Notation

superscript (i) denotes the i^{th} example
superscript $[l]$ denotes the l^{th} layer
m = number of examples in the dataset
n_x = input size
n_y = output size/number of classes
$n^{[l]}$ = number of units in the l^{th} layer
L =
number of layers in the network, hidden and output
$x^{(i)} \in \mathbb{R}^{n_x}$ =
i^{th} example represented as a column vector
$X \in \mathbb{R}^{n_x \times m}$ = input matrix
$y^{(i)} \in \mathbb{R}^{n_y}$ = output label for the i^{th} example
$Y \in \mathbb{R}^{n_y \times m}$ = label matrix
$Z^{[l]} \in \mathbb{R}^{n^{[l]} \times m}$ = linear bits of layer l
$A^{[l]} \in \mathbb{R}^{n^{[l]} \times m}$ = activations of layer l
$W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$ = weight matrix for layer l
$b^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}$ = bias vector for layer l
$\hat{y} = a^{[L]} \in \mathbb{R}^{n_y}$ predicted output vector

Introduction to Deep Learning

- AI is the new Electricity. Electricity had once transformed countless industries. AI will now bring about an equally big transformation.
- Supervised learning is by far the most important kind of learning, where you go from input x to output y
- Structured data is when you have a database of data. Unstructured data is stuff like audio, images, and text; features are words and pixel values; harder to make sense of unstructured data.
- Traditional learning methods don't get much better with more and more data

Basics of Neural Network Programming

Logistic Regression for One Example

Forward Propagation:
$z^{(i)} = w^T x^{(i)} + b$
$\hat{y}^{(i)} = a^{(i)} = \sigma(z^{(i)}) = \frac{1}{1+e^{-(w^T x+b)}}$
Cross Entropy Loss:
$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)})$
Cost: Computed by summing over all training examples:
$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$
Backward Propagation:
$\frac{\partial J}{\partial w} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}(a^{(i)} - y^{(i)})$
$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$

Logistic Regression Vectorized

Forward Propagation:
$Z = w^T X + b$
$\hat{Y} = A = \sigma(Z) = \sigma(w^T X + b)$
$\mathcal{L}(A, Y) = -Y * \log(A) - (1-Y) * \log(1-A)$
Cost: Computed by summing over all training examples:
$J(w,b) = \frac{1}{m} \sum \mathcal{L}(A, Y)$
$J(w,b) = -\frac{1}{m} \sum (Y * \log(A) + (1-Y) * \log(1-A))$
Backward Propagation:
$\frac{\partial J}{\partial w} = \frac{1}{m} \sum X(A-Y)^T$
$\frac{\partial J}{\partial b} = \frac{1}{m} \sum (A-Y)$

Gradient Descent

Want to find (w, b) that minimize J(w, b), which is a convex function. Gradient descent goes the steepest direction downwards.
$w := w - \alpha \frac{\partial J}{\partial w}$
$b := b - \alpha \frac{\partial J}{\partial b}$

Cross Entropy Loss Function

If $y = 1$, then $p(y|x) = \hat{y}$. If $y = 0$, then $p(y|x) = 1 - \hat{y}$. Therefore, $p(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$. Take the log likelihood:
$\log p(y|x) = \log(\hat{y}^y (1-\hat{y}^{1-y})) =$
$y \log \hat{y} + (1-y) \log(1-\hat{y}) = -\mathcal{L}(\hat{y}, y)$. We want to minimize the loss, so we maximize the log likelihood estimation.

One Hidden Layer Neural Network

Formulation for a Single Example

Forward Propagation:
$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$
$a^{[1](i)} = g^{[1]}(z^{[1](i)})$
$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$
$\hat{y}^{(i)} = a^{[2](i)} = g^{[2]}(z^{[2](i)})$

$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Cost
$J =$
$-\frac{1}{m} \sum_{i=0}^{m} (y^{(i)} \log(a^{[2](i)}) + (1-y^{(i)}) \log(1-a^{[2](i)}))$
Backward Propagation:
$dz^{[2]} = a^{[2]} - y$
$dW^{[2]} = dz^{[2]} a^{[1]}$
$db^{[2]} = dz^{[2]}$
$da^{[1]} = W^{[2]^T} dz^{[2]}$
$dz^{[1]} = dz^{[1]} * g'^{[1]}(z^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$
$db^{[1]} = dz^{[1]}$

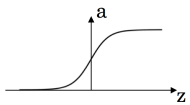
Vectorized Formulation

Forward Propagation:
$Z^{[1]} = W^{[1]} X + b^{[1]}$
$A^{[1]} = g^{[1]}(Z^{[1]})$
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
$\hat{Y} = A^{[2]} = g^{[2]}(Z^{[2]})$
Cost
$J = -\frac{1}{m} \sum (Y * \log(A^{[2]}) + (1-Y) * \log(1-A^{[2]}))$
Backward Propagation:
$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$
$db^{[2]} = \frac{1}{m} \sum dZ^{[2]}$
$dA^{[1]} = W^{[2]^T} dZ^{[2]}$
$dZ^{[1]} = dA^{[1]} * g'^{[1]}(Z^{[1]})$
$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = \frac{1}{m} \sum dZ^{[1]}$

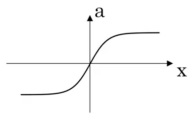
Need for Non-Linear Activations

If we have linear activation functions, then our output is just a linear combination of the input features. Therefore, no matter how many layers we include, we would just get a standard linear regression.

