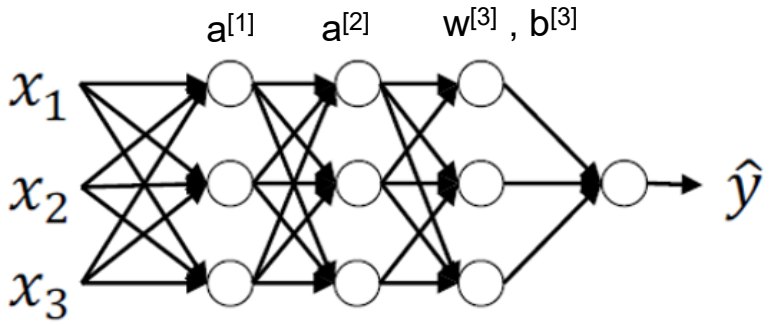
- Batch normalization, developed by Sergey Ioffe and Christian Szegedy, normalizes mean and variance of hidden unit values in deep neural networks.
- Similar to input feature normalization in logistic regression, batch normalization enhances the efficiency of training in networks with multiple hidden layers.
- The process of batch normalization implementation involves computing mean and variance, followed by normalization.
- Learnable parameters, gamma and beta, are used to adjust mean and variance to desired values, preventing them from being fixed at 0 and 1.

Normalizing inputs to speed up learning

- Let's recall how a logistic regression looks like:



- We have seen how normalizing the input in this case can speed up the learning process. In case of deep neural networks, we have a lot of hidden layers and this results in a lot of activations:



- **Wouldn't it be great if we can normalize the mean and variance of these activations (a[2]) in order to make the training of w[3], b[3] more effective**?

Normalizing inputs to speed up learning

- This is how batch normalization works. We normalize the activations of the hidden layer(s) so that the weights of the next layer can be updated faster. Technically, we normalize the values of $Z^{[2]}$ and then use an activation function of the normalized values. Here is how we can implement batch normalization:
- Given some intermediate values in NN $Z^{[1]},\ldots,Z^{[m]}$:

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{\text{norm}} + \beta$$

- Here, ɣ and β are learnable parameters.

Implementing Batch Norm

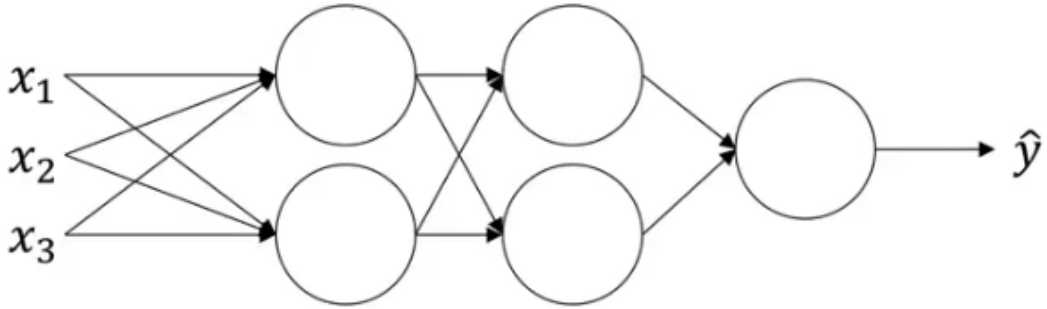
- Gamma and beta in batch normalization allow you to customize the mean of normalized hidden unit values, providing flexibility in subsequent computations.
- Batch normalization simplifies hyperparameter search, enhances neural network robustness, and facilitates training deep networks.

FPT UNIVERSITY

Fitting Batch Norm into a neural network

Adding Batch Norm to a network

Consider the neural network shown below:



Each unit of the neural network computes two things. It first computes Z, and then applies the activation function on it to compute A. If we apply batch norm at each layer, the computation will look like:

$$X \xrightarrow{W^{[1]},\ b^{[1]}} Z^{[1]} \xrightarrow[\text{Batch norm}]{\beta^{[1]},\ \gamma^{[1]}} \tilde{Z}^{[1]} \Longrightarrow A^{[1]} \xrightarrow{W^{[2]},\ b^{[2]}} Z^{[2]} \xrightarrow[\text{Batch norm}]{\beta^{[2]},\ \gamma^{[2]}} \tilde{Z}^{[2]} \Longrightarrow A^{[2]} \ldots$$

