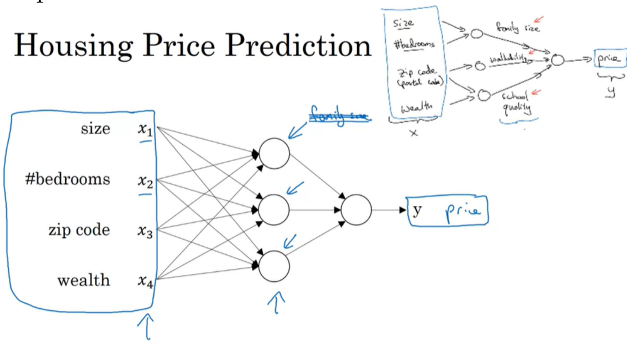
1 Intro to Deep Learning

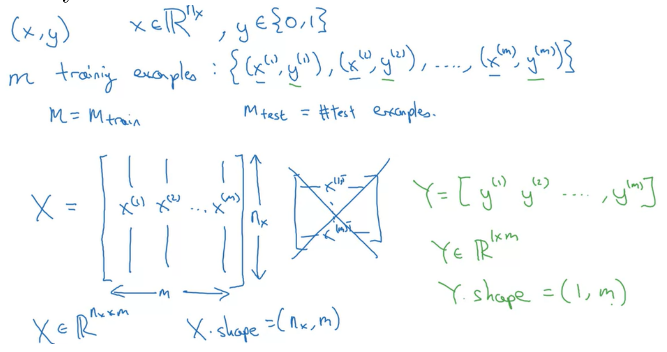
- Simple Neural Network



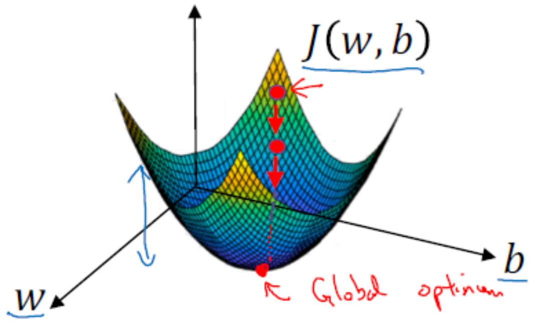
- Supervised Learning Use Cases

Input(x)	Output(y)	Application
Home features	Price	Real Estate
Ad, user info	Click on ad? (0/1)	Online Advertising
Image	Object (1,...,1000)	Photo Tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine Translation
Image, Radar info	Position of other cars	Autonomous driving

- Scale drives deep learning progress
  - Data
  - Computation
  - Algorithms (i.e. Sigmoid vs. reLU)
- Binary Classification

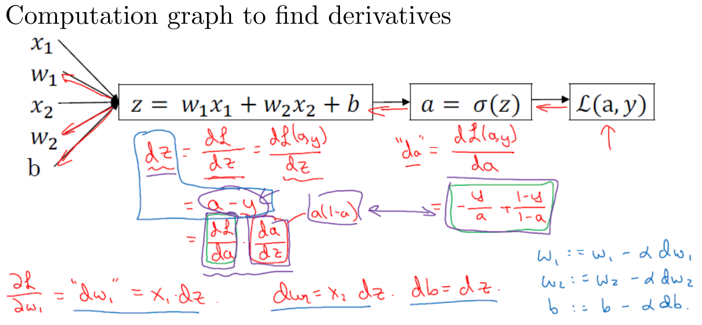

(x, y) $x \in \mathbb{R}^{n_x}$, $y \in \{0,1\}$
m training examples : $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$
$M = M_{train}$ M_{test} = #test examples.
$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix}$ n_x m
$Y = [y^{(1)} \; y^{(2)} \dots, y^{(m)}]$
$Y \in \mathbb{R}^{1 \times m}$
$Y.shape = (1, m)$
$X \in \mathbb{R}^{n_x \times m}$ $X.shape = (n_x, m)$

- Logistic Regression
  - $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^-z}$
  - Loss: $\mathcal{L}(\hat{y}, y) = -(ylog\hat{y} + (1-y)log(1-\hat{y}))$
  - Cost: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$
- Gradient Descent
  - Want to find w, b that minimizes $J(w, b)$

