Deep Learning Notes

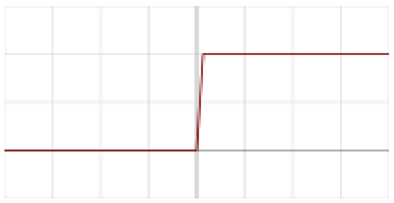
Haotian Chen

Contents

1 Neural Networks and Deep Learning

1.1 Activation Function

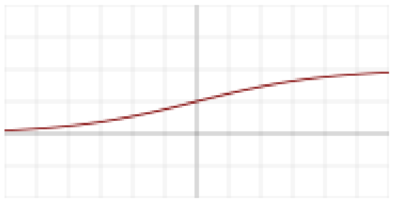
binary step:



$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}, f(x) \in \{0, 1\}$$

$$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

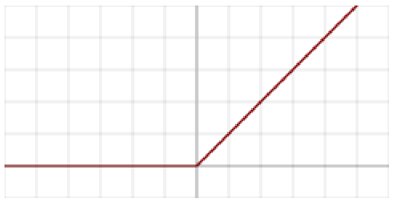
sigmoid function: (mainly used in output layer)



$$f(x) = \sigma(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}, f(x) \in (0, 1)$$

$$f'(x) = f(x)(1 - f(x))$$

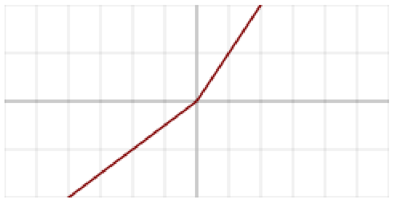
rectified linear unit: (relu, mainly used in hidden layer)



$$f(x) = max\{0, x\} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}, f(x) \in [0, +\infty)$$

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

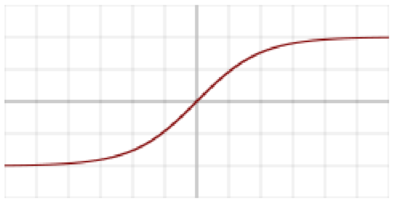
Leaky rectified linear unit: (leaky relu, mainly used in hidden layer)



$$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}, f(x) \in (-\infty, +\infty)$$

$$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

hyperbolic tangent: (tanh, mainly used in hidden layer)



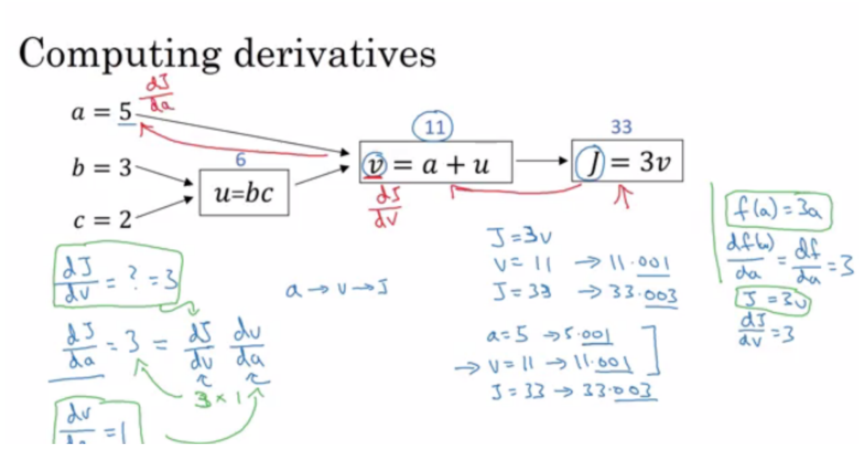
$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, f(x) \in (-1, 1)$$

$$f'(x) = 1 - f(x)^2$$

It turns out that the tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer. Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the vanishing gradient problem.

Linear activation function will output linear activations. No matter how many hidden layers you add, the activation will be always linear like logistic regression (So its useless in a lot of complex problems). You might use linear activation function in the output layer if the output is real numbers (regression problem).

1.2 Computation Graphs of Derivatives:



apply chain rule:

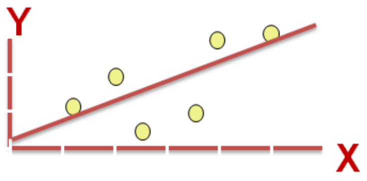
$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial a} = 3 \times 1 = 3$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial b} = 3 \times 1 \times 2 = 6$$

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial c} = 3 \times 1 \times 3 = 9$$

1.3 Binary Classification

Use logistic regression to build a binary classifier.



training data:

$$x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$$

$$m \text{ training examples: } (x^{(i)}, y^{(i)})\ for\ i = 1, \ldots, m$$

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix}, y^{(i)} = y_1^{(i)}$$

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix}, Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix}$$

$$X \in \mathbb{R}^{n_x \times m}, Y \in \mathbb{R}^{1 \times m}$$

1.4 Logistic Regression

Logistic regression is a statistical model that uses a logistic function to model a binary dependent variable.

Given x, want $\hat{y} = P(y = 1|x)$, where $0 \le \hat{y} \le 1$, $x \in \mathbb{R}^{n_x}$, $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

$$z = w^T x + b$$
$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$P(y|x) = \begin{cases} \hat{y} & \text{if } y = 1 \\ 1 - \hat{y} & \text{if } y = 0 \end{cases} = \hat{y}^y (1 - \hat{y})^{(1-y)}$$

We want to maximize $P(y|x)$. To make it simpler, because *log* function is a strictly increasing function, we can maximize $log(P(y|x))$ instead.

$$log(P(y|x)) = log(\hat{y}^y (1 - \hat{y})^{(1-y)}) = ylog(\hat{y}) + (1 - y)log(1 - \hat{y})$$

Or in reverse, we can minimize $-log(P(y|x))$, which is called **loss function**.
loss function: (convex)

$$L(\hat{y}, y) = -(ylog(\hat{y}) + (1 - y)log(1 - \hat{y}))$$

cost function: (convex)

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)})]$$

gradient descent: with learning rate α, find best w, b to minimize $J(w, b)$
repeat until convergence {

$$w_j =: w_j - \alpha \frac{\partial}{\partial w_j} J(w, b) = w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) x_j^{(i)}$$

$$b =: b - \alpha \frac{\partial}{\partial b} J(w, b) = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})$$

simultaneously update w_j and b, $j \in [1, n]$

}

vectorized implementation:
repeat until convergence {

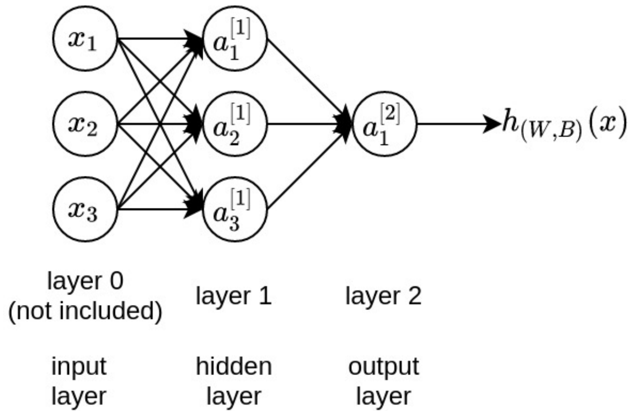
$$w =: w - \alpha\frac{1}{m}X(\sigma(w^TX + b) - Y)^T$$

$$b =: b - \alpha\frac{1}{m}sum\{\sigma(w^TX + b) - Y\}$$

}

1.5 Neural Network

Basic Structure:



$$a_i^{[j]} = \text{"activation" of unit i in layer j}$$

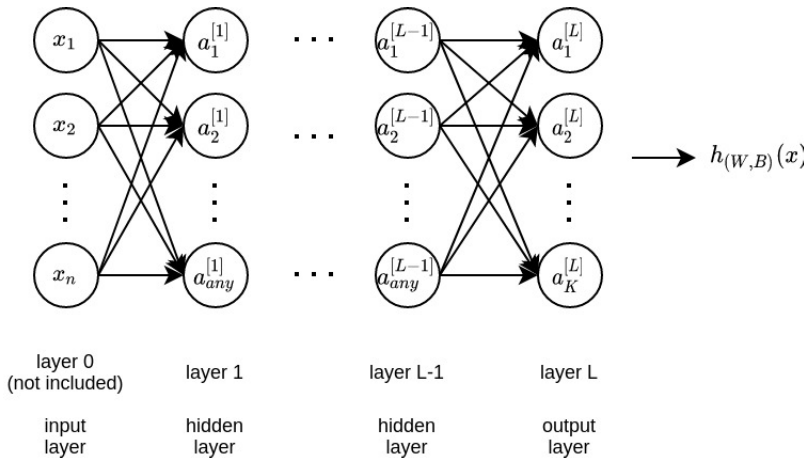
$$W^{[j]} = \text{matrix of weights (edges) from layer j - 1 to j}$$

$$B^{[j]} = \text{vector of biases (nodes) from layer j - 1 to j}$$

$$a_1^{[1]} = \sigma(W_{11}^{[1]}x_1 + W_{12}^{[1]}x_2 + W_{13}^{[1]}x_3 + B_1^{[1]})$$
$$a_2^{[1]} = \sigma(W_{21}^{[1]}x_1 + W_{22}^{[1]}x_2 + W_{23}^{[1]}x_3 + B_2^{[1]})$$
$$a_3^{[1]} = \sigma(W_{31}^{[1]}x_1 + W_{32}^{[1]}x_2 + W_{33}^{[1]}x_3 + B_3^{[1]})$$
$$h_{(W,B)}(x) = a_1^{[2]} = \sigma(W_{11}^{[2]}a_1^{[1]} + W_{12}^{[2]}a_2^{[1]} + W_{13}^{[2]}a_3^{[1]} + B_1^{[2]})$$