After calculating the Z-value, we apply batch norm and then the activation function on that.
Parameters in this case are:
$W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$, …, $W^{[L]}$, $b^{[L]}$
$\beta^{[1]}$, $\gamma^{[1]}$, $\beta^{[2]}$, $\gamma^{[2]}$, …, $\beta^{[L]}$, $\gamma^{[L]}$

Implementing gradient descent

- For t=1, ….., number of mini batches:
- Compute forward propagation on $X^{\{t\}}$
- In each hidden layer, use batch normalization
- Use backpropagation to compute $dW^{[l]}$, $db^{[l]}$, $d\beta^{[l]}$ and $d\gamma^{[l]}$
- Update the parameters:
- $W^{[l]} = W^{[l]} - \alpha * dW^{[l]}$
- $\beta^{[l]} = \beta^{[l]} - \alpha * d\beta^{[l]}$

Why does Batch Norm work?

Why does Batch Norm work?

FPT Education
FPT UNIVERSITY

- The first reason is the same reason as why we normalize X.
- The second reason is that batch normalization reduces the problem of input values changing (shifting).
- Batch normalization does some regularization:

  - Each mini batch is scaled by the mean/variance computed of that mini-batch.

  - This adds some noise to the values $Z^{[l]}$ within that mini batch. So similar to dropout it adds some noise to each hidden layer's activations.

  - This has a slight regularization effect.

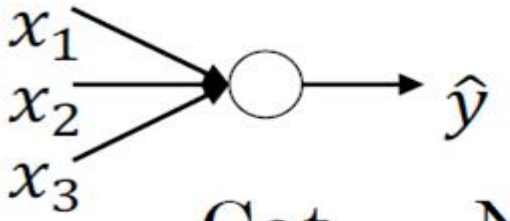
  - Using bigger size of the mini-batch you are reducing noise and therefore regularization effect.

  - Don't rely on batch normalization as a regularization. It's intended for normalization of hidden units, activations and therefore speeding up learning. For regularization use other regularization techniques (L2 or dropout).

Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $Z^{[L]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