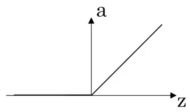
Activation Functions



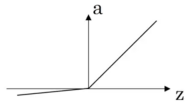
sigmoid
$a = \frac{1}{1+e^{-z}}$
$g'(z) = a(1-a)$



tanh
$a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
$g'(z) = 1 - a^2$



ReLU
$a = \max(0, x)$
$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & z \geq 0 \end{cases}$



Leaky ReLU
$a = \max(0.01x, x)$
$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & z \geq 0 \end{cases}$

Random Weight Initialization

- In logistic regression it wasn't important to initialize the weights randomly, while in a neural network with more than one layer we have to initialize them randomly.
- If we initialize all the weights with zeros, then on each gradient descent iteration all the hidden units will always update the same way. This means all hidden units will be completely identical and won't be able to break symmetry.
- We initialize weights with small random values in [0, 0.01].
- If weights are too large, then we end up on the edges of tanh or sigmoid activation function, so learning is slow.
- We can initialize biases to 0 and still break symmetry.

Deep Neural Networks

Why Deep Representations?

There are functions you can compute with a small L-layer deep neural network that shallower networks require exponentially more hidden units to compute. If we have a network with n inputs and n, $\frac{n}{2}$, $\frac{n}{4}$, $\ldots$, 1 units, the total depth is $O(\log n)$. To be able to compute the same types of functions we would need $O(2^n)$ units in a single layer neural network.

Forward and Backward Propagation

Forward Propagation:
$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
$A^{[l]} = g^{[l]}(Z^{[l]})$
Backward Propagation:
$dZ^{[l]} = dA^{[l]} * g'^{[l]}(Z^{[l]})$
$dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]^T}$
$db^{[l]} = \frac{1}{m} \sum dZ^{[l]}$
$dA^{[l-1]} = W^{[l]^T} dZ^{[l]}$

Parameters versus Hyperparameters

- Hyperparameters control the algorithm and therefore determine the value of the parameters.
- Parameters: weights, biases
- Hyperparameters: number of iterations, number of hidden layers, number of units in each layer, activation functions, minibatch size, regularization,

Setting Up Your ML Application

Train, Dev, and Test Data Sets

- Build model using train set then optimize hyperparameters on dev set. Once final model is selected, then evaluate on test set.
- For small data (100 to 10,000 examples), split 60-20-20.
- For large data (1,000,000+ examples), split 98-1-1.
- Make sure the dev and test set are coming from the same distribution, but it's okay if the training set comes from a different distribution.

Bias and Variance

We want low bias and low variance. High bias means model is underfit. High variance means model is overfit.

Train Error	1%	15%	15%	0.5%
Dev Error	11%	16%	30%	1%
Bias	Low	High	High	Low
Variance	High	Low	High	Low

For high bias, we can train a larger network or perform an architecture search. For high variance, we can use more data, add regularization, or perform an architecture search.

Regularizing Your Neural Network

L1 and L2 Cost Function

L1 matrix norm:
$||W||_1 = \sum_{j=1}^{n_x} |w(j)|$
L2 matrix norm (a.k.a. Frobenius norm):
$||W||_2^2 = W^T W = \sum_{j=1}^{n_x} w(j)^2$
Cost:
$J(w,b) = \frac{1}{m} \sum_{i=1}^{n} (\mathcal{L}(\hat{y}^{(i)}, y^{(i)})) + \frac{\lambda}{2m} \sum_{l=1}^{L} ||W^{[l]}||_2^2$
Then $dW^{[l]} =$ (from backprop) $+ \frac{\lambda}{m} W^{[l]}$.
Substitute this into the update rule and we get $W^{[l]} := (1 - \frac{\alpha\lambda}{m}) W^{[l]} - \alpha$ (from backprop). The new term $(1 - \frac{\alpha\lambda}{m})$ causes the weight to decay in proportion to its size.

Why Regularization Reduces Overfitting

- If λ is too large, W's will be close to zero which will use the linear part of the tanh activation function. We will go from non-linear to roughly linear activation which will make the NN simpler (behave closer to logistic regression).
- If λ good, it will just reduce some weights that make the neural network overfit by making some of the tanh activations roughly linear.

Dropout Regularization

Why Dropout Works: Dropout randomly eliminates units and their connections on each iteration. 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. Prevents weights from relying too heavily on any single feature. Do not use dropout at test time.
Inverted Dropout: Maintains the expected value of $A^{[l]}$.
In forward propagation, perform each layer after computing A and include D in cache:

```
D1 = np.random.rand(A1.shape[0], A1.shape[1]) <
    ↪ keep_prob
A1 = A1 * D1
A1 / keep_prob
```

In backward propagation, immediately after computing dA:

```
dA2 = dA2 * D2
dA2 = dA2 / keep_prob
```

Other Regularization Methