Deep learning: The more data, the better the performance.
ReLU: Makes Gradient descent faster than tanh or sigmoid.
Logistic regression: algorithm for binary classification.
Input matrix: $(n_x \times m)$; Output matrix: $(n_y \times m)$.
Parameters W: $(n^{[l-1]}, n^{[l]})$; Bias b: $\left(n^{[l]}, 1\right)$
Output: $g(W^T x + b)$. For logistic regression, $\sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$
Loss function: per training sample; cost function: all samples + regularization
Logistic regression loss function (binary cross-entropy): $\mathcal{L}(\hat{y}, y) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$
Cost function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$
Gradient descent: $w := w - \alpha \frac{dJ(w,b)}{dw}$. Probes contour of J but not guaranteed termination at global minima unless convex.
Derivative of sigmoid function: $\sigma(1-\sigma)$
Backpropagation: $dZ = A - Y; dW = \frac{1}{m} X dZ^T; dv = \frac{1}{m} np.sum(dZ); update\ w, b$
Number of layers: Do not count output layer.
Derivative of tanh: $1 - (\tanh(z))^2$
Randomized initialization: Initializing weights as zero makes all hidden layers identical (fine for logistic regression though). Bias can be zero. W can be $np.random.randn((a,b))$

- $Z^{[l]}$ and $A^{[l]}$ have size activations $(n^{[l]}, m)$.
- $W^{[l]}$ is a matrix of weights $\left(n^{[l]}, n^{[l-1]}\right)$
- $b^{[l]}$ is a vector of biases $\left(n^{[l]}, 1\right)$

Derivative versions of above four have same dimensions
Deep representation because this breaks down signals into hierarchies. Circuit theories: Need exponentially less hidden units if you have deep NN.
Note on backward propagation: For vectorization over multiple training examples:

- $dZ^{[l]} = dA^{[l]} * g'^{[l]}\left(Z^{[l]}\right)$
- $dw^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$
- $db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepdims = True)$
- $dA^{[l-1]} = w^{[l]T} \cdot dZ^{[l]}$

Choice of activation function, momentum and minibatch size are hyperparameters.
High bias: from underfitting → bigger network, train longer, NN architecture search.
High variance: from overfitting → more data; regularization, NN architecture search.
There is no longer tradeoff between bias and variance.
L_2 regularization: $J(w,b) = \frac{1}{f}\sum_{i}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}||w||_F^2$ → Smaller magnitudes of weights. A.k.a. weight decay.

$$w_i^{k+1} = w_i^k - 2\alpha\lambda w_i - \alpha\frac{\partial J}{\partial w_i}$$

L_1 regularization: $J(w,b) = \frac{1}{f}\sum_{i}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}||w||_1$ → Sparse network. $w_i^{k+1} = w_i^k - \alpha\lambda sign(w_i) - \alpha\frac{\partial J}{\partial w_i}$

- λ is called the regularization parameter (lambd in our code)

Regularization prevents overfitting.
Dropout Regularization: Say we have *l=3,* and *keep_prob* = 0.8.

- d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob → Results in matrix of Booleans
- a3 = np.multiply(a3, d3)
- a3 /= keep_prob (inverted dropout)

Don't do dropout during test time. Works because NN does not rely on one feature and spreads out weight.
Data augmentation: Fake examples don't add much, but useful.
Early stopping: Get intermediate size weights. Similar effect as L2 regularization to some extent.

- Combined cost function optimization and not overfitting, so not always to good (keep things orthogonal)

Normalizing inputs: to avoid elongated contours.

- Zero Out mean: $\mu = \frac{1}{m}\sum x_i$ and $X := X - \mu$
- Normalize by Variance: $\sigma^2 = \frac{1}{f}\sum^{n} x_i^{i\,2}; X := X/\sigma^2$

Vanishing and exploding gradients: When there are multiple hidden layers.