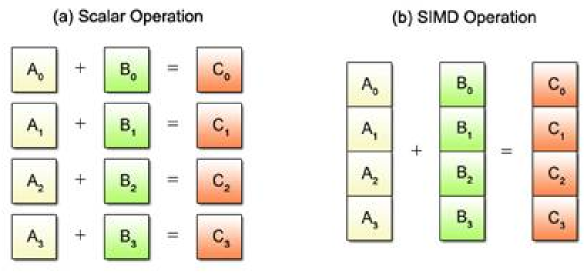


  - Update parameters as follows:
    * $w_i := w_i - \alpha\frac{\partial J(w_i,b)}{\partial w_i}$
    * $b_i := b_i - \alpha\frac{\partial J(w,b_i)}{\partial b_i}$

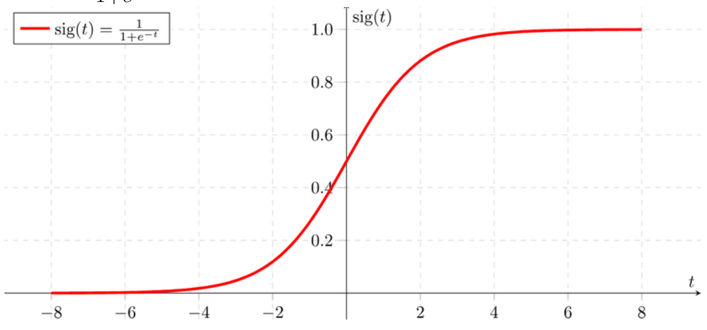
  - Computation graph to find derivatives


x_1
w_1
x_2
w_2
b
$z = w_1x_1 + w_2x_2 + b$
$a = \sigma(z)$
$\mathcal{L}(a, y)$
$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz}$
$= a - y$
$= \frac{d\mathcal{L}}{da}\frac{da}{dz}$
$a(1-a)$
"da" $= \frac{d\mathcal{L}(a,y)}{da}$
$= -\frac{y}{a} + \frac{1-y}{1-a}$
$\frac{\partial \mathcal{L}}{\partial w_1} =$ "dw_1" $= x_1 \cdot dz$
$dw_2 = x_2\, dz$. $db = dz$.
$w_1 := w_1 - \alpha\, dw_1$
$w_2 := w_2 - \alpha\, dw_2$
$b := b - \alpha\, db$.

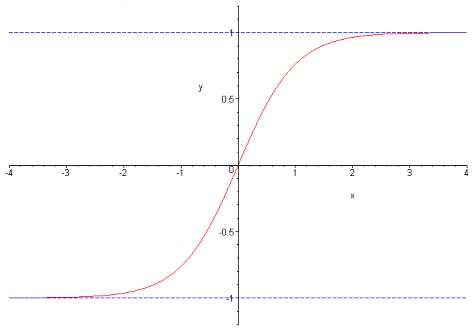
- Vectorization
  - Whenever possible, avoid explicit for-loops
  - Vectorize examples by adding columns to matrices: Z, X, A, b
  - Can by-pass using a for-loop and take advantage of the SIMD paradigm.
  - SIMD: Single Instruction Multiple Data, a method for combining multiple operations into a single computer instruction



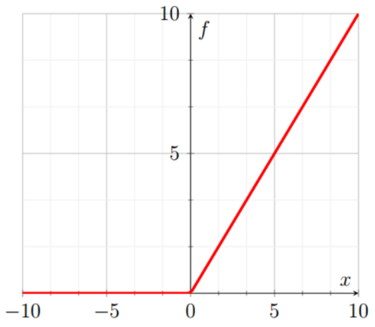
  - Why use GPUs over CPUs?
    * Are optimized for parallel computing
    * Have thousands of cores
    * Have multiple hyperthreads per core
  - Why use CPUs over GPUs?
    * CPUs process data sequentially
    * They do not know what instruction will be next (i.e. input from keyboard, mouse, ...)
    * Has resources to manage an Operating System
- Activation functions
  - Activation functions are needed in order for the model to learn non-linear decision boundaries
  - Four common activation functions. All are monotonic, continuous, and differentiable
  - Sigmoid: $(-\infty, +\infty) \rightarrow (0, 1)$
    $f(x) = \frac{1}{1+e^{-x}}$



  - Tanh: $(-\infty, +\infty) \rightarrow (-1, 1)$
    $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