Learning on shifting input distribution



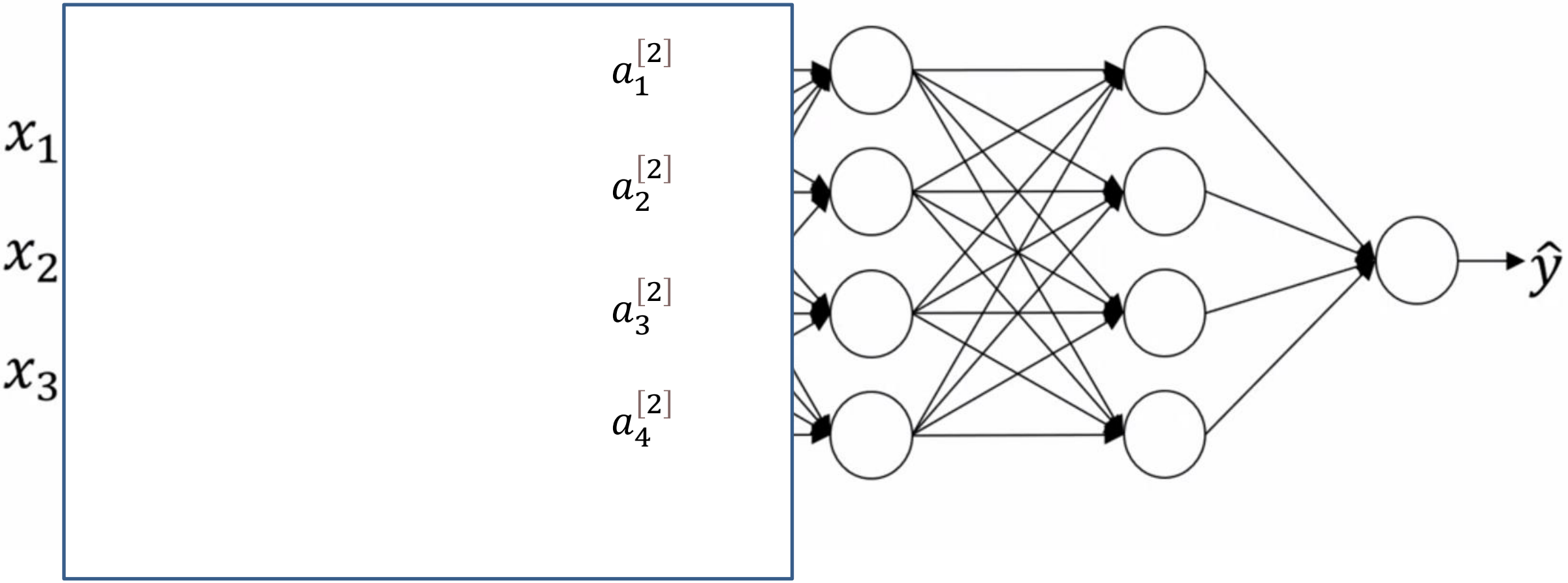
Cat $y = 1$

Non-Cat $y = 0$

$y = 1$

$y = 0$

Why this is a problem with neural networks?



FPT Education

FPT UNIVERSITY

Multi-class classification

Softmax regression

Softmax regression

- In multi-class classification, we deal with more than two classes, and softmax regression is employed to address such problems.
- The output layer has units corresponding to the total number of classes, each indicating the probability of the input belonging to a specific class.
- The Softmax layer transforms the linear output into probabilities by exponentiating and normalizing the values. This layer allows the representation of linear decision boundaries between multiple classes, and adding hidden layers enables the learning of complex non-linear decision boundaries.

Recognizing cats, dogs, and baby chicks

- Consider an example of the images and the classes they belong to. That's a picture of a baby chick, a cat, a dog and a koala (other).







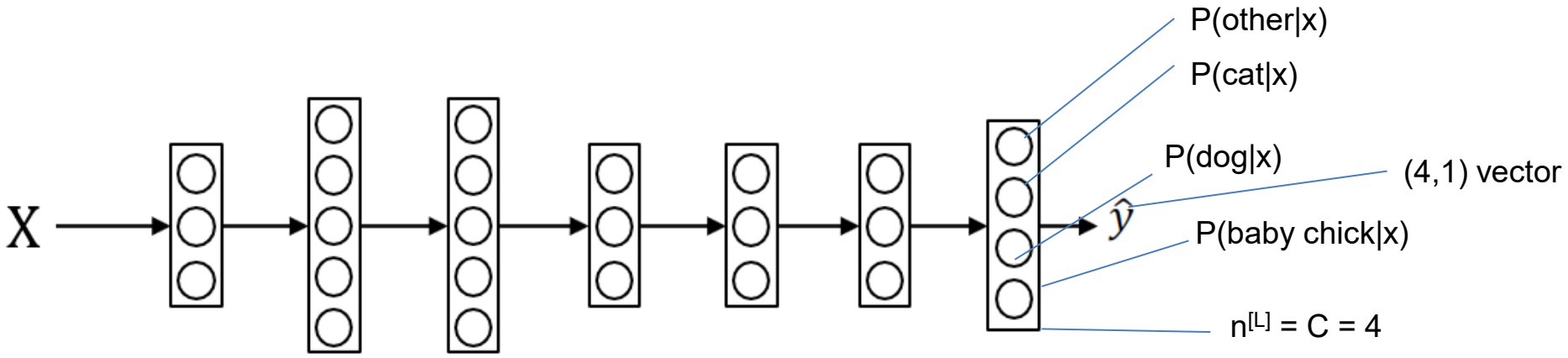

3 1 2 0 3 2 0 1

Recognizing cats, dogs, and baby chicks

- If we are classifying by classes ‘cat’, ‘dog’, ‘baby chick’ and ‘other’:
  - Cat class = 1
  - Dog class = 2
  - Baby chick class = 3
  - None class = 0
  - To represent a cat vector y = [0 1 0 0]
  - To represent a dog vector y = [0 0 1 0]
  - To represent a baby chick vector y = [0 0 0 1]
  - To represent a other vector y = [1 0 0 0]
  - Notations:
- - C = no. of classes
- - Range of classes is (0, ..., C-1)
- - In output layer Ny = C

Recognizing cats, dogs, and baby chicks

3 1 2 0 3 2 0 1



- Each of C values in the output layer will contain a probability of the example to belong to each of the classes.
- In the last layer we will have to activate the Softmax activation function instead of the sigmoid activation.

