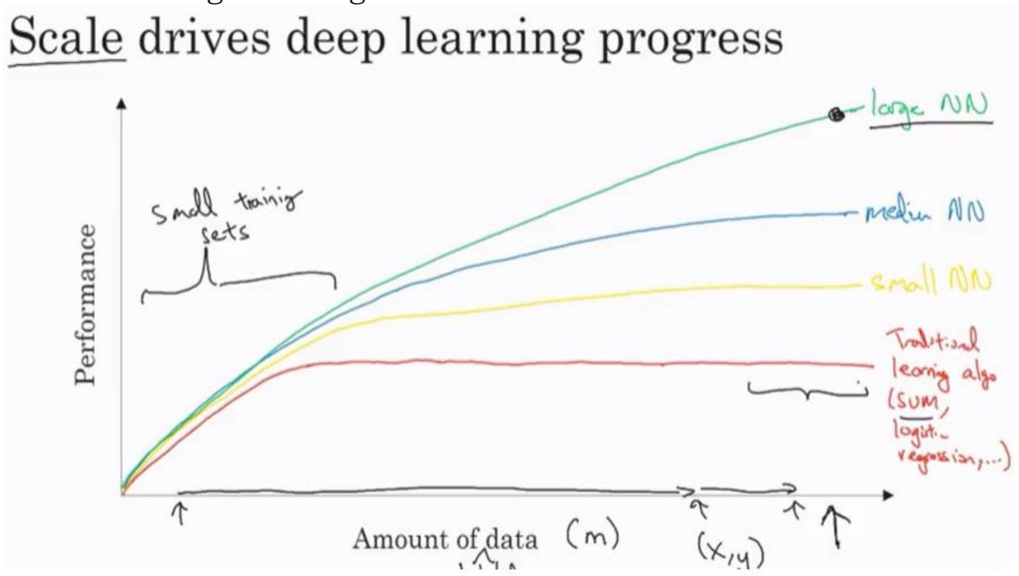
Neural Networks/Hyperparameters optimization

Neural networks are multi-layer networks of neurons that we use to classify things, make predictions, etc.

Why is deep learning taking off?

- Deep learning is taking off for 3 reasons:

  i. Data:
    - Using this image we can conclude:



    - For small data NN can perform as Linear regression or SVM (Support vector machine)
    - For big data a small NN is better that SVM
    - For big data a big NN is better that a medium NN is better that small NN.
    - Hopefully we have a lot of data because the world is using the computer a little bit more
      - Mobiles
      - IOT (Internet of things)

  ii. Computation:
    - GPUs.
    - Powerful CPUs.
    - Distributed computing.
    - ASICs

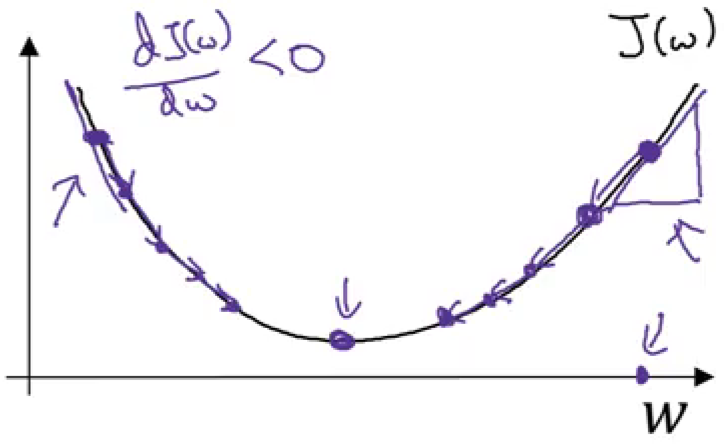
  iii. Algorithm:
    a. Creative algorithms has appeared that changed the way NN works.
      - For example using RELU function is so much better than using SIGMOID function in training a NN because it helps with the vanishing gradient problem.