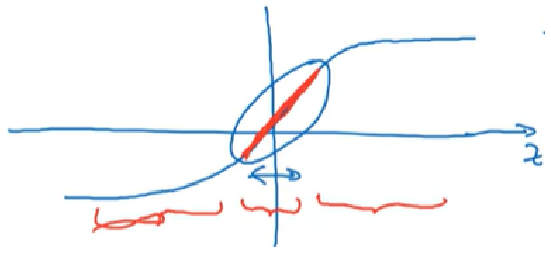
  - ReLu: $(-\infty, +\infty) \rightarrow (0, +\infty)$
    $f(x) = max(0, x)$



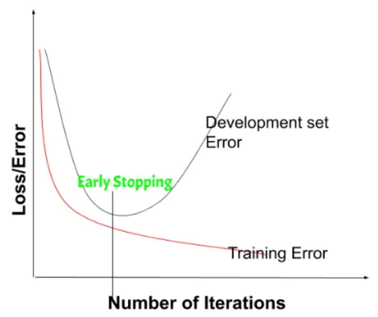
  - Softmax: $(-\infty, +\infty) \rightarrow (0, 1)$
    $f(x) = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}}$
    Softmax is most often used in the final output layer. The output layer has values between 0 and 1. The function transforms a bunch of numbers into a valid probability distribution.
- Random initialization
  - Initialize weights to random numbers. Do not initialize them to zero. When all weights are zero, the network has perfect symmetry, meaning every neuron updates identically (that is zero) during backpropagation, leading to no meaningful learning
  - Initialize bias to zero vector.
    * do not initilize weights and bias to zeros
    * initializing to large random values doesn't work well. The last activiation outputs results very close to 0 or 1 and then incurs a very high loss.
    * initializing to small random values is best
    * choose random weights from gaussian distribution to avoid being too close to the extremes (as in a uniform distribution). It is easier to find the slope in the center of the distribution
    * He initialization works well for networks with ReLU activations (see 2. Optimization)
- Matrix dimensions
  - $W^{[l]} : (n^{[l]}, n^{[l-1]})$
  - $b^{[l]} : (n^{[l]}, 1)$
- Hyperparameters: pre-set by the practitioner
- Parameters: set by optimization

2 Hyperparameter tuning, regularization, and optimization

- Train/dev/test sets
  - In the age of big data, it's fine if you don't comply with the rule of thumb of 70/30 (train/test split)
  - Make sure dev and test set come from same distribution
  - Not having a test set might be okay (only dev set)
- Bias/Variance
  - Bias - Variance used to assess how to best improve the model. You can assess them by examining the train/dev set errors and comparing them to the human (or bayes error).
  - High Bias - a non-complex model that underfits the training data. Create bigger network, train longer to lower it.
  - High Variance - a complex model that overfits the training data. Add more data and regularize to lower it.
- Regularization
  - Generally, the first step to fixing high variance. Adding more data isn't always possible.
  - 1. Frobenius Norm
    * The Frobenius Norm penalty will penalize the weight matrices from being too large
    * Frobenius Norm:
      $||w^{[l]}||_F^2 = \Sigma_{i=1}^{n^{[l]}} \Sigma_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$
    * $J(w, b) = \frac{1}{m} \Sigma_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \Sigma_{l=1}^{L} ||w^{[l]}||_F^2$
    * Weight decay - regularization technique that results in gradient descent shrinking the weights on every iteration
    * $\uparrow \lambda \Rightarrow \downarrow w^{[l]} \Rightarrow \downarrow z^{[l]} \Rightarrow$ every layer of the network is linear

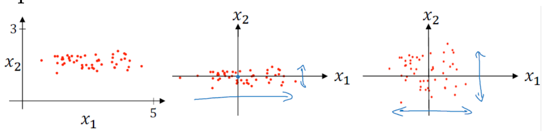


  - 2. Dropout
    * Randomly shutting down neurons in each iteration - mostly applied in computer vision problems
    * Inverted dropout - divide the activations by the keep probability to ensure the result of the cost will have the same expected value without drop-out.
    * It works by spreading out the weights and preventing reliance on any one feature.
    * Dropout is only used in training.
    * Dropout is applied during forward and backward propagation
  - 3. Data augmentation
    * An inexpensive way of generating more data to reduce overfitting (i.e. invert, distort, crop, ... to images)
  - 4. Early stopping
    * Stop training a neural network once the dev set error starts to increase.