- Weight initialization: Partial solution to vanishing/exploding gradients.
  In python: $W^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$
- Xavier initialization for tanh: $\sqrt{\frac{1}{n^{[l-1]}}}$ (This is standard deviation) Other variant: $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$

Gradient checking: $\epsilon \approx 10^{-7}$ is a good starting point value for epsilon. Check $\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2}$
Note: gradient checking does not work with dropout.
Mini-batch gradient descent: Too big, too long per iteration. Too small, lost speedup from vectorization.
Exponentially weighted averages: $v_0 = 0$ and $v_t = \beta v_{t-1} + (1-\beta)\theta_t$. Bias correction: $\frac{v_t}{1-\beta^t}$
Gradient descent with momentum: $v_{dW} = \beta_1 v_{dW} + (1-\beta_1) dW$ and also for db.
RMSprop: Reduces oscillation. $S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2; W := W - \alpha\frac{dW}{\sqrt{S_{dw}}+\epsilon}$
Adam: Momentum + RMSprop.
Learning rate decay: Reduce oscillation as learning approaches convergence. $\alpha = 0.95^{epoch\ num}\alpha_0$
Hyperparameter tuning: First tune α, momentum (β_1), number of layers, and learning rate decay. Then number of units in layer, and then mini-batch size. $\beta_1, \beta_2, \epsilon$ are rarely tuned.
Random hyperparameters: $r = -4 * np.random.rand()$ gives r within [-4, 0]; $\alpha = 10^r$ to get 0.0001, …, 1
Babysitting one model: low on computational resources. Many models: With high resources (Caviar)
Batch normalization: normalizing activation in a network. After simply normalizing, $\tilde{Z}^{(i)} = \gamma Z_{norm}^{(i)} + \beta$
$\gamma\ and\ \beta$ are parameters here. Works well with momentum, RMSprop, or Adam.

- Weights in future networks are more robust to weights in previous network, so can learn independently
- Each minibatch is scaled by its mean and variance. This adds noize to $z^{[l]}$ and thus adds a slight regularization effect.
- At test time, use exponentially weighted average across minibatches for each layer.

Softmax Regression: Multiple classification. Activation is $a_i^{[l]} = \frac{e^{z^{[l]}}}{\sum t_i} = \frac{t_i}{\sum t_i}$. Results in linear decision boundaries. Softmax is a generalization of logistic regression but with more classes. Loss

function is: $\mathcal{L}(\hat{y}, y) = -\sum_j y_j \log \hat{y}_j$ and cost function is $J(w^{[1]}, b^{[l]}, \dots) = \frac{1}{m}\sum \mathcal{L}(\hat{y}, y)$