Gradient Descent

This is a technique that helps to learn the parameters w and b in such a way that the cost function is minimized. The cost function for logistic regression is convex in nature (i.e. only one global minima) and that is the reason for choosing this function instead of the squared error (can have multiple local minima).

Let's look at the steps for gradient descent:

1. Initialize w and b (usually initialized to 0 for logistic regression)
2. Take a step in the steepest downhill direction
3. Repeat step 2 until global optimum is achieved



The updated equation for gradient descent becomes: $\omega := \omega - \alpha \frac{dJ(\omega)}{d\omega}$

Here, α is the learning rate that controls how big a step we should take after each iteration.

If we are on the right side of the graph shown above, the slope will be positive. Using the updated equation, we will move to the left (i.e. downward direction) until the global minima is reached. Whereas if we are on the left side, the slope will be negative and hence we will take a step towards the right (downward direction) until the global minima is reached. Pretty intuitive, right?

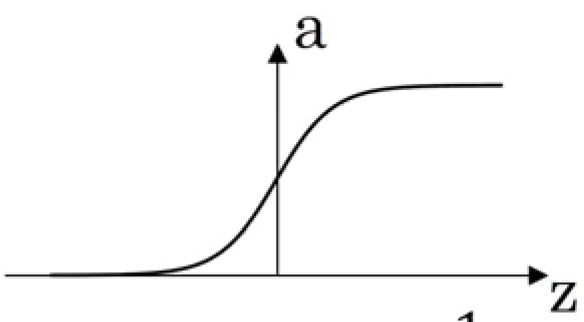
The updated equations for the parameters of logistic regression are:

$$\omega := \omega - \alpha \frac{dJ(\omega, b)}{d\omega}$$

$$b := b - \alpha \frac{dJ(\omega, b)}{db}$$

Activation Function

While calculating the output, an activation function is applied. The choice of an activation function highly affects the performance of the model. So far, we have used the sigmoid activation function:

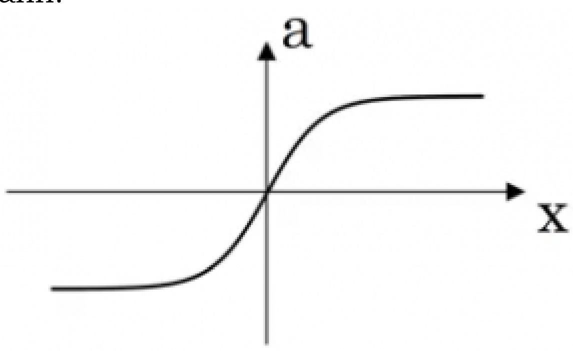


$$\text{sigmoid:}\quad a = \frac{1}{1+e^{-z}}$$

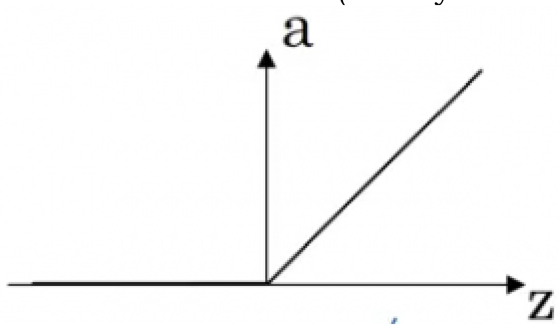
However, this might not the best option in some cases. Why? Because at the extreme ends of the graph, the derivative will be close to zero and hence the gradient descent will update the parameters very slowly.

There are other functions which can replace this activation function:

- tanh:



- ReLU (already covered earlier):



Activation Function	Pros	Cons

Sigmoid	Used in the output layer for binary classification	Output ranges from 0 to 1
tanh	Better than sigmoid	Updates parameters slowly when points are at extreme ends
ReLU	Updates parameters faster as slope is 1 when x>0	Zero slope when x<0

We can choose different activation functions depending on the problem we're trying to solve.

Why do we need non-linear activation functions?