- ∗ Con: not using orthogonalization which de-couples (1) optimizing the cost function and (2) not overfitting the model

· Optimization
- – Normalization
  - ∗ Technique of putting all values on the same scale. Generally by taking the input, subtracting the mean, and dividing by the standard deviation.
  - ∗ Cost function will be more round and faster to optimize



- – Vanishing/Exploding Gradients
  - ∗ When parameters are all slightly larger than one $->$ Exploding Gradient
  - ∗ When parameters are all slightly smaller than one $->$ Vanishing Gradient
  - ∗ Common in deep networks - both will prevent the model from converging to the optimal set of parameters
  - ∗ Xavier initialization solution - initialize the weight matrix of each layer such that its variance is $1/n$ where n is the number of input features.
    Relu: $\sqrt{\frac{2}{n^{[l-1]}}}$
    Tanh: $\sqrt{\frac{1}{n^{[l-1]}}}$
- – Gradient checking
  - ∗ Gradient checking verify if you computed the gradients properly. It checks the closeness between the gradients (via backpropagation) and the gradients (via numerical approximation using forward propagation).
  - ∗ Only use it to make sure your code is correct.
  - ∗ $gradapprox = \frac{\partial J}{\partial \theta} = \lim_{\epsilon \to 0} \frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$
    $difference = \frac{||grad-gradapprox||_2}{||grad||_2+||gradapprox||_2}$
    If $difference \ < \ 2 * 10^{-7}$, then your backward propagation works fine.

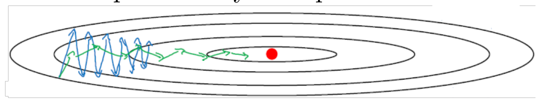
· Optimization Algorithms
- – Mini-batch - partition examples into mini batches and use them to compute the parameters. Get advantage of vectorization and faster convergence
- – Stochastic gradient descent - special case where mini-batch size = 1
- – Epoch - one pass through the training set
- – For large datasets, mini-batch runs faster than batch gradient descent
- – Ideal size
  - ∗ if <= 2,000 examples, use batch gradient descent
  - ∗ mini-batch sizes are a power of 2 (i.e. 64, 128, 256, ...) to take advantage of how ram is layed out in hardware
  - ∗ make sure mini-batch size fits in cpu/gpu memory
- – Implementation

```
t = 1,...,num_batches:
forward prop on $X^t$ (vectorized implementation)
    $Z^{[1]} = W^{[1]}X^t + b^1$
    $A^{[1]} = g^{[1]}(Z^{[1]})$
    .
    .
    $A^{[L]} = g^{[L]}(Z^{[L]})$
compute cost
    $J^t = \frac{1}{batchsize}\sum_{i=1}^{l}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2*batchsize}\sum_{i=1}^{l}||W^{[l]}||_F^2$
back prop to compute gradients
update parameters
    $W^{[l]} := W^{[l]} - \alpha\partial W^{[l]}$
    $b^{[l]} := b^{[l]} - \alpha\partial b^{[l]}$
```

- – Gradient descent with momentum
  - ∗ Exponentially weighted average (momentum)
    - · β is our smoothening parameter (i.e. $\beta = 0.9$ is averaging our loss over 10 gradients)
    - · But this works bad for initial few examples. You can do better if you divide by $1 - \beta^t$
    - · $v_t = \beta * v_{t-1} + (1-\beta)\theta_t$
    - · $v_{dW^{[l]}} = \beta * v_{dW^{[l]}} + (1-\beta)dW^{[l]}$
      $W^{[l]} := W^{[l]} - \alpha * v_{dW^{[l]}}$
    - · $v_{db^{[l]}} = \beta * v_{db^{[l]}} + (1-\beta)db^{[l]}$
      $b^{[l]} := b^{[l]} - \alpha * v_{db^{[l]}}$
  - ∗ RMS Prop
    - · Adapt learning rate based on moving average of squared gradients
    - · $S_{dW} = \beta * S_{dW} + (1-\beta)dW^2$
      $S_{db} = \beta * S_{db} + (1-\beta)db^2$
      $W := W - \alpha\frac{\partial dW}{\sqrt{S_{dW}}}$
      $b := b - \alpha\frac{\partial db}{\sqrt{S_{db}}}$
  - ∗ Adam
    - · Combines momentum (moving average of the first gradient) with RMS prop (moving average of the squared gradient)