Recognizing cats, dogs, and baby chicks

- This is how a neural network for a multi-class classification looks like. So, for layer L, the output will be:

$$Z^{[L]} = W^{[L]} * a^{[L-1]} + b^{[L]}$$

- The activation function will be:

$$t = e^{(z^{[L]})}$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i} \quad , a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

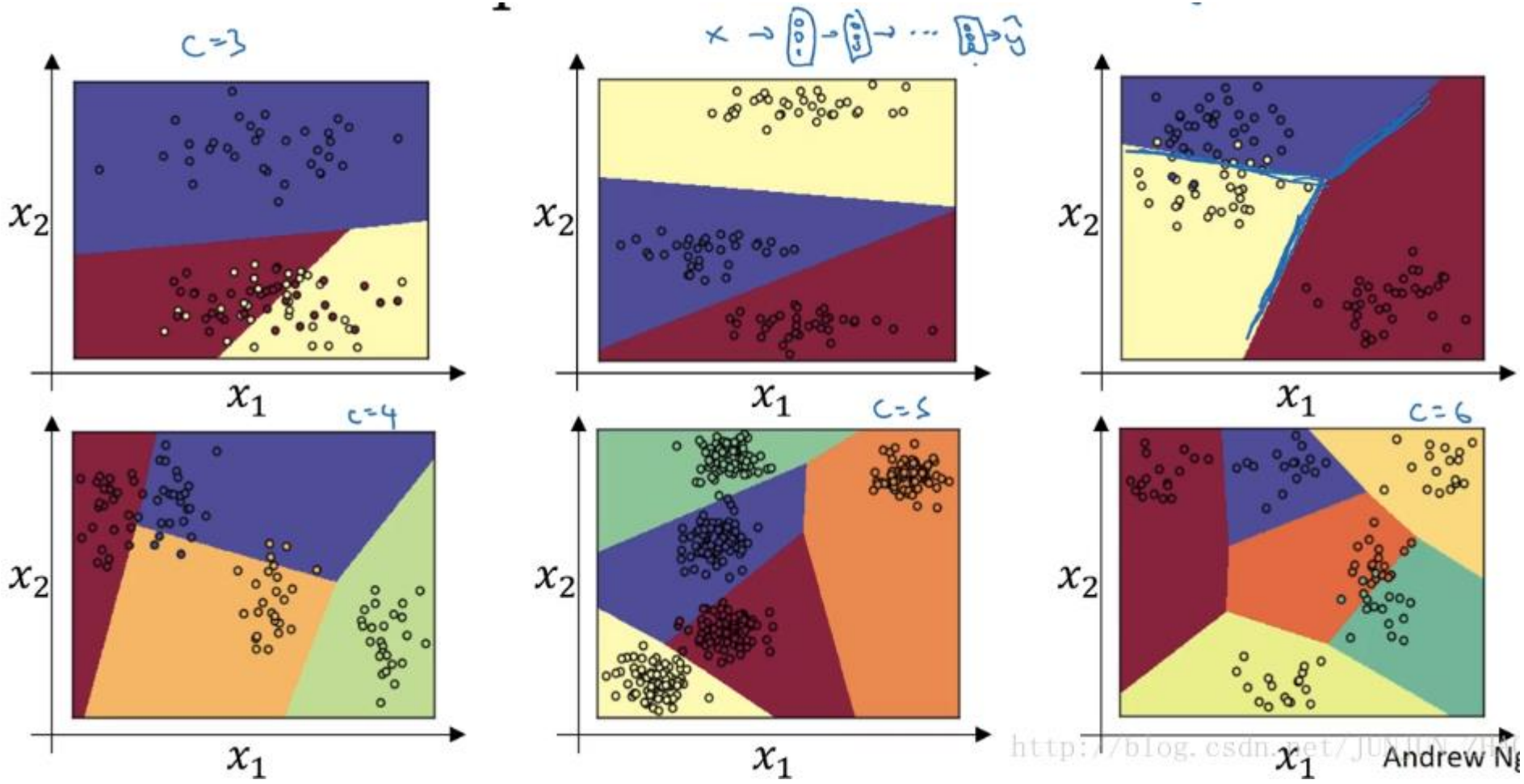
Recognizing cats, dogs, and baby chicks

- Let's say that $Z^{[L]}$ is a four dimensional vector 5, 2, -1, 3

$$Z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \Rightarrow t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \Rightarrow t = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \Rightarrow \sum_{j=1}^{4} t_j = 176.3$$