If network has s_j units in layer j, s_{j-1} units in layer $j - 1$, then $W^{[j]}$ will be of dimension $s_j \times s_{j-1}$, $B^{[j]}$ will be of dimension $s_j \times 1$.

Generalized Model (one vs all):



For a neural network that has:

$$L = \text{total number of layers in the network}$$
$$s_l = \text{number of units in layer } l$$
$$K = \text{number of output units/classes}$$

assume $a^{[0]} = x, a^{[L]} = h_{(W,B)}(x)$, let:

$$z^{[l]} = W^{[l]}a^{[l-1]} + B^{[l]}$$
$$a^{[l]} = \sigma(z^{[l]})$$
$$h_{(W,B)}(x) = a^{[L]} = \sigma(z^{[L]})$$

regularized cost function:

$$J(W,B) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)}log(h_{(W,B)}(x^{(i)})_k)+(1-y_k^{(i)})log(1-h_{(W,B)}(x^{(i)})_k)]+\frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{j,i}^{[l]})^2$$

To reduce over-fitting, we can reduce(penalize) the weight of the features in our function carry by increasing their cost. The λ is the regularization parameter. It determines how much the costs of our theta parameters are inflated.

vectorized implementation: (with different activation function for each layer)

$$Z^{[l]} = W^{[l]}A^{[l-1]} + B^{[l]}$$
$$A^{[l]} = \sigma^{[l]}(Z^{[l]})$$
$$h_{(W,B)}(X) = A^{[L]} = \sigma^{[L]}(Z^{[L]})$$

$$J(W,B) = -\frac{1}{m}np.sum(Y\odot log(h_{(W,B)}(X))+(1-Y)\odot log(1-h_{(W,B)}(X)))+\frac{\lambda}{2m}np.sum(W\odot W)$$

1.6 Backpropagation Preliminary

matrix calculus: see Wikipedia
chaine rule:
Suppose the variable J depends on the variables $w_1, \dots, w_p$ via the intermediate variable $z_1, \dots, z_k$.

$$z_j = z_j(w_1, \dots, w_p), \forall j \in \{1, \dots, k\}$$
$$J = J(z_1, \dots, z_k)$$

Expand J, we can find:

$$\frac{\partial J}{\partial w_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial z_j}\frac{\partial z_j}{\partial w_i}, \forall i \in \{1, \dots, p\}$$

chain rule derivation for matrix:
Suppose J is a real-valued output variable, $z \in \mathbb{R}^m$ is the intermediate variable and $W \in \mathbb{R}^{m\times d}$, $B \in \mathbb{R}^m$, $a \in \mathbb{R}^d$ are the input variables. Suppose they satisfy:

$$z = Wa + B$$
$$J = J(z)$$

Then we can get:

$$\frac{\partial J}{\partial a} = \begin{bmatrix} \frac{\partial J}{\partial a_1} \\ \vdots \\ \frac{\partial J}{\partial a_d} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^m \frac{\partial J}{\partial z_j}\frac{z_j}{a_1} \\ \vdots \\ \sum_{j=1}^m \frac{\partial J}{\partial z_j}\frac{z_j}{a_d} \end{bmatrix} = \begin{bmatrix} \frac{\partial z_1}{\partial a_1} & \dots & \frac{\partial z_m}{\partial a_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial a_d} & \dots & \frac{\partial z_m}{\partial a_d} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} = \frac{\partial z}{\partial a}\frac{\partial J}{\partial z} = W^T \frac{\partial J}{\partial z}$$

$$\frac{\partial J}{\partial W_{ij}} = \sum_{k=1}^{m} \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial W_{ij}} = \frac{\partial J}{\partial z_i}\frac{\partial z_i}{\partial W_{ij}} = \frac{\partial J}{\partial z_i} a_j$$

$$\frac{\partial J}{\partial B_i} = \sum_{k=1}^{m} \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial B_i} = \frac{\partial J}{\partial z_i}\frac{\partial z_i}{\partial B_i} = \frac{\partial J}{\partial z_i}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} \begin{bmatrix} a_1 & \dots & a_d \end{bmatrix} = \frac{\partial J}{\partial z} a^T$$

$$\frac{\partial J}{\partial B} = \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} = \frac{\partial J}{\partial z}$$

element-wise chain rule:
Assume $z, a \in \mathbb{R}^d$:

$$a = \sigma(z), \text{ where } \sigma \text{ is an element-wise activation}$$
$$J = J(a)$$

Then we have:

$$\frac{\partial J}{\partial z} = \frac{\partial J}{\partial a} \odot \sigma'(z)$$

Where σ' is the element-wise derivative of the activation function σ.

1.7 Backpropagation

To train the model, we need to update W and B for each epoch: (gradient decent)

$$W := W - \alpha \frac{\partial}{\partial W} J(W, B)$$
$$B := B - \alpha \frac{\partial}{\partial B} J(W, B)$$

We can see $\frac{\partial}{\partial W} J(W, B)$ and $\frac{\partial}{\partial B} J(W, B)$ are hard to get directly. There is an easier way to calculate it. For each training example $(x^{(q)}, y^{(q)}), q \in \{1, \ldots, m\}$, define **loss function**:

$$J = -\sum_{k=1}^{K} [y_k^{(q)} log(h_{(W,B)}(x^{(q)})_k) + (1 - y_k^{(q)}) log(1 - h_{(W,B)}(x^{(q)})_k)]$$

Apply chain rule we have:

$$\frac{\partial J}{\partial W^{[l]}} = \frac{\partial J}{\partial z^{[l]}} (a^{[l-1]})^T$$
$$\frac{\partial J}{\partial B^{[l]}} = \frac{\partial J}{\partial z^{[l]}}$$
$$\frac{\partial J}{\partial a^{[l]}} = (W^{[l+1]})^T \frac{\partial J}{\partial z^{[l+1]}}$$
$$\begin{aligned} \frac{\partial J}{\partial z^{[l]}} &= \frac{\partial J}{\partial a^{[l]}} \odot \sigma'(z^{[l]}) \\ &= (W^{[l+1]})^T \frac{\partial J}{\partial z^{[l+1]}} \odot \sigma'(z^{[l]}) \\ &= (W^{[l+1]})^T \frac{\partial J}{\partial z^{[l+1]}} \odot (a^{[l]} \odot (1 - a^{[l]})) \end{aligned}$$

For $p \in \{1, ..., K\}$:

$$\begin{aligned}
\frac{\partial J}{\partial z_p^{[L]}} &= \frac{\partial}{\partial z_p^{[L]}} \sum_{k=1}^{K} -[y_k^{(q)} log(h_{(W,B)}(x^{(q)})_k) + (1 - y_k^{(q)}) log(1 - h_{(W,B)}(x^{(q)})_k)] \\
&= \frac{\partial}{\partial z_p^{[L]}} \{-[y_p^{(q)} log(\frac{1}{1 + e^{-z_p^{[L]}}}) + (1 - y_p^{(q)}) log(1 - \frac{1}{1 + e^{-z_p^{[L]}}})]\} \\
&= -[y_p^{(q)} (1 + e^{-z_p^{[L]}}) \frac{0 - (-e^{-z_p^{[L]}})}{(1 + e^{-z_p^{[L]}})^2} + (1 - y_p^{(q)}) \frac{1 + e^{-z_p^{[L]}}}{e^{-z_p^{[L]}}} \frac{(-e^{-z_p^{[L]}})(1 + e^{-z_p^{[L]}}) - e^{-z_p^{[L]}}(-e^{-z_p^{[L]}})}{(1 + e^{-z_p^{[L]}})^2}] \\
&= -[y_p^{(q)} \frac{e^{-z_p^{[L]}}}{1 + e^{-z_p^{[L]}}} + (1 - y_p^{(q)}) \frac{-1}{1 + e^{-z_p^{[L]}}}] \\
&= -\frac{y_p^{(q)} e^{-z_p^{[L]}} + y_p^{(q)} - 1}{1 + e^{-z_p^{[L]}}} \\
&= \frac{1}{1 + e^{-z_p^{[L]}}} - y_p^{(q)} \\
&= a_p^{[L]} - y_p^{(q)}
\end{aligned}$$