If we use linear activation functions on the output of the layers, it will compute the output as a linear function of input features. We first calculate the Z value as:

$Z = WX + b$

In case of linear activation functions, the output will be equal to Z (instead of calculating any non-linear activation):

$A = Z$

Using linear activation is essentially pointless. The composition of two linear functions is itself a linear function, and unless we use some non-linear activations, we are not computing more interesting functions. That's why most experts stick to using non-linear activation functions.

There is only one scenario where we tend to use a linear activation function. Suppose we want to predict the price of a house (which can be any positive real number). If we use a sigmoid or tanh function, the output will range from (0,1) and (-1,1) respectively. But the price will be more than 1 as well. In this case, we will use a linear activation function at the output layer.

Once we have the outputs, what's the next step? We want to perform backpropagation in order to update the parameters using gradient descent.

Derivatives of activation functions

- Derivation of Sigmoid activation function:

```
g(z) = 1 / (1 + np.exp(-z))
g'(z) = (1 / (1 + np.exp(-z))) * (1 - (1 / (1 + np.exp(-z))))
g'(z) = g(z) * (1 - g(z))
```

- Derivation of Tanh activation function:

```
g(z)  = (e^z - e^-z) / (e^z + e^-z)
```

```
g'(z) = 1 - np.tanh(z)^2 = 1 - g(z)^2
```

- Derivation of RELU activation function:

```
g(z)  = np.maximum(0,z)
g'(z) = { 0  if z < 0
          1  if z >= 0  }
```

- Derivation of leaky RELU activation function:

```
g(z)  = np.maximum(0.01 * z, z)
g'(z) = { 0.01  if z < 0
          1     if z >= 0   }
```

Gradient Descent on m Examples

- Lets say we have these variables:

```
  X1                                          Feature
  X2                    Feature
  W1                    Weight of the first feature.
  W2                    Weight of the second feature.
  B                     Logistic Regression parameter.
  M                     Number of training examples
  Y(i)                                Expected output of i
```

- So we have:

```
X1  \
W1   \
X2   ===>  z(i) = X1W1 + X2W2 + B ===> a(i) = Sigmoid(z(i)) ===> l(a(i),Y(i)) = - (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))
Y2   /
B   /
```

- Then from right to left we will calculate derivations compared to the result:

```
  d(a)  = d(l)/d(a) = -(y/a) + ((1-y)/(1-a))
  d(z)  = d(l)/d(z) = a - y
  d(W1) = X1 * d(z)
  d(W2) = X2 * d(z)
  d(B) = d(z)
```

- From the above we can conclude the logistic regression pseudo code:

```
  J = 0; dw1 = 0; dw2 =0; db = 0;                 # Devs.
  w1 = 0; w2 = 0; b=0;                                                # 
Weights
  for i = 1 to m
          # Forward pass
          z(i) = W1*x1(i) + W2*x2(i) + b
          a(i) = Sigmoid(z(i))
          J += (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))

          # Backward pass
          dz(i) = a(i) - Y(i)
          dw1 += dz(i) * x1(i)
          dw2 += dz(i) * x2(i)
          db  += dz(i)
  J /= m
  dw1/= m
  dw2/= m
  db/= m

  # Gradient descent
  w1 = w1 - alpa * dw1
  w2 = w2 - alpa * dw2
  b = b - alpa * db
```

- The above code should run for some iterations to minimize error.
- So there will be two inner loops to implement the logistic regression.
- Vectorization is so important on deep learning to reduce loops. In the last code we can make the whole loop in one step using vectorization!

Vectorization

- Deep learning shines when the dataset are big. However for loops will make you wait a lot for a result. Thats why we need vectorization to get rid of some of our for loops.
- NumPy library (dot) function is using vectorization by default.
- The vectorization can be done on CPU or GPU thought the SIMD operation. But its faster on GPU.
- Whenever possible avoid for loops.
- Most of the NumPy library methods are vectorized version.