$$\Rightarrow a^{[L]} = \frac{t}{176.3} \Rightarrow A^{[L]} = \begin{bmatrix} 148.4/176.3 \\ 7.4/176.3 \\ 0.4/176.3 \\ 20.1/176.3 \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

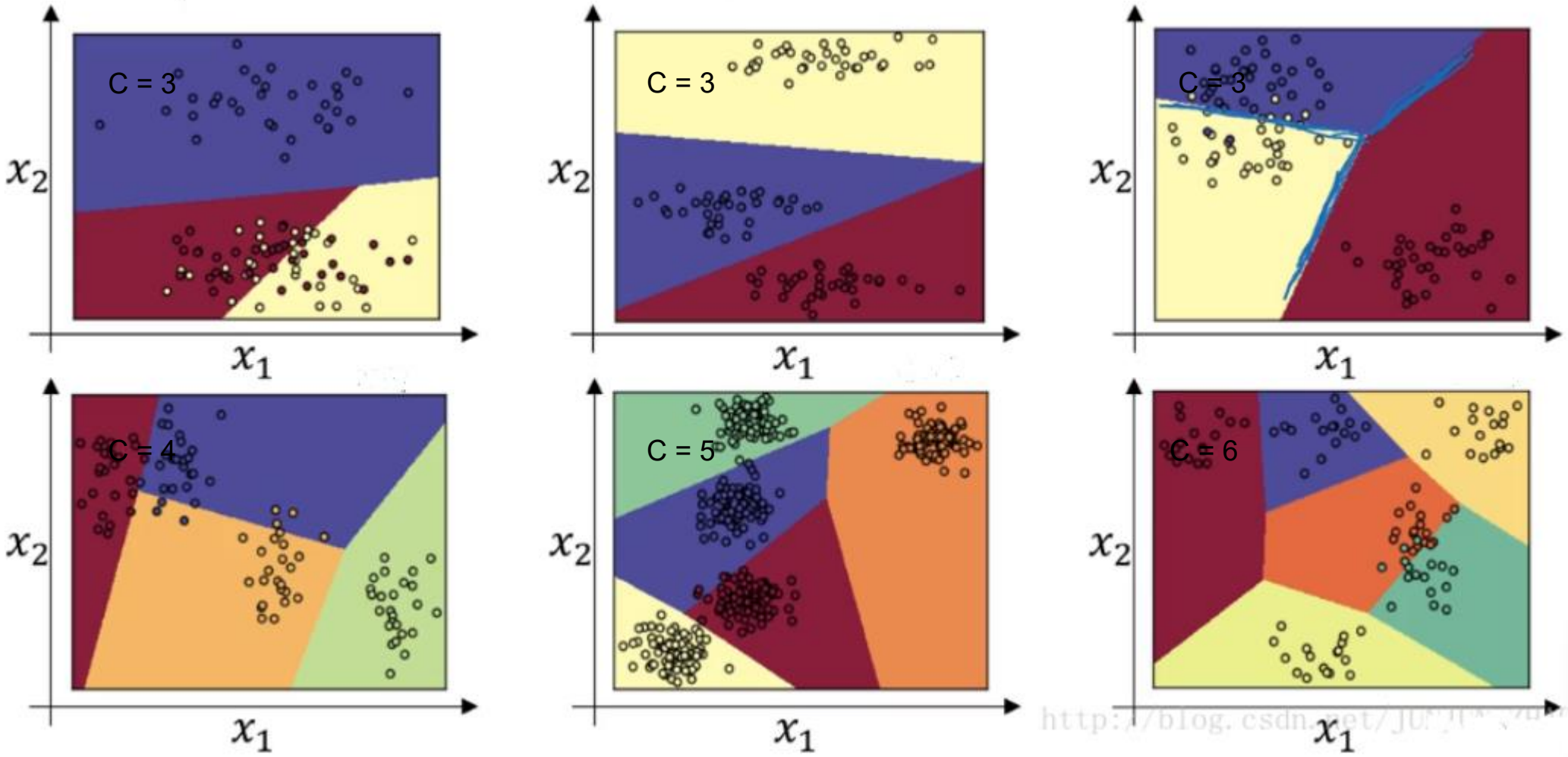
Softmax examples



Softmax examples

- Let's look at the plot some examples with more classes. So it's an example with C equals 4, so that the green class and Softmax can continue to represent these types of linear decision boundaries between multiple classes. So here's one more example with C equals 5 classes, and here's one in the plot above last example with C equals 6.
- So the plot shows the type of things the Softmax classifier can do when there is no hidden layer of class, even much deeper neural network with x and then some hidden units, and then more hidden units, and so on. Then you can learn even more complex non-linear decision boundaries to separate out multiple different classes.

Softmax examples



Multi-class classification

Training a Softmax Classifier

Training a Softmax Classifier

FPT Education
FPT UNIVERSITY

- In binary classification, we have two possible labels, either 0 or 1, and we are trying to recognize one of two classes.
- However, in multiclass classification, we have multiple possible classes, and we want to recognize one of C classes. For example, instead of just recognizing cats, we want to recognize cats, dogs, and baby chicks. In this case, we have four possible classes, including an "other" or "none of the above" class.
- To implement this, we need to build a new neural network with an upper layer that has C units, where C is the number of classes we want to categorize the inputs into.
- The output labels, represented as y hat, will be a C by 1 dimensional vector that outputs the probability of each of the C classes.

Programming Frameworks

Deep Learning frameworks

Deep learning frameworks

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Choosing deep learning frameworks

- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)

Programming Frameworks

TensorFlow

TensorFlow

- TensorFlow is an open-source machine learning library developed by the Google Brain team. It is one of the most popular and widely used frameworks for building and deploying machine learning models.