Then we get:

$$\frac{\partial J}{\partial z^{[L]}} = \begin{bmatrix} \frac{\partial J}{\partial z_1^{[L]}} \\ \vdots \\ \frac{\partial J}{\partial z_K^{[L]}} \end{bmatrix} = a^{[L]} - y^{(q)}$$

For convenience, define **error term**:

$$\delta^{[l]} = \frac{\partial J}{\partial z^{[l]}}$$

Then we get:

$$\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T$$

$$\frac{\partial J}{\partial B^{[l]}} = \delta^{[l]}$$

$$\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \odot (a^{[l]} \odot (1 - a^{[l]}))$$

$$\delta^{[L]} = a^{[L]} - y^{(q)}$$

```
backpropagation algorithm: (compute $\frac{\partial}{\partial W} J(W,B)$, $\frac{\partial}{\partial B} J(W,B)$)
training set: $(x^{(q)}, y^{(q)}), q \in \{1, \ldots, m\}$
set $\Delta(W)_{ij}^{[l]} = 0, \Delta(B)_i^{[l]} = 0, l \in \{1, \ldots, L\}$
for $q \in \{1, \ldots, m\}$:
    forward propagation: compute $a^{[l]}$ for $l \in \{1, \ldots, L\}$
    compute $\delta^{[L]} = \frac{\partial J}{\partial z^{[L]}} = a^{[L]} - y^{(q)}$
    for $l \in \{L-1, \ldots, 1\}$:
        compute $\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \odot \sigma'(z^{[l]}) = (W^{[l+1]})^T \delta^{[l+1]} \odot (a^{[l]} \odot (1 - a^{[l]}))$
    for $l \in \{1, \ldots, L\}$:
        compute $\Delta(W)^{[l]} := \Delta(W)^{[l]} + \frac{\partial J}{\partial W^{[l]}} = \Delta(W)^{[l]} + \delta^{[l]} (a^{[l-1]})^T$
        compute $\Delta(B)^{[l]} := \Delta(B)^{[l]} + \frac{\partial J}{\partial B^{[l]}} = \Delta(B)^{[l]} + \delta^{[l]}$
compute $\frac{\partial}{\partial W_{ij}^{[l]}} J(W,B) = D(W)_{ij}^{[l]} = \frac{1}{m}\Delta(W)_{ij}^{[l]} + \frac{\lambda}{m} W_{ij}^{[l]}$
compute $\frac{\partial}{\partial B_{i}^{[l]}} J(W,B) = D(B)_{i}^{[l]} = \frac{1}{m}\Delta(B)_{i}^{[l]}$
```

vectorized backpropagation algorithm: (compute $\frac{\partial}{\partial W} J(W,B)$, $\frac{\partial}{\partial B} J(W,B)$) with different activation function for each layer:

$$\partial Z^{[L]} = \frac{\partial}{\partial Z^{[L]}} J(W,B) = A^{[L]} - Y$$

$$\partial Z^{[l]} = \frac{\partial}{\partial Z^{[l]}} J(W,B) = (W^{[l+1]})^T \partial Z^{[l+1]} \odot \sigma'^{[l]}(Z^{[l]})$$

$$\partial W^{[l]} = \frac{\partial}{\partial W^{[l]}} J(W,B) = \frac{1}{m} \partial Z^{[l]} (A^{[l-1]})^T + \frac{\lambda}{m} W^{[l]}$$

$$\partial B^{[l]} = \frac{\partial}{\partial B^{[l]}} J(W,B) = \frac{1}{m} np.sum(\partial Z^{[l]}, axis = 1, keepdims = True)$$

1.8 Random Initialization (symmetry breaking)

Initializing bias matrices B with zero is OK. Initializing all the weight matrices W with zero does not work with neural networks, because all hidden units will be completely identical (symmetric) - compute exactly the same function. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our W matrices using the following method:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * 0.01$$

1.9 Parameters vs Hyperparameters

Main parameters of the Neural Network are W and B.
Hyper parameters (parameters that used to find best main parameters) are like:

- Learning rate.
- Number of iteration.
- Number of hidden layers.
- Number of hidden units.
- Choice of activation functions.

- Momentum term.
- Mini-batch size.
- Regularization parameters.

2 Hyperparameter Tuning, Regularization and Optimization

2.1 Train/Dev/Test Sets

find better hyperparameters:
repeat: Idea $\longrightarrow$ Code $\longrightarrow$ Experiment

split dataset:

- Training set
- Validation set / Development or ”dev” set.
- Testing set

ratio of splitting:

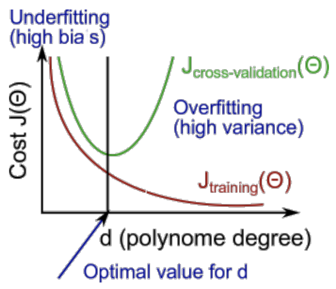
- If size of the dataset is 100 to 1000000: 60%/20%/20%
- If size of the dataset is 1000000 to INF: 98%/1%/1% or 99.5%/0.25%/0.25%

Make sure the dev and test set are coming from the same distribution. For example if cat training pictures is from the web and the dev/test pictures are from users cell phone they will mismatch. It is better to make sure that dev and test set are from the same distribution.

The dev set rule is to try them on some of the good models you've created. Its OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a dev set as its used in the development.

2.2 Bias/Variance

The training error will tend to decrease as we increase the degree d of the polynomial. At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d is increased, forming a convex curve.



High bias(underfitting): both $J_{train}(W, B)$ and $J_{cv}(W, B)$ will be high. Also, $J_{cv}(W, B) \approx J_{train}(W, B)$.
High variance(overfitting): $J_{train}(W, B)$ will be low and $J_{cv}(W, B)$ will be high. Also, $J_{cv}(W, B) \gg J_{train}(W, B)$.

underfitting:
High bias, which means hypothesis fits training data poorly, is usually caused by a function that is too simple or using too few features.

overfitting:
High variance, which means hypothesis fits training data well, but does not generalize well to predict new data. It is usually caused by a complicated function with too many features.

2.3 Basic Recipe for Machine Learning

If your algorithm has a high bias:

- Try to make your NN bigger (size of hidden units, number of layers)
- Try a different model (architecture) that is suitable for your data.
- Try to run it longer.
- Different (advanced) optimization algorithms.

If your algorithm has a high variance:

- More data.
- Try regularization.
- Try a different model (architecture) that is suitable for your data.

You should try the previous two points until you have a low bias and low variance. In the older days before deep learning, there was a "Bias/variance tradeoff". But because now you have more options/tools for solving the bias and variance problem its really helpful to use deep learning. With enough training data, training a bigger neural network never hurts.

Regularization: (L2 Regularization)
Add regularization term to the cost function to reduce variance (overfitting):

$$\frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{j,i}^{[l]})^2 = \frac{\lambda}{2m}np.sum(W \odot W)$$

weight backpropagation with regularization:

$$W^{[l]} := W^{[l]} - \alpha\partial W^{[l]} = (1 - \frac{\alpha\lambda}{m})W^{[l]} - \frac{\alpha}{m}\partial Z^{[l]}(A^{[l-1]})^T$$

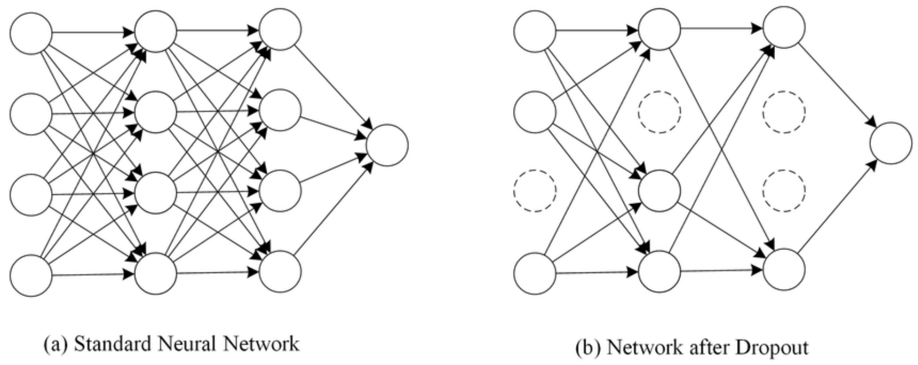
The new term $(1 - \frac{\alpha\lambda}{m})W^{[l]}$ causes the weight to decay in proportion to its size. In practice this penalizes large weights and effectively limits the freedom in your model.