$$\begin{cases} v_{W^{[l]}} = \beta_1 v_{W^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial W^{[l]}} \\ v^{corrected}_{W^{[l]}} = \frac{v_{W^{[l]}}}{1-(\beta_1)^t} \\ s_{W^{[l]}} = \beta_2 s_{W^{[l]}} + (1-\beta_2)(\frac{\partial J}{\partial W^{[l]}})^2 \\ s^{corrected}_{W^{[l]}} = \frac{s_{W^{[l]}}}{1-(\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha\frac{v^{corrected}_{W^{[l]}}}{\sqrt{s^{corrected}_{W^{[l]}}}+\varepsilon} \end{cases}$$

  - ∗ Learning rate decay - lower the learning rate as you get closer to the minima. One variation is below
    $\alpha = \frac{1}{1+decayrate*epochnum} * \alpha_0$
  - ∗ Learning rate decay controls the step size while momentum controls the direction of the steps taken by the optimizer



  - ∗ In a high dimensional space, most optima are saddle points, so you will not get stuck in local optima. However, plateaus are a problem and the gradient may become very close to 0 and learning will be very slow. This is where momentum, rms prop, or adam

can help.
  - Hyperparameter tuning
    * Try random values: don't use a grid. Focus more resources on searching in a fine focused region.
    * Appropriately scale for hyperparameters (i.e. scale is wide, consider log scale). This is especially true for exponentially weighted averages (i.e. β)
  - Batch normalization
    * Normalizing the units in the hidden layers to speed up the learning
    * Want to avoid normalizing hidden units to have $\mu = 0$ and $\sigma = 1$ so that model can learn non-linearity. Ensure hidden units have standardized mean and variance controlled by γ and β

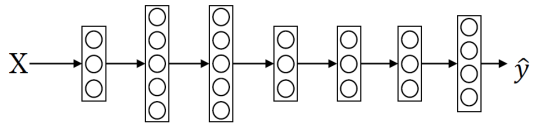
$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

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

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

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

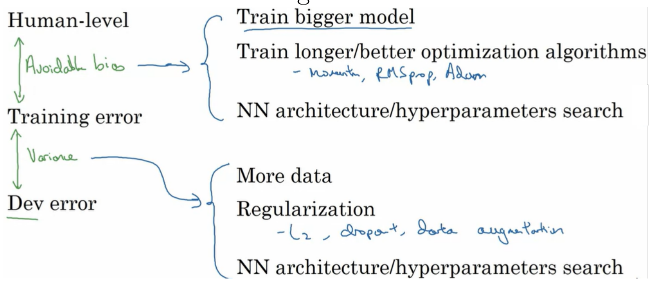
    * Slight regularization effect: makes units in later hidden units more robust to changes in earlier input and/or hidden layers.
    * What μ and σ do we use for testing? Both parameters are estimated using an exponentially weighted average across mini-batches used during training.
  - Softmax regression
    * Generalizes logistic regression to multiple classes



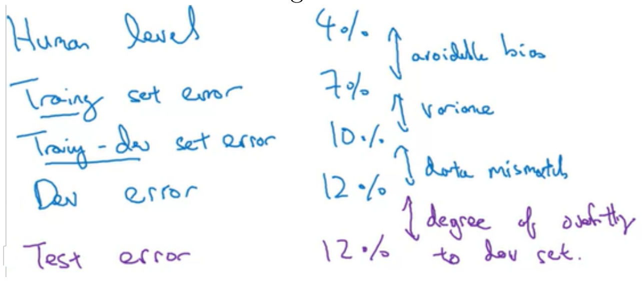
    * Activation function:
      $t = e^{z^{[L]}}$
      $a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{c} t_j}$
    * Loss: $\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{c} y_j log\hat{y}_j$
    * Cost: $J(w^{[1]}, b^{[1]}, ...) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

3 Structuring Machine Learning Projects

- Having a **single number evaluation metric** can improve decision making (i.e. F1 score for precision and recall)
- If there are multiple things, pick one **optimizing metric subject to a satisfying metric** (i.e. maximize accuracy subject to <= false positives)
- If dev and test set are from different distributions, consider shuffling them together and partitioning into both sets (i.e. otherwise throwing darts at a bullseye that's now in another location). It's okay for the training and dev distributions to be different.
- In the era of big data, the old rule of thumb of 70/30 for train/test no longer applies. The trend has been to **use more data for training and less for dev and test sets**.
- **Bayes optimal error - the theoretical best possible error** - it is unachievable. Human error is a proxy for Bayes error. Compare human-level error to training error to determine if you need to reduce bias.
  Avoidable bias = Training Error - Human Error.

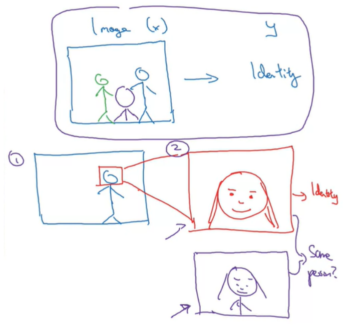

Human-level
Avoidable bias
Training error
Variance
Dev error
Train bigger model
Train longer/better optimization algorithms
- momentum, RMSprop, Adam
NN architecture/hyperparameters search
More data
Regularization
- L2, dropout, data augmentation
NN architecture/hyperparameters search

- Error Analysis - Look at misclassifications to understand where most of the misclassifications are coming from so you can focus your attention on that
- **Build your first system quickly and then iterate**
  - Set up dev/test set and metric
  - Build initial system quickly
  - Use Bias/Variance and Erorr Analysis to prioritize next steps
- Data Mismatch - Algorithm does well on training data but not on dev data from a different distribution.
  - Do manual error analysis to understand difference between training and dev/test sets
  - Generate more training data similar to dev sets

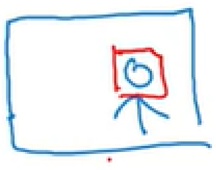


- Transfer Learning
  - Remove last layer and its weights and create new layer(s) relevant to new problem. Retrain the last few layer(s) or all if you have enough data.
  - Pre-training - initial training phase where model learns general features
  - Fine-tuning - updating weights on a pre-training model (either all or a subset of its layers)
  - When to use?
    * Task A and Task B have the same input x
    * A lot more data for Task A than Task B
    * Low level features from A could be helpful for learning B
- Multi-task learning
  - Allows you to train one neural network to do many tasks and can give better performance than if you were to do the tasks in isolation
  - $\mathcal{L}(\hat{y_j}^{(i)}, y_j^{(i)}) = -y_j^{(i)} log\hat{y_j}^{(i)} - (1 - y_j^{(i)}) log(1 - \hat{y_j}^{(i)})$
    $Cost = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{C}\mathcal{L}(\hat{y_j}^{(i)}, y_j^{(i)})$
  - When to use?
    * 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 training a big enough neural network to do well on all of the tasks

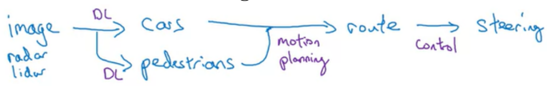
- Transfer Learning is used a lot more than Multi-task Learning (computer vision is the one major exception)
- End-to-end Deep Learning
  - Several data processing systems require multiple stages of processing. End-to-end deep learning takes all of those stages and replaces it with a single neural network. **Not always useful**
  - Not work well
    - Face recognition



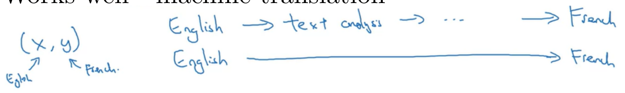
- Image of skeleton to predict age of person



- Autonomous driving



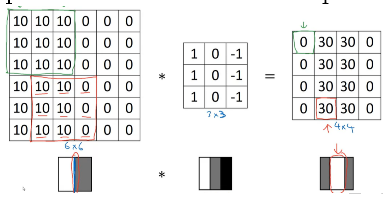
  - Works well - machine translation



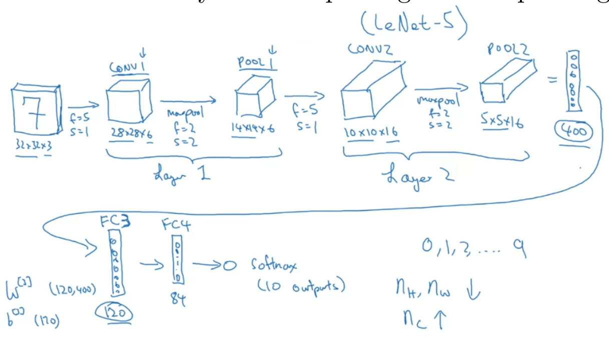
  - When to use?
    - ∗ Pros:
      - Let the data speak
      - Less hand-designing of components needed
    - ∗ Cons:
      - May need large amount of data
      - Excludes potentially useful hand-designed components

4 Convolutional Neural Networks

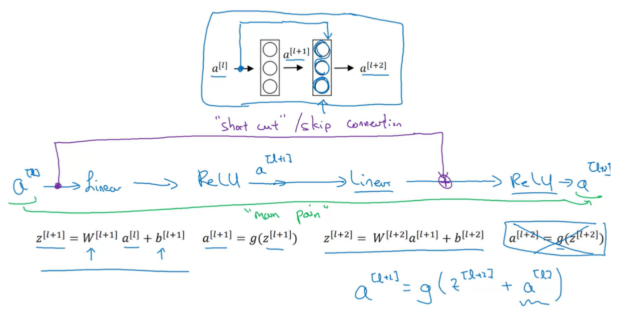
- **4.1 Introduction**
  - Convolutional layer
    - ∗ Valid convolution - combining two functions to form a third function, padding is 0.
    - ∗ Edge Detection - when you have bright pixels on the left and dark pixels on the right



    - ∗ Color scale is [0, 255]. 255 - most intense color, 0 - least intense color
    - ∗ Padding - Add additional rows/columns to make the output size the same as the input size
    - ∗ Mathematicians call the above cross-correlation because true convolution includes a flip operation. But in practice, we don't flip the filter.
    - ∗ Output size: $(n, n, n_c) * (f, f, n_c) -> (\lfloor\frac{n+2p-f}{s} + 1\rfloor, \lfloor\frac{n+2p-f}{s} + 1\rfloor, n_c^{'})$ where n is height, f is width, p is padding, s is stride, n_c is number of channels, $n_c^{'}$ is number of filters.
    - ∗ Convolution is critical because it extracts features from an image by applying filters that slide across the image, effectively identifying important patterns like edges, textures, and corners at different locations in the image, which is key to understanding spatial relationships between pixels
  - Pooling layer
    - ∗ Max Pooling - take the maximum value in the filter. High values mean the network has detected a particular feature in that quadrant in the image.
    - ∗ Hyperparameters: $f - filtersize, s - stride$
    - ∗ Most commonly use max pooling with no padding

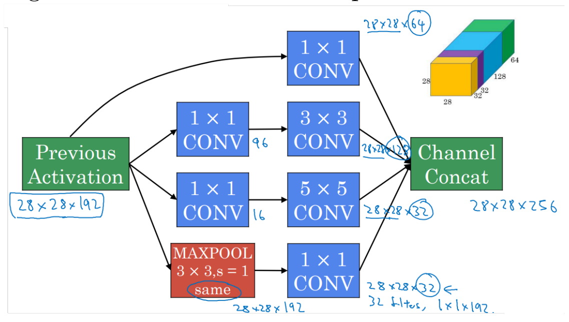


- **4.2 Case Studies**
  - LeNet-5
    - ∗ 60,000 parameters
    - ∗ $n_H \downarrow, n_W \downarrow, n_C \uparrow$
  - AlexNet
    - ∗ Similar to LeNet, but much bigger and deeper. 60 million parameters.
    - ∗ Used ReLU activation and multiple GPUs to train the model.
  - VGG-16
    - ∗ 16 layers that have weights. 138 million parameters
    - ∗ Simplified neural network architecture to have **same combination of CONV and POOL layers**. Thereby reducing the number of hyperparameters to specify ahead of time
    - ∗ CONV = 3x3 filter, s = 1, same padding
    - ∗ MAX-POOL = 2x2 filter, s = 2
  - ResNet
    - ∗ Uses skip connections (aka residual blocks) to **train very deep neural networks** without worrying too much about vanishing/exploding gradients
    - ∗ Skip connections take the activations of one layer and feed it to another layer even much deeper in the neural network