- TensorFlow provides a comprehensive ecosystem of tools, libraries, and community resources that make it suitable for a range of applications, from research to production deployment.
- TensorFlow is a deep learning programming framework that can help developers be more efficient in implementing and using deep learning algorithms.

TensorFlow

FPT Education
FPT UNIVERSITY

- Key features of TensorFlow include:
- **Flexible Architecture**: TensorFlow allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. It supports both CPU and GPU acceleration, making it suitable for a wide range of hardware configurations.
- **High-Level APIs**: TensorFlow provides high-level APIs like Keras, which simplifies the process of building and training neural networks. Keras is now integrated as the official high-level API of TensorFlow, making it user-friendly and accessible to beginners while maintaining the flexibility needed by researchers and experts.
- **Scalability**: TensorFlow is designed to scale from single machines to large clusters of GPUs and TPUs (Tensor Processing Units). This makes it suitable for both small-scale and large-scale machine learning applications.
- **Community and Documen**tation: TensorFlow has a large and active community of developers, researchers, and practitioners. This means there are plenty of resources, tutorials, and documentation available for users at all levels of expertise.

TensorFlow

- **TensorBoard**: TensorFlow comes with a visualization tool called TensorBoard, which allows you to visualize and debug your machine learning models. You can monitor metrics, visualize the graph of your model, and explore the learning process in real-time.
- **Support for Various Platforms**: TensorFlow supports a variety of platforms, including desktops, servers, and mobile devices. This versatility makes it easy to deploy models across different environments.
- **Support for Different Types of Models**: While TensorFlow is widely used for deep learning, it also supports other machine learning paradigms such as reinforcement learning, natural language processing, and more.
- **Extensibility**: TensorFlow is highly extensible and allows users to add custom functionality through the use of extensions and custom operators.