- If λ is too large - $W_{i,j}^{[l]}$ will be close to zeros which will make the NN simpler.
- If λ is good enough it will just reduce some weights and prevent the overfitting.
- If λ is too large, $Z_i^{[l]}$ will be small (close to zero) - assume tanh activation function is used, then it will behave like a linear function, so we will go from non linear activation to roughly linear which would make the NN a roughly linear classifier.
- If λ good enough it will just make some of tanh activations roughly linear which will prevent overfitting.

Implementation tip: if you implement gradient descent, plot the cost function as a function of the number of iterations of gradient descent and you want to see that the cost function decreases monotonically after every elevation of gradient descent with regularization.

Dropout:
The dropout regularization eliminates some neurons/weights on each iteration based on a probability. It is used to reduce overfitting. A most common technique to implement dropout is called ”Inverted dropout”.

$$keep\_prob = 0.8 \text{ (for example)}$$
$$D^{[l]} = np.random.rand(A^{[l]}.shape[0], A^{[l]}.shape[1]) < keep\_prob$$
$$A^{[l]} = A^{[l]} \odot D^{[l]}$$
$$A^{[l]} = keep\_prob^{-1}A^{[l]}$$



Drop 20% of units in $A^{[l]}$ by replacing them with 0, then scale $A^{[l]}$ by $keep\_prob^{-1}$.

We need to scale $A^{[l]}$ because $Z^{[l+1]} = W^{[l+1]}A^{[l]} + B^{[l+1]}$. And we want to increase $A^{[l]}$ to not reduce the expected value of output $Z^{[l+1]}$ after dropout.

$D^{[l]}$ is used for forward and back propagation and is the same for them, but it is different for each iteration of training example. At test time we don't use dropout. Otherwise it would add noise to predictions.

Understanding Dropout:

- The intuition is dropout randomly knocks out units in your network. So it's as if on every iteration you're working with a smaller NN, and so using a smaller NN seems like it should have a regularizing effect.
- Another intuition: can't rely on any single feature, so have to spread out weights.
- Dropout can have different keep_prob per layer.
- The input layer dropout has to be near 1 (or 1 - no dropout) because you don't want to eliminate a lot of features.
- If you're more worried about some layers overfitting than others, you can set a lower keep_prob for some layers than others. The downside is, this gives you even more hyperparameters to search for using cross-validation. One other alternative might be to have some layers where you apply dropout and some layers where you don't apply dropout and then just have one hyperparameter, which is a keep_prob for the layers for which you do apply dropouts.
- A lot of researchers are using dropout with Computer Vision (CV) because they have a very big input size and almost never have enough data, so overfitting is the usual problem.

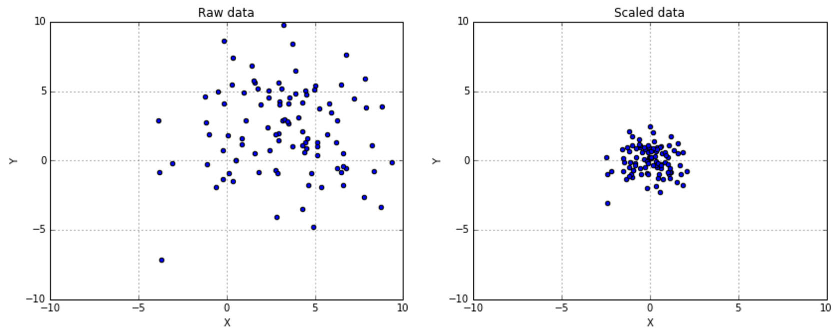
- A downside of dropout is that the cost function is not well defined and it will be hard to debug (plot cost by iteration). To solve that you'll need to turn off dropout, set all the keep_prob to 1, and then run the code and check that it monotonically decreases cost and then turn on the dropouts again.

Other regularization methods:

- Data augmentation. For example, you can flip all your pictures horizontally this will give you m more data instances. You could also apply a random position and rotation to an image to get more data. New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.
- Early stopping. We will pick the point at which the training set error and dev set error are best (lowest training cost with lowest dev cost). We will take these parameters as the best parameters.

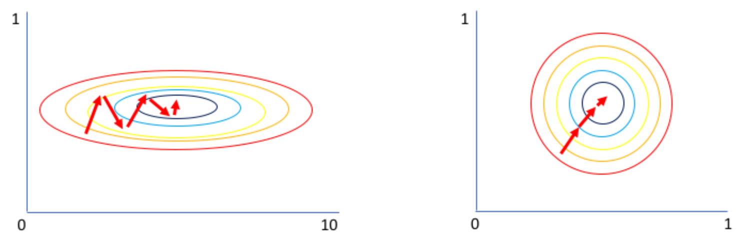
Normalizing Inputs: (Feature Scaling)
All features should be normalized so that they are scaled into a certain range and each feature contributes approximately proportionately to the result. It helps to center the dataset. Also it helps gradient descent converge much faster.

This should be applied to training, dev, and testing sets (but using mean and variance of the training set).



If we normalize inputs, the shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate alpha - the optimization will be faster.

Why normalize?



Gradient of larger parameter dominates the update

Both parameters can be updated in equal proportions

mean normalization:

$$x_i = \frac{x_i - mean(x_i)}{max(x_i) - min(x_i)}$$

standardization:
Feature standardization makes the values of each feature in the data have zero-mean (when subtracting the mean in the numerator) and unit-variance.

$$\mu : \text{mean}, \ \sigma^2 : \text{variance}, \ \sigma : \text{standard deviation}$$

$$\mu_i = \frac{1}{m} \sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_i^2 = \frac{1}{m} \sum_{j=1}^{m} (x_i^{(j)} - \mu_i)^2$$

$$x_i = \frac{x_i - \mu_i}{\sigma_i}$$

Gradient Vanishing/Exploding:
In a deep neural network, when the weights or derivatives for each layer get a little bit smaller/larger, the final gradient could be exponentially smaller/larger with respect to L. Too small/large gradient could cause weights update overshot or in tiny steps, and make training process unable to continue.

Weight Initialization:
A partial solution to the Vanishing / Exploding gradients is better or more careful choice of the random initialization of weights.

Some researcher found that: (He Initialization / Xavier Initialization)

If $tanh$ is used as activation function, the variance of $W^{[l]}$ should be $\frac{1}{s_{l-1}}$, then $W^{[l]}$ should be initialized as:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * np.sqrt(\frac{1}{s_{l-1}})$$

If $relu$ is used as activation function, the variance of $W^{[l]}$ should be $\frac{2}{s_{l-1}}$, then $W^{[l]}$ should be initialized as:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * np.sqrt(\frac{2}{s_{l-1}})$$

Some old paper use this as well:

$$W^{[l]} = np.random.randn(s_{l-1}, s_l) * np.sqrt(\frac{2}{s_{l-1} + s_l})$$

Gradient checking:
Compose W, B into one big vector Θ. We can approximate the derivative of $J(\Theta)$ with respect to Θ_i as:

$$\frac{\partial}{\partial \Theta_i} J(\Theta) \approx \frac{J(\dots, \Theta_i + \epsilon, \dots) - J(\dots, \Theta_i - \epsilon, \dots)}{2\epsilon}$$

A small value for ϵ such as $\epsilon = 10^{-7}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems. Then we can check if $gradApprox \approx deltaVector$.

- if it is $< 10^{-7}$ - great, very likely the backpropagation implementation is correct
- if around 10^{-5} - can be OK, but need to inspect if there are no particularly big values in $d\_theta\_approx - d\_theta$ vector
- if it is $> 10^{-3}$ - bad, probably there is a bug in backpropagation implementation

gradient checking implementation notes:

- Don't use the gradient checking algorithm at training time because it's very slow. Use gradient checking only for debugging.
- If algorithm fails grad check, look at components to try to identify the bug.
- Don't forget to add regularization term to $J(W, B)$ if you are using L1 or L2 regularization.