Vectorizing Logistic Regression

- We will implement Logistic Regression using one for loop then without any for loop.
- As an input we have a matrix X and its [Nx, m] and a matrix Y and its [Ny, m].
- We will then compute at instance [z1,z2...zm] = W' * X + [b,b,...b]. This can be written in python as:

```
  Z = np.dot(W.T,X) + b    # Vectorization, then broadcasting, Z shape is (1, m)
          A = 1 / 1 + np.exp(-Z)   # Vectorization, A shape is (1, m)
```

- Vectorizing Logistic Regression's Gradient Output:

```
          dz = A - Y                  # Vectorization, dz shape is (1, m)
          dw = np.dot(X, dz.T) / m    # Vectorization, dw shape is (Nx, 1)
          db = dz.sum() / m           # Vectorization, dz shape is (1, 1)
```

Notes on Python and NumPy

- In NumPy, obj.sum(axis = 0) sums the columns while obj.sum(axis = 1) sums the rows.
- In NumPy, obj.reshape(1,4) changes the shape of the matrix by broadcasting the values.

- Reshape is cheap in calculations so put it everywhere you're not sure about the calculations.

- Broadcasting works when you do a matrix operation with matrices that doesn't match for the operation, in this case NumPy automatically makes the shapes ready for the operation by broadcasting the values.

- In general principle of broadcasting. If you have an (m,n) matrix and you add(+) or subtract(-) or multiply(*) or divide(/) with a (1,n) matrix, then this will copy it m times into an (m,n) matrix. The same with if you use those operations with a (m , 1) matrix, then this will copy it n times into (m, n) matrix. And then apply the addition, subtraction, and multiplication of division element wise.

- Some tricks to eliminate all the strange bugs in the code:

    - If you didn't specify the shape of a vector, it will take a shape of (m,) and the transpose operation won't work. You have to reshape it to (m, 1)
    - Try to not use the rank one matrix in ANN
    - Don't hesitate to use assert(a.shape == (5,1)) to check if your matrix shape is the required one.
    - If you've found a rank one matrix try to run reshape on it.

- Jupyter / IPython notebooks are so useful library in python that makes it easy to integrate code and document at the same time. It runs in the browser and doesn't need an IDE to run.

    - To open Jupyter Notebook, open the command line and call: jupyter-notebook It should be installed to work.

- To Compute the derivative of Sigmoid:

  ```
  s = sigmoid(x)
  ds = s * (1 - s)       # derivative  using calculus
  ```

  To make an image of (width,height,depth) be a vector, use this:

  ```
  v = image.reshape(image.shape[0]*image.shape[1]*image.shape[2],1)   #reshapes the image.
  ```

- Gradient descent converges faster after normalization of the input matrices.

Question 1: Does the model have high bias?

Solution: We can figure out whether the model has high bias by looking at the training set error. High training error results in high bias. In such cases, we can **try bigger networks**, **train models for a longer period of time**, or **try different neural network architectures**.

Question 2: Does the model have high variance?

Solution : If the dev set error is high, we can say that the model has high variance. To reduce the variance, we can **get more data**, **use regularization,** or **try different neural network architectures**.

Regularization in NN

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

L2 & L1 regularization

L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

Cost function = Loss (say, binary cross entropy) + Regularization term

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

However, this regularization term differs in L1 and L2.

In deep learning, we wish to minimize the following cost function:

$$J\left(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}\right) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Cost function

Where L can be any loss function (such as the cross-entropy loss function). Now, for **L2 regularization** we add a component that will penalize large weights.

Therefore, the equation becomes:

$$J\left(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}\right) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m} \sum_{l=1}^{L} \left\| w^{[L]} \right\|_F^2$$

Here, **lambda** is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as *weight decay* as it forces the weights to decay towards zero (but not exactly zero). Notice the addition of the **Frobenius norm**, denoted by the subscript *F*. This is in fact equivalent to the squared norm of a matrix.

In L1, we have:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

In this, we penalize the absolute value of the weights. Unlike L2, the weights may be reduced to zero here. **Hence, it is very useful when we are trying to compress our model. Otherwise, we usually prefer L2 over it.**

In keras, we can directly apply regularization to any layer using the regularizers.
Below is the sample code to apply L2 regularization to a Dense layer.

```
from keras import regularizers

model.add(Dense(64, input_dim=64,

              kernel_regularizer=regularizers.l2(0.01)
```

Note: Here the value 0.01 is the value of regularization parameter, i.e., lambda, which we need to optimize further. We can optimize it using the grid-search method.