Orthogonalization: Tune one thing and change only the target. First define metric. Then worry about doing well
If training set does not fit well, try better optimization algorithm; If dev set does not fit well, try regularization or getting a larger training and dev sets; If does well on test set, but app does not well, change either the dev set or cost function.
Steps of ML: Fit training set well on cost function, then dev set, and then test set.
Single number evaluation metric: precision = (true positive)/(all positive) no mistaken alarms are allowed; recall = (true positive)/(true positive + false negative) for cases when no misses are allowed; F1 scorer = harmonic mean of precision and recall.
Satisficing → Meet the bar; Optimizing → The better the better.
Dev set also called cross validation set. Dev & test sets should always come from the same distribution.
Metrics: Can give weight to false positive or true negatives.
If app does not do well in practice but does well in dev/test set, change metric or dev/test set.
Human error can often be used to approximate Bayes error since humans are actually quite good.
It is hard to improvement beyond human level error
Reducing avoidable bias: train bigger model, train longer, better optimization algorithms (Adam), NN architecture/hyperparameters search (RNN, CNN).
Reducing variance: More data, regularization (L2, dropout, data augmentation), NN architecture/hyperparameters
Data analysis: Look at mislabeled samples. Fix systematic errors. Random errors should be fine.
When correcting, fix dev and test set. Build system quickly and then iterate (metric, system, and error analysis)
Mismatched training and dev/test set: Give half of dev/test to training. Create training-dev set.
Avoidable bias ←→ Data mismatch problem ←→ Variance problem
Manual error analysis to address data mismatch → Collect more data via data synthesis
Transfer learning: Pretraining and fine tuning. Makes sense when you have small data.
Multi-task learning → Simultaneously do task for shared features. $J = \frac{1}{m}\sum^{m}\sum^{num\ Tasks}\mathcal{L}\left(\hat{y}_j^{(i)}, y_j^{(i)}\right)$. Makes sense when amount of data per task is similar.
End-to-end deep learning: Let's the data speak.
CNN: Popular for vision problems. Reduces the number of weights. A (1000x1000x3) images connected to 1,000 units results in (1000x1000x3x1000) weights. Use Spatial context
Edge detection filter: Convolve kernel to image. In practice, we don't handpick the filters and let the NN learn. E ach filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$. I nput: $n_H^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$. Output is $n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$. $n_w^{[l]}$ and $n_H^{[l]}$ are $\left\lfloor\frac{n+2p-f}{s}+1\right\rfloor$. Bias has size $\left(1,1,1,n_c^{[l]}\right)$.
Valid filter: Smaller size. Same filter: Same output size.
Pooling: Reduce size of representation and speed up computation. Size and strides are hyperparameters. Needs to cache how images were pooled for back propagation. Output dimensions: $\left\lfloor\frac{n_H+f}{s}+1\right\rfloor \times \left\lfloor\frac{n_H+f}{s}+1\right\rfloor \times n_c$.
Number of parameters: $\left(f \times f + 1(bias)\right) \times (numb\ filters)$.
Much larger if fully connected. Parameter sharing makes the process effective but increases bias. Sparsity of connections: Only nearby pixels matter.
Case studies: Back in the days, tanh was popular. Nowadays ReLU is popular.
ResNet: $a^{[l+1]} = g\left(z^{[l+1]}\right)$; $z^{[l+2]} = w^{[l+2]}a^{[l+1]} + b^{[l+2]}$; $a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right)$. Empirically fixes the issue of plain networks performing worse when NN gets deeper. Identity function is an easy function to learn.
1x1 convolution: Helps a lot to condense channels and add non-linearity to the network (network in a network).
Inception network: Instead of choosing filter size, first do 1x1 convolution and then do 3x3, 5x5 filters.
Transfer learning: Decide how much to transfer, freeze preexisting network and how much to add.
Data augmentation: Almost always help. Mirror, random, crop. Also rotation, shear, local warping. Done in CPU before feeding data to GPU. Reduces overfitting
Tips for doing well: Ensembling (train on many networks then average output). Costly. For competitions.
Multi-crop at test time: costly. For competition. Use open source code as pretrained models and fine-tune on your data set.
Neural Style Transfer: Minimize: $\mathcal{L} = \left|\left|Style_S - Style_G\right|\right|_2^2 + \left|\left|Content_C - Content_G\right|\right|_2^2$
Trigger word detection: $\mathcal{L} = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$
Adversarial examples: Create image. Loss from past midterm: $\frac{1}{2}\left|\left|f(x) - \hat{y}_{STOP}\right|\right|^2 + \lambda\left|\left|x - x_{NOPARK}\right|\right|^2$. Optimize image x via $x = x - \alpha\partial L/\partial x$
State space of real images is small. Defense is to create safety net, train on advesarial examples, or do adversarial training: $\mathcal{L}_{new} = \mathcal{L}(W, b, x, y) + \lambda\mathcal{L}(W, b, x_{adv}, y)$. Adversarial logit pairing: $\mathcal{L}_{new} = L(W, b, x, y) + \lambda\left|\left|f(x; W, b) - f(x_{adv}; W, b)\right|\right|_2^2$
White-box attack: Know wieghts and biases. ↔ Black box
GAN: generator G makes fake image. Discriminator D classifies (0 if from G, 1 otherwise). Run Adam simultaneously on two minibatches (true and generated data). Update D every iteration. G every k iterations.

$$J^{(D)} = -\frac{1}{m_{real}}\sum y_{real}^{(i)} \log\left(D\left(x^{(i)}\right)\right) - \frac{1}{m_{gen}}\sum\left(1 - y_{gen}^{(i)}\right)\log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

Saturating: $J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}}\sum \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$

Non-saturating: $J^{(G)} = -\frac{1}{m_{gen}}\sum \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$

End goal: G outputs images that are indistinguishable from ream images for D (probability of D guessing right is 50%)
Evaluating GANs: Human annotators, inception score (measures quality and diversity)
Data augmentation can hurt if there is a class imbalance or incorrect label.
Fast Gradient Sign Method: to generate adversarial examples that make NN robust against perturbations
Other random stuff:

- Learning rate decay can cause sudden drop in cost
- Try to overfit 1 training example to check your code
- Better to not penalize outliers too much.