- Gradient checking doesn't work with dropout because cost function is not consistent. You can first turn off dropout $keep\_prob = 1.0$, run gradient checking and then turn on dropout again.
- Run gradient checking at random initialization and train the network for a while maybe there's a bug which can be seen when w's and b's become larger (further from 0) and can't be seen on the first iteration (when w's and b's are very small).

Mini-batch gradient descent:
When training set is getting too big, it is impossible to use entire training set X, Y to do a vectorized gradient descent. Instead we can separate the training set into mini batches, and do a vectorized gradient descent for each of them at a time.

After we iterate through the entire training set (all batches) for one time, we call it an *epoch*. The model should be trained for enough *epochs* until the cost function converges.

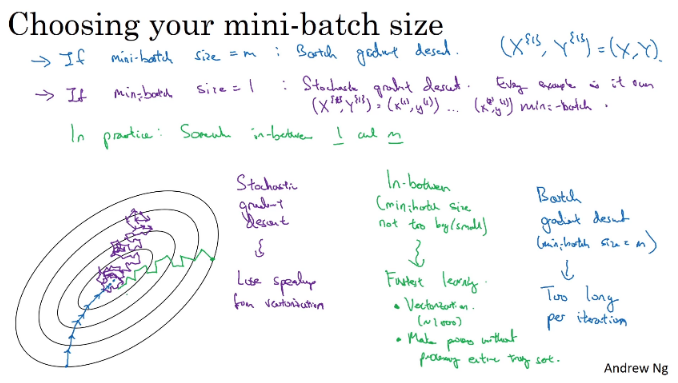
Suppose we have m training examples, and we use $batch\_size = 1000$, then we have $num\_batches = \frac{m}{batch\_size}$. The mini batches are $X^{\{t\}}$, $Y^{\{t\}}$ for $t \in [1, num\_batches]$. And we want to train the model with entire training set for $num\_epochs = 10000$ times.

mini-batch gradient descent algorithm:

```
for $epoch \in \{1, \ldots, num\_epochs\}$:
    for $t \in \{1, \ldots, num\_batches\}$:
        vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
        $W =: W - \alpha \partial W$
        $B =: B - \alpha \partial B$
```

Understanding Mini-Batch Gradient Descent:
In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm (not enough training data in one mini batch, not as effective as batch algorithm). It could contain some ups and downs but generally it has to go down (unlike the batch gradient descent where cost function decreases on each iteration).

- $batch\_size = m$, Batch gradient descent, too long per iteration
- $batch\_size = 1$, Stochastic gradient descent (SGD), too noisy regarding cost minimization (can be reduced by using smaller learning rate), won't ever converge (reach the minimum cost), lose speedup from vectorization
- $batch\_size \in [1, m]$, Mini-batch gradient descent, have the vectorization advantage, make progress without waiting to process the entire training set, doesn't always exactly converge (oscillates in a very small region, but you can reduce learning rate)



Choosing your mini-batch size

→ If mini-batch size = m : Batch gradient descent. $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$.

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is it own mini-batch.
$(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \ldots (x^{(2)}, y^{(1)})$ mini-batch.

In practice: Somewhere in-between 1 and m

Stochastic gradient descent → Lose speedup from vectorization

In-between (mini-batch size not too big/small) → Fastest learning.
- Vectorization. (~1000)
- Make progress without processing entire training set.

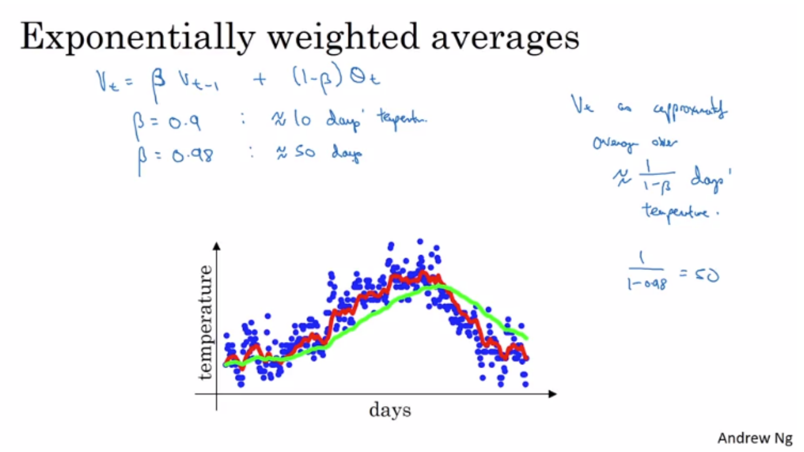
Batch gradient descent (mini-batch size = m) → Too long per iteration

Andrew Ng

Guidelines for choosing mini-batch size:

- If small training set - use batch gradient descent.
- It has to be a power of 2 (because of the way computer memory is Laid out and accessed, sometimes your code runs faster): $2^6, 2^7, 2^8, 2^9, 2^{10} \ldots$
- Make sure that mini-batch fits in CPU/GPU memory.
- Mini-batch size is a hyperparameter.

2.4 Exponentially Weighted Averages



Exponentially weighted averages

$V_t = \beta V_{t-1} + (1-\beta)\theta_t$

$\beta = 0.9$: ≈ 10 days' temperature.

$\beta = 0.98$: ≈ 50 days

V_t as approximately average over $\approx \frac{1}{1-\beta}$ days' temperature.

$\frac{1}{1-0.98} = 50$

Andrew Ng

If we have a daily temperature list like this:

$$\theta_1 = 40, \theta_2 = 49, \theta_3 = 45, \ldots, \theta_{50} = 56$$

We call v_t is the exponentially weighted average of daily temperature over the last $\frac{1}{1-\beta}$ days: ($\beta \in [0, 1)$)

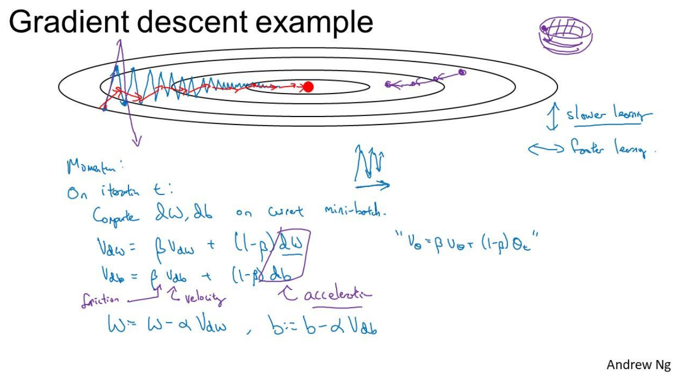
$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= \beta v_0 + (1-\beta)\theta_1 \\
v_2 &= \beta v_1 + (1-\beta)\theta_2 \\
&\dots \\
v_t &= \beta v_{t-1} + (1-\beta)\theta_t
\end{aligned}
$$

Bias Correction of Exponentially Weighted Averages:
The bias correction helps make the exponentially weighted averages more accurate. Because $v_0 = 0$, the initial weighted averages are too small and the accuracy suffers at the start. To fix the bias for the initial estimates we have to use this equation: (As t becomes larger the $1-\beta^t$ becomes close to 1)

$$v_t = \frac{\beta v_{t-1} + (1-\beta)\theta_t}{1-\beta^t}$$

2.5 Gradient Descent With Momentum

The momentum algorithm almost always works faster than standard gradient descent. It smooths out the steps of gradient descent by using the exponentially weighted average of the gradients. In practice people don't bother implementing bias correction.



Gradient descent example
slower learning
faster learning
Momentum:
On iteration t:
Compute dW, db on current mini-batch.
$V_{dw} = \beta V_{dw} + (1-\beta)dW$
$V_{db} = \beta V_{db} + (1-\beta)db$
"$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"
friction
velocity
acceleration
$W := W - \alpha V_{dw}$, $b := b - \alpha V_{db}$
Andrew Ng

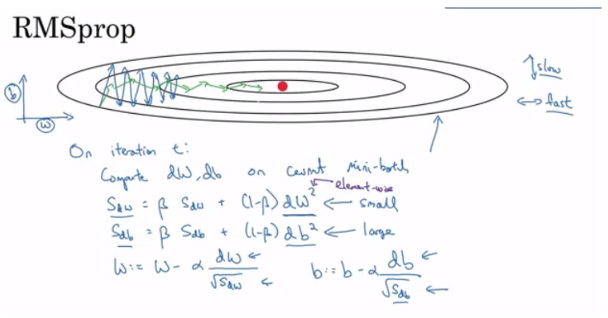
gradient descent with momentum:

```
$v_{\partial W} = 0, v_{\partial B} = 0$
for $epoch \in \{1, \dots, num\_epochs\}$:
    for $t \in \{1, \dots, num\_batches\}$:
        vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
        $v_{\partial W} =: \beta v_{\partial W} + (1-\beta)\partial W$
        $v_{\partial B} =: \beta v_{\partial B} + (1-\beta)\partial B$
        $W =: W - \alpha v_{\partial W}$
        $B =: B - \alpha v_{\partial B}$
```

2.6 RMSProp(Root mean square prop)

RMSprop can speed up the gradient descent by dividing the gradient ∂W, ∂B by a manually constructed term: If gradient is small, divided by a small term will make it larger, and make gradient descent faster in this direction; If gradient is large, divided by a large term will make it smaller, and make gradient descent slower in this direction.