Why regularization works?

As aforementioned, adding the regularization component will drive the values of the weight matrix down. This will effectively decorrelate the neural network.

Recall that we feed the activation function with the following weighted sum:

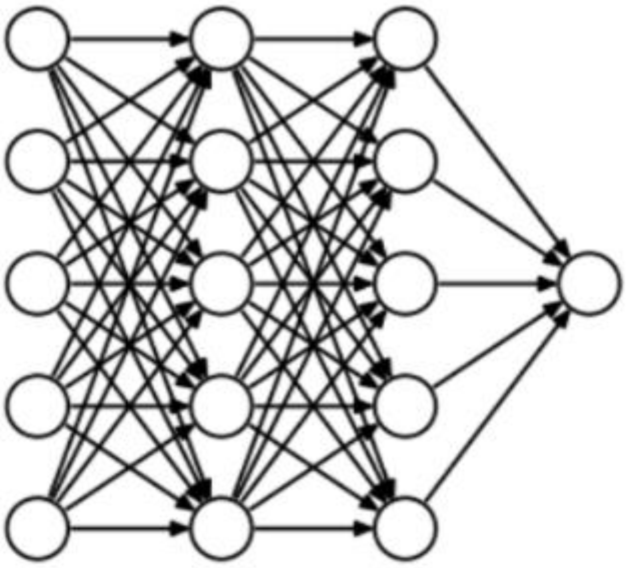
$$z = w^T x + b$$

Weighted sum

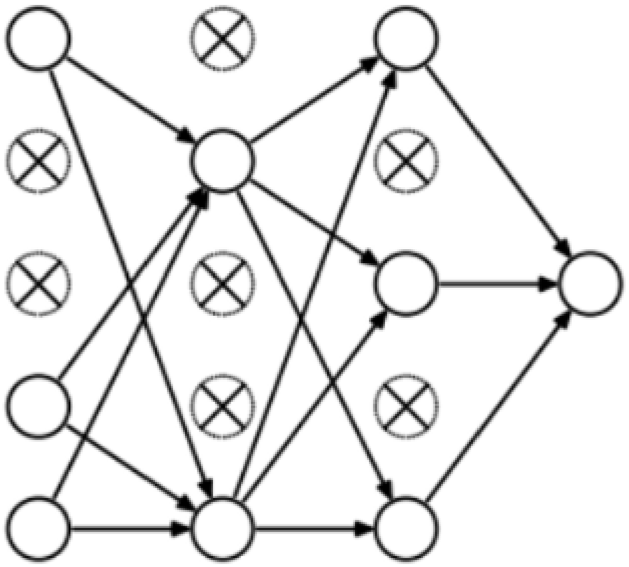
By reducing the values in the weight matrix, *z* will also be reduced, which in turns decreases the effect of the activation function. Therefore, a less complex function will be fit to the data, effectively reducing overfitting.

Dropout

This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.
To understand dropout, let's say our neural network structure is akin to the one shown below:

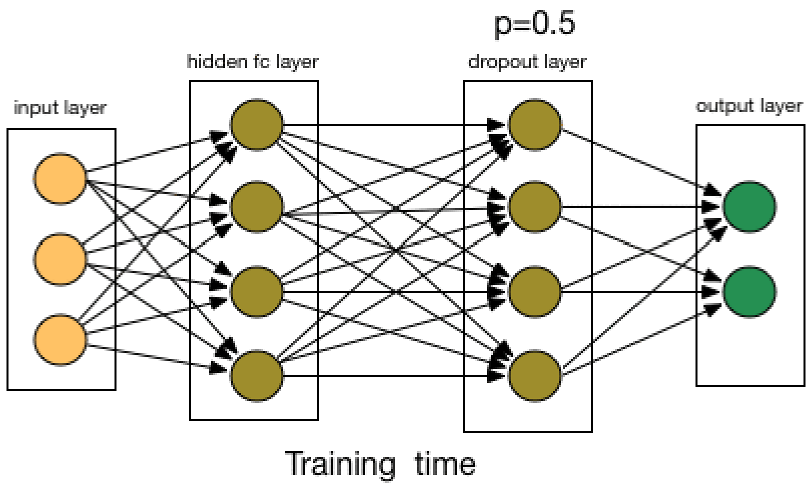


So what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below.



So each iteration has a different set of nodes and this results in a different set of outputs. **It can also be thought of as an ensemble technique in machine learning.**
Ensemble models usually perform better than a single model as they capture more randomness. Similarly, dropout also performs better than a normal neural network model.

This probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function. As seen in the image above, dropout can be applied to both the hidden layers as well as the input layers.



Due to these reasons, dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.
In *keras*, we can implement dropout using the keras core layer. Below is the python code for it:

```
from keras.layers.core import Dropout



model = Sequential([

 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),

 Dropout(0.25),



Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'
),

 ])
```

As you can see, we have defined 0.25 as the probability of dropping. We can tune it further for better results using the grid search method.

Why dropout works?

It might seem to crazy to randomly remove nodes from a neural network to regularize it. Yet, it is a widely used method and it was proven to greatly improve the performance of neural networks. So, why does it work so well?

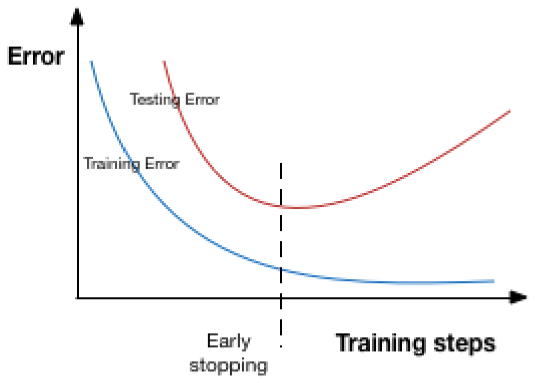
Dropout means that the neural network cannot rely on any input node, since each have a random probability of being removed. Therefore, the neural network will be reluctant to give high weights to certain features, because they might disappear.

Consequently, the weights are spread across all features, making them smaller. This effectively shrinks the model and regularizes it.

Link
https://towardsdatascience.com/how-to-improve-a-neural-network-with-regularization-8a18ecda9fe3
https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

Early stopping

Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model. This is known as early stopping.



In the above image, we will stop training at the dotted line since after that our model will start overfitting on the training data.
In *keras*, we can apply early stopping using the callbacks function. Below is the sample code for it.

```
from keras.callbacks import EarlyStopping



EarlyStopping(monitor='val_err', patience=5)
```

Here, monitor denotes the quantity that needs to be monitored and '**val_err**' denotes the validation error.

Patience denotes the number of epochs with no further improvement after which the training will be stopped. For better understanding, let's take a look at the above image again. After the dotted line, each epoch will result in a higher value of validation error. Therefore, 5 epochs after the dotted line (since our patience is equal to 5), our model will stop because no further improvement is seen.

Note: It may be possible that after 5 epochs (this is the value defined for **patience** *in general), the model starts improving again and the validation error starts decreasing as well. Therefore, we need to take extra care while tuning this hyperparameter.*

Links

https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/

https://machinelearningmastery.com/weight-regularization-to-reduce-overfitting-of-deep-learning-models/

https://www.analyticsvidhya.com/blog/2018/11/neural-networks-hyperparameter-tuning-regularization-deeplearning/

ML Strategy

Orthogonalization

Single evaluation metric

Use one single metric that will optimize learning as well as produce faster results

Optimizing metric

Model must have accuracy(optimising metric) as well as running time constraint, False positive(satisfying metric)

Test/dev SET

Test and dev test should come from same distribution.