TensorFlow

- Lets see how to implement a minimization function:
- $J(w) = w^2 - 10w + 25 = (w - 5)^2$
- (cost)
- The result should be $w = 5$ as the function is $(w - 5)^2 = 0$
-

TensorFlow

FPT Education
FPT UNIVERSITY

- Code version 1:

```
import numpy as np
import tensorflow as tf
w = tf.Variable(0, dtype=tf.float32)  # creating a variable w
cost = tf.add(tf.add(w**2, tf.multiply(-10.0, w)), 25.0) # can be written as this
                                                         # cost = w**2 - 10*w + 25

train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
session.run(w)    # Runs the definition of w, if you print this it will print zero
session.run(train)
print("W after one iteration:", session.run(w))
for i in range(1000):
       session.run(train)
print("W after 1000 iterations:", session.run(w))
```

TensorFlow

FPT Education
FPT UNIVERSITY

- Code version 2:

```
import numpy as np
import tensorflow as tf
coefficients = np.array([[1.], [-10.], [25.]])
x = tf.placeholder(tf.float32, [3, 1])
w = tf.Variable(0, dtype=tf.float32)   # Creating a variable w
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
session.run(w)    # Runs the definition of w, if you print this it will print zero
session.run(train, feed_dict={x: coefficients})
print("W after one iteration:", session.run(w))
for i in range(1000):
       session.run(train, feed_dict={x: coefficients})
print("W after 1000 iterations:", session.run(w))
```

TensorFlow

- In TensorFlow you implement only the forward propagation and TensorFlow will do the backpropagation by itself.
- In TensorFlow a placeholder is a variable you can assign a value to later.
- If you are using a mini-batch training you should change the `feed_dict={x: coefficients}` to the current mini-batch data.
- Almost all TensorFlow programs use this:

```
tf.Session() as session:# better for cleaning up in
                        # case of error/exception
session.run(init)
session.run(w)
```

- In deep learning frameworks there are a lot of things that you can do with one line of code like changing the optimizer.

TensorFlow

- In deep learning frameworks, tasks like changing the optimizer can often be achieved with a single line of code.
- TensorFlow programming involves creating Tensors, performing operations on them, initializing, creating a session, and running it to execute the operations.

TensorFlow

- We can use this line to compute the cost function:
  `tf.nn.sigmoid_cross_entropy_with_logits(logits = ... , labels = ...)`
- To initialize weights in NN using TensorFlow use:
  ```
  W1 = tf.get_variable("W1", [25,12288], initializer = tf.contrib.layers.xavier_initializer(seed = 1))
  b1 = tf.get_variable("b1", [25,1], initializer = tf.zeros_initializer())
  ```
- For 3-layer NN, it is important to note that the forward propagation stops at $Z^{[3]}$. The reason is that in TensorFlow the last linear layer output is given as input to the function computing the loss. Therefore, you don't need $A^{[3]}$!
- To reset the graph use `tf.reset_default_graph()`

Summarization

- Tuning involves iteratively adjusting hyperparameters and model aspects for optimal performance.
- Hyperparameters should be explored on a logarithmic scale for efficient tuning. "Pandas" (random search) and "Caviar" (Bayesian optimization) are compared for hyperparameter tuning, with Caviar often yielding better results.
- Activation normalization stabilizes learning by ensuring consistent distributions of neuron activations.
- Batch Normalization (Batch Norm) normalizes layer inputs, enhancing training stability, convergence, and generalization. Batch Norm addresses internal covariate shift, allowing higher learning rates.
- Softmax regression is used for multiclass classification, assigning probabilities to each class.
- Deep learning frameworks like TensorFlow provide tools for building, training, and deploying neural networks. TensorFlow, developed by Google, offers a flexible ecosystem for various neural network architectures.

Question

FPT Education
FPT UNIVERSITY

1. How is Batch Norm applied at test time?
2. Name three key features provided by modern deep learning frameworks.
3. Why is random search often better than grid search for high-dimensional hyperparameter tuning?
4. Are all hyperparameters equally important to tune? Why or why not?
5. What does the choice between Panda and Caviar search strategies depend on?
6. How do you sample $\beta \in [0.9, 0.99$ uniformly in log space?
7. Why shouldn't you tune hyperparameters only once at the start?
8. What exactly is normalized in the lth layer during Batch Norm?
9. Why is ε added in the Batch Norm formula?
10. What is the role of γ and β in Batch Normalization?