It will make the cost function move slower on the vertical direction and faster on the horizontal direction in the following example:


RMSprop
b
w
slow
fast
On iteration t:
Compute dW, db on current mini-batch
element-wise
$S_{dW} = \beta S_{dW} + (1-\beta) dW^2$ ← small
$S_{db} = \beta S_{db} + (1-\beta) db^2$ ← large
$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$
$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$

```
RMSProp:
$s_{\partial W} = 0, s_{\partial B} = 0$
make sure denominator is not too small: $\epsilon = 10^{-8}$
for $epoch \in \{1, \ldots, num\_epochs\}$:
    for $t \in \{1, \ldots, num\_batches\}$:
        vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
        $s_{\partial W} =: \beta s_{\partial W} + (1-\beta)\partial W \odot \partial W$
        $s_{\partial B} =: \beta s_{\partial B} + (1-\beta)\partial B \odot \partial B$
        $W =: W - \alpha \frac{\partial W}{\sqrt{s_{\partial W}} + \epsilon}$
        $B =: B - \alpha \frac{\partial B}{\sqrt{s_{\partial B}} + \epsilon}$
```

2.7 Adam Optimization Algorithm

Adam (Adaptive Moment Estimation) optimization algorithm combined momentum and RMSProp methods. Hyperparameters for Adam:

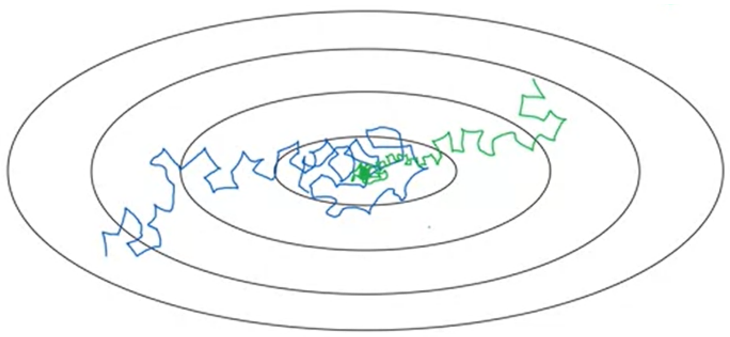
- α: learning rate
- β_1: parameter of the momentum - 0.9 is recommended by default.
- β_2: parameter of the RMSprop - 0.999 is recommended by default.
- ϵ: 10^{-8} is recommended by default.

adam optimization algorithm:

```
$v_{\partial W} = 0, v_{\partial B} = 0, s_{\partial W} = 0, s_{\partial B} = 0$
make sure denominator is not too small: $\epsilon = 10^{-8}$
for $epoch \in \{1, \dots, num\_epochs\}$:
    for $t \in \{1, \dots, num\_batches\}$:
        vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
        $v_{\partial W} =: \beta_1 v_{\partial W} + (1 - \beta_1)\partial W$
        $v_{\partial B} =: \beta_1 v_{\partial B} + (1 - \beta_1)\partial B$
        $s_{\partial W} =: \beta_2 s_{\partial W} + (1 - \beta_2)\partial W \odot \partial W$
        $s_{\partial B} =: \beta_2 s_{\partial B} + (1 - \beta_2)\partial B \odot \partial B$
        $v_{\partial W}^{corrected} =: \frac{v_{\partial W}}{1-\beta_1^t}$
        $v_{\partial B}^{corrected} =: \frac{v_{\partial B}}{1-\beta_1^t}$
        $s_{\partial W}^{corrected} =: \frac{s_{\partial W}}{1-\beta_2^t}$
        $s_{\partial B}^{corrected} =: \frac{s_{\partial B}}{1-\beta_2^t}$
        $W =: W - \alpha \frac{v_{\partial W}^{corrected}}{\sqrt{s_{\partial W}^{corrected}} + \epsilon}$
        $B =: B - \alpha \frac{v_{\partial B}^{corrected}}{\sqrt{s_{\partial B}^{corrected}} + \epsilon}$
```

2.8 Learning Rate Decay

Slowly reduce learning rate for each epoch helps cost function to converge near the optimum point.



learning rate decay:
initial learning rate: α_0, decay rate: d
for each epoch p: $\alpha_p = \frac{1}{1+d\times p}\alpha_0$

other learning rate decay:

- exponentially decay: $\alpha_p = o.95^p \alpha_0$
- $\alpha_p = \frac{k}{\sqrt{p}}\alpha_0$ or $\alpha_t = \frac{k}{\sqrt{t}}\alpha_0$
- manually learning rate decay.

2.9 Tuning Process

Hyperparameters priority: (as for Andrew Ng)

- α
- β
- s_l
- $batch\_size$
- L
- d
- $\beta_1, \beta_2, \epsilon$

Tune Hyperparameters:

- give each Hyperparameter a range
- randomly pick a value for each Hyperparameter in its range, then train with them
- randomly picking helps to explore Hyperparameters in a larger range (compare with increasing by steps), which can give hints to narrow the range and speed up the tuning process
- use Coarse to fine sampling scheme: when you find some hyperparameter values that give you a better performance - zoom into a smaller region around these values and sample more densely within this space

Using scale to pick hyperparameters
Let's say you have a specific wide range for a hyperparameter from "a" to "b". It's better to search for the right ones using the logarithmic scale rather then in linear scale:

$$a = -4, b = 0$$
$$r \in [a, b]$$
$$r = -4 \times np.random.rand()$$
$$\alpha = 10^r$$

Panda vs. Caviar
Intuitions about hyperparameter settings from one application may or may not transfer to a different one.

- If you don't have much computational resources you can use the "babysitting model" (Panda Approach): Day 0 you might initialize your parameter as random and then start training. Then you watch your learning curve gradually decrease over the day. And each day you nudge your parameters a little during training.
- If you have enough computational resources, you can run several models in parallel and at the end of the day(s) you check the results (Caviar Approach).

Batch Normalization:
Normalized inputs could be shifted in the hidden layers. Batch normalization reduces the problem of input values changing (shifting). And it speeds up learning as well. In practice, normalizing $Z^{[l]}$ is done much more often and that is what Andrew Ng presents.

For a layer l, given a mini batch of samples $Z^{[l](1)}, \dots, Z^{[l](m)}$:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} Z^{[l](i)}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (Z^{[l](i)} - \mu) \odot (Z^{[l](i)} - \mu)$$

$$Z_{norm}^{[l](i)} = (Z^{[l](i)} - \mu) \oslash \sqrt{\sigma^2 + \epsilon}$$

$$\overset{N^{[l](i)}}{Z} = \gamma^{[l]} \odot Z_{norm}^{[l](i)} + \beta^{[l]}$$

ϵ is added for numerical stability (in case σ is too small). γ, β are learnable parameters of the model, and they make inputs belong to other distribution (with other mean and variance). It makes the NN learn the distribution of the outputs.

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.

Gradient Descent with Batch Normalization:
(also can work with Momentum, RMSprop, Adam...)

```
for $epoch \in \{1, \dots, num\_epochs\}$:
    for $t \in \{1, \dots, num\_batches\}$:
        vectorized gradient descent with mini batch $X^{\{t\}}$, $Y^{\{t\}}$:
        use $\overset{N^{\{t\}}}{Z}$ to replace $Z^{\{t\}}$
```

B will be replaced by β
$W =: W - \alpha\partial W$
$\gamma =: \gamma - \alpha\partial\gamma$
$\beta =: \beta - \alpha\partial\beta$

Batch normalization at test time:
When we train a NN with Batch normalization, we compute the mean and the variance of the mini-batch. In testing we might need to process examples one at a time. The mean and the variance of one example won't make sense. We have to compute an estimated value of mean and variance to use it in testing time. We can use the exponentially weighted average of μ, σ^2 across the mini-batches. In practice most often you will use a deep learning framework and it will contain some default implementation of doing such a thing.