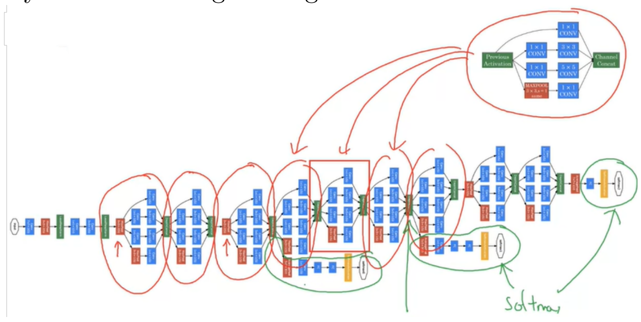


$a^{[l]}$
$a^{[l+1]}$
$a^{[l+2]}$
"short cut" / skip connection
$a^{[l]}$ → Linear → ReLU $a^{[l+1]}$ → Linear → ReLU → $a^{[l+2]}$
"main path"
$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$
$a^{[l+1]} = g(z^{[l+1]})$
$z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]}$
$\cancel{a^{[l+2]} = g(z^{[l+2]})}$
$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

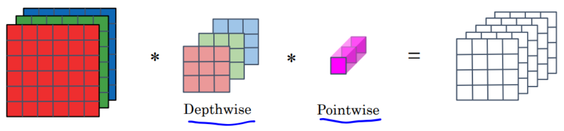
- 1 x 1 convolutions
  - ∗ 1 x 1 convolutions are used to reduce the number of channels in a layer. They are computationally efficient and do not affect the height and width of the output volume.
- Inception network
  - ∗ Lets you do all modules in one layer (CONV, POOL, 1x1, 5x5, ...) and then concatenate them together. This creates an inception module.



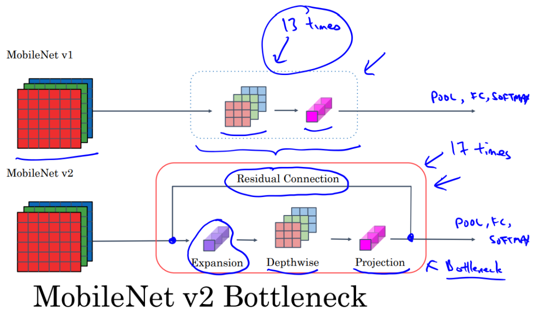
  - ∗ Stacking inception modules coupled with three softmax classifiers at intermediary layers creates a regularizng effect



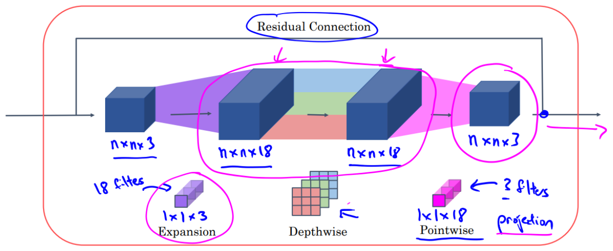
  - ∗ Can reduce computational cost significantly by using a 1x1 convolution to create a bottleneck layer.
- MobileNet
  - ∗ Computational cost = #filter params x #filter positions x #filters
  - ∗ **Low computational cost at deployment**
  - ∗ Utilizes the Depthwise-separable convolution



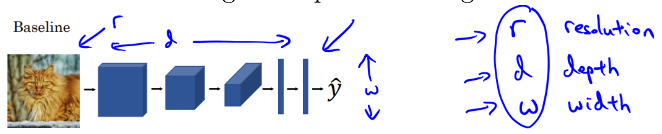
    - · depthwise convolution - applies a single filter to each input channel, unlike a regular convolution which applies multiple filters to each input channel.
    - · pointwise convolution - applies a 1x1 convolution to combine the outputs of the depthwise convolution.
  - ∗ MobileNet v2 uses an expansion layer to increase the number of channels before the depthwise convolution. This allows the network to learn more complex features.



MobileNet v2 Bottleneck



- EfficientNet - given a fixed computational budget, EfficientNet scales up all dimensions of depth, width, and resolution using a compound scaling method.



Baseline
r
d
ŷ
w
r resolution
d depth
w width