SIZE of TEST/DEV

Use 98 perc for train and 1/1 for test/dev for dataset with 1 million examples

Change DEV/TEST set

In other words, if we discover that your dev test set has these very high quality images but evaluating on this dev test set is not predictive of how well your app actually performs, because your app needs to deal with lower quality images,
then that's a good time to change your dev test set so that your data better reflects the type of data you actually need to do well on.

But the overall guideline is if your current metric and data you are
evaluating on doesn't correspond to doing well on what you actually care about,
then change your metrics and/or your dev/test set to
better capture what you need your algorithm to actually do well on.

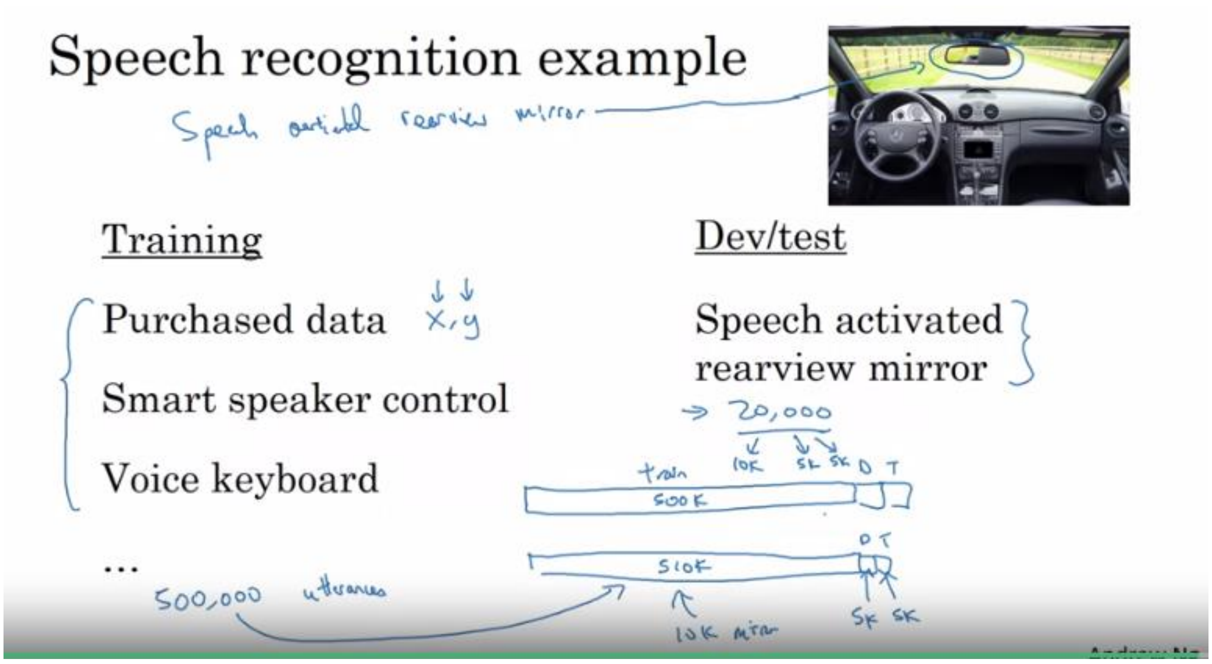
ERROR ANALYSIS

Often we have to tackle error manually and this is case when we have a small error percentage, like 10 perc of a small dataset or 2 perc of large dataset.

When can then manually verify what things are incorrectly classified or wrong and make a understanding of what next can be done.

Often in case of any new model, try to use any algo to create a sample model and reiterate over process if it doesn't turn out well. Basically don't overthink on how to proceed.

Mismatched Train/Dev



Speech recognition example

Training

Purchased data

Smart speaker control

Voice keyboard

...

Dev/test

Speech activated rearview mirror

Link

https://github.com/mbadry1/DeepLearning.ai-Summary/tree/master/1-%20Neural%20Networks%20and%20Deep%20Learning#derivatives

https://github.com/mbadry1/DeepLearning.ai-Summary