Given t mini batches of training examples:
μ_{test} is exponentially weighted average of $\mu^{\{1\}}, \mu^{\{2\}}, \ldots, \mu^{\{t\}}$
σ^2_{test} is exponentially weighted average of $\sigma^{2\{1\}}, \sigma^{2\{2\}}, \ldots, \sigma^{2\{t\}}$

$$Z_{norm} = (Z - \mu_{test}) \oslash \sqrt{\sigma^2_{test} + \epsilon}$$

$$\overset{N}{Z} = \gamma \odot Z_{norm} + \beta$$

Softmax Regression:
Softmax regression is used for multiclass classification/regression. It is normally used in the last layer. Each of values in the output layer represents a probability of the example to belong to each of the classes, and they sum up to 1.

$$t = e^{Z^{[L]}}$$

$$A^{[L]} = \frac{t}{sum\{t\}}$$

Train with Softmax:
With machine learning framework like Tensorflow, Forward Propagation needs to be built, Backpropagation part can be handled automatically .

$$L(\hat{y}, y) = -\sum_{k=1}^{K} y_k log(\hat{y}_k)$$

$$J(W, B) = -\frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

$$\partial Z^{[L]} = \hat{y} - y$$

$$\partial Z^{[l]} = \ldots$$

Local Optima:

- The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.
- It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the saddle point rather to the local optima, which is not a problem.
- Plateaus can make learning slow. Plateau is a region where the derivative is close to zero for a long time. This is where algorithms like momentum, RMSprop or Adam can help.

3 Structuring Machine Learning Projects

3.1 Orthogonalization

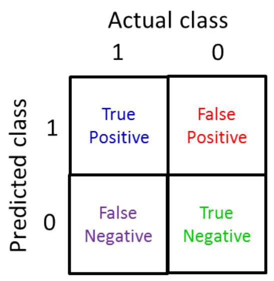
Some deep learning developers know exactly what hyperparameter to tune in order to try to achieve one effect. This is a process we call orthogonalization. In orthogonalization, you have some controls, but each control does a specific task and doesn't affect other controls.

For a supervised learning system to do well, you usually need to tune the knobs of your system to make sure that four things hold true - chain of assumptions in machine learning:

- You'll have to fit training set well on cost function (near human level performance if possible). If it's not achieved you could try bigger network, another optimization algorithm (like Adam)...
- Fit dev set well on cost function. If its not achieved you could try regularization, bigger training set...
- Fit test set well on cost function. If its not achieved you could try bigger dev set...
- Performs well in real world. If its not achieved you could try change dev set, change cost function...

3.2 Single Number Evaluation Metric

Precision & Recall: (deal with skewed classes)
High $F_{score} \in [0, 1]$ (high precision and high recall) represents a good prediction model. (use average value of precision and recall for all classes if there are multiple classes)



$$\text{Precision} = \frac{\text{True positives}}{\text{predicted as positive}} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

$$\text{Recall} = \frac{\text{True positives}}{\text{actual positives}} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

$$F_{score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Satisficing and Optimizing Metrics:
Assume we have N metrics to evaluate the model, normally we will have 1 optimizing metric, and $N-1$ satisficing metrics. For example: Maximize optimizing metric F_{score}, subject to satisficing metrics $RunningTime < 100ms$, $ModelSize < 1gb$.

3.3 Train/Dev/Test Set Distributions

Dev and test sets have to come from the same distribution. Choose dev set and test set to reflect data you expect to get in the future and consider important to do well on. Setting up the dev set, as well as the validation metric is really defining what target you want to aim at.

Size of the dev and test sets:
An old way of splitting the data was 70% training, 30% test or 60% training, 20% dev, 20% test. The old way was valid for a number of examples < 100000 In the modern deep learning if you have a million or more examples a reasonable split would be 98% training, 1% dev, 1% test.

When to change dev/test sets and metrics:
Orthogonalization for deep learning: break a machine learning problem into distinct steps.

- Figure out how to define a metric that captures what you want to do - place the target.
- Worry about how to actually do well on this metric - how to aim/shoot accurately at the target.

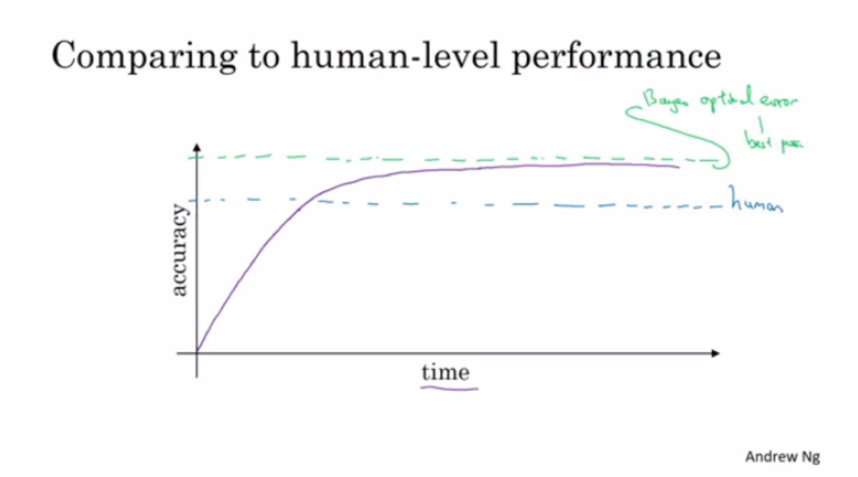
Conclusion: if doing well on your metric + dev/test set doesn't correspond to doing well in your application, change your metric and/or dev/test set.

3.4 Human-level Performance

We compare to human-level performance because of two main reasons:

- Because of advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance.
- It turns out that the workflow of designing and building a machine learning system is much more efficient when you're trying to do something that humans can also do.

After an algorithm reaches the human level performance the progress and accuracy slow down.



You won't surpass an error that's called "Bayes optimal error" (the minimum possible error that can be made). There isn't much error range between human-level error and Bayes optimal error.

Humans are quite good at a lot of tasks. So as long as Machine learning is worse than humans, you can:

- Get labeled data from humans.
- Gain insight from manual error analysis: why did a person get it right?
- Better analysis of bias/variance.

Avoidable bias:
Suppose that the cat classification algorithm gives these results:

Humans	1%	7.5%
Training error	8%	8%
Dev Error	10%	10%

The human-level error as a proxy (estimate) for Bayes optimal error. Bayes optimal error is always less (better), but human-level in most cases is not far from it. You can't do better than Bayes error unless you are overfitting.

$$\text{Avoidable Bias} = \text{Training error} - \text{Human (Bayes) error}$$

$$\text{Measure of Variance} = \text{Dev error} - \text{Training error}$$

In the left example, because the Avoidable Bias is larger (7%) then we need to focus on the bias. In the right example, because the Measure of Variance is larger (2%) then we need to focus on the variance.

You might have multiple human-level performances based on different groups of people. Choose the best human-level performance as it is closer to Bayes optimal error.

These techniques allow you to make decisions more quickly as to whether you should focus on trying to reduce the bias or trying to reduce the variance of your algorithm. This tend to work well until you surpass human-level performance, whereupon you might no longer have a good estimate of Bayes error that still helps you make this decision really clearly.

Surpassing human-level performance:
In some problems, deep learning has surpassed human-level performance. Like:

- Online advertising.
- Product recommendation.
- Logistics (predict transit time)
- Loan approval.

These examples are learning on structural data with lots of data, and they are not natural perception task. Humans behave well in natural perception tasks like medical, computer vision and speech recognition, and it's harder to surpass human-level performance in these tasks.

3.5 Error analysis

Error analysis - process of manually examining mistakes that your algorithm is making. It can give you insights into what to do next. Error analysis helps you to analyze the error before taking an action which could take unnecessary efforts.

Sometimes, you can evaluate multiple error analysis ideas in parallel and choose the best idea, by creating a spreadsheet of them. This quick counting procedure, which takes small numbers of hours, can really help you make much better prioritization decisions, and understand how promising different approaches are to work on.

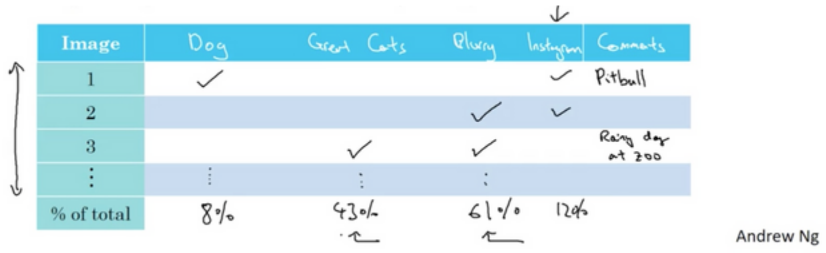
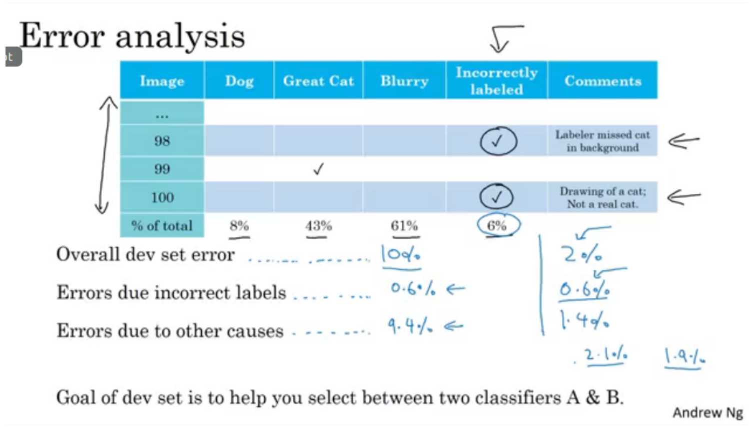


Image	Dog	Great Cats	Blurry	Instagram	Comments
1	✓			✓	Pitbull
2			✓	✓	
3		✓	✓		Rainy day at zoo
⋮	⋮	⋮	⋮		
% of total	8%	43%	61%	12%	

Andrew Ng

Incorrectly Labeled Data:
DL algorithms are quite robust to random errors in the training set but less robust to systematic errors. Only fix these labels for training set if you can. If you want to check for mislabeled data in dev/test set, try error analysis with the mislabeled column.



Error analysis

Image	Dog	Great Cat	Blurry	Incorrectly labeled	Comments
…					
98				✓	Labeler missed cat in background
99		✓			
100				✓	Drawing of a cat; Not a real cat.
% of total	8%	43%	61%	6%	

Overall dev set error	10%	2%
Errors due incorrect labels	0.6%	0.6%
Errors due to other causes	9.4%	1.4%
		2.1% 1.9%

Goal of dev set is to help you select between two classifiers A & B.

Andrew Ng

If other errors take up the majority of dev set error, fixing mislabeled data is not the first priority. If mislabeled data is a significant part of dev set error, then you need to fix it.

Consider these guidelines while correcting the dev/test mislabeled examples:

- Apply the same process to your dev and test sets to make sure they continue to come from the same distribution.
- Consider examining examples your algorithm got right as well as ones it got wrong. (Not always done if you reached a good accuracy)
- Train and dev/test data may now come from a slightly different distributions. It's very important to have dev and test sets to come from the same distribution. But it could be OK for a train set to come from slightly other distribution.

Build First System Quickly, Then Iterate:
The steps you take to make your deep learning project:

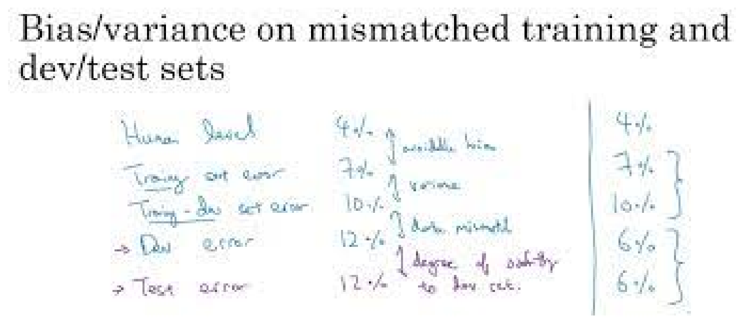
- Setup dev/test set and metric.
- Build initial system quickly.
- Use Bias/Variance Analysis and Error Analysis to prioritize next steps.

Training and Dev/Test set on different distributions:
A lot of teams are working with deep learning applications that have training sets that are different from the dev/test sets due to the hunger of deep learning to data.

There is a strategy to split data into train, dev/test sets:

- Use data from real application feedback data for dev/test sets (for example user image, experience or feedback). This sets a more accurate target and reduces bias for the DL problem.
- Use purchased data, collected data from different sources, plus the real application feedback data for training set. This provides enough amount of data to reduce variance.

Bias and Variance with mismatched data distributions:
Bias and Variance analysis changes when training and Dev/test set is from the different distribution, because you need to consider the influence of distribution.



To solve this issue we create a new set called train-dev set as a random subset of the training set (so it has the same distribution with training set, and we don't use this part of training set to train the model) and we get:

$$\text{Avoidable Bias} = \text{Training error} - \text{Human (Bayes) error}$$

$$\text{Variance} = \text{Training-dev error} - \text{Training error}$$

$$\text{Data mismatch} = \text{Dev error} - \text{Training-dev error}$$

$$\text{Degree of Overfitting to Dev set} = \text{Test error} - \text{Dev error}$$

Addressing data mismatch:
There aren't completely systematic solutions to this, but there some things you could try.

- Carry out manual error analysis to try to understand the difference between training and dev/test sets.
- Make training data more similar to dev/test sets, or collect more data similar to dev/test sets.

If your goal is to make the training data more similar to your dev set, one of the techniques you can use is Artificial data synthesis. (Combine some of your training data with some other data and produce some data that similar to the dev/test set distribution.) For example, combine normal audio with car noise to get audio with car noise example, generate cars using 3D graphics.

Be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples because your NN might overfit these generated data (like particular car noise or a particular design of 3D graphics cars).

3.6 Transfer learning

Apply the knowledge you took in a task A and apply it in another task B. For example, you have trained a cat classifier with a lot of data, you can use part of the trained NN to solve x-ray classification problem.

To do transfer learning, delete the last layer(s) of NN and it's weights and:

- Option 1: if you have a small data set - keep all the other weights as fixed weights. Add new last layer(s) and initialize the weights for the new layer(s), then feed the new data to the NN and learn the new weights.
- Option 2: if you have enough data you can keep less fixed weights and retrain the weights for more layers.

Training on task A called pre-training and Option 1 and 2 are called fine-tuning.

When transfer learning make sense:

- Task A and B have the same input X (e.g. image, audio).
- You have a lot of data for the task A you are transferring from and relatively less data for the task B your transferring to.
- Low level features from task A could be helpful for learning task B.

3.7 Multi-task learning

In multi-task learning, you try to have one neural network do several things at the same time.

For example, build an object recognition system that detects pedestrians, cars, stop signs, and traffic lights (image has multiple labels). Then Y shape will be $(4, m)$ because we have 4 classes and each one is a binary one.

Multi-task learning makes sense:

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually, amount of data you have for each task is quite similar.
- Can train a big enough network to do well on all the tasks.

If you can train a big enough NN, the performance of the multi-task learning compared to splitting the tasks is better. Today transfer learning is used more often than multi-task learning.

3.8 End-to-end deep learning

Some systems break a task into multiple steps, and build a NN for each step. An end-to-end deep learning system implements all these steps with a single NN.

For example:

- Face recognition system. In practice, the best approach is the multi-steps approach for now. It uses one NN for face detection, and another NN which takes two faces as inputs, then outputs if the two faces are the same person or not.

  $$\text{Image} - - > \text{Face detection} - - > \text{Face recognition}$$

  $$\text{Image} - - - - - - - - - - - - - > \text{Face recognition}$$

- Machine translation system. Here end-to-end deep leaning system works better because we have enough data to train the NN.

  $$\text{English} - - > \text{Text analysis} - - > \cdots - - > \text{French}$$

  $$\text{English} - - - - - - - - - - - - - - - - - > \text{French}$$

To build the end-to-end deep learning system that works well, we need a big dataset (more data than in non end-to-end system). If we have a small dataset, the multi-steps approach could work better.

Whether to use end-to-end deep learning:
Key question: Do you have sufficient data to learn a function of the complexity needed to map X to Y? When applying supervised learning you should carefully choose what types of X to Y mappings you want to learn depending on what task you can get data for.

Pros of end-to-end deep learning:

- By having a pure deep learning approach, your NN learning input from X to Y may be more able to capture whatever statistics are in the data, rather than being forced to reflect human preconceptions.
- Less hand-designing of components needed.

Cons of end-to-end deep learning:

- May need a large amount of data.
- Excludes potentially useful hand-design components (it helps more on the smaller dataset).

Multi-steps approach could be better if a complex system can be divided into multiple tasks and they have a good mapping from data to tasks (autonomous driving), and you are able to build ML/DL models for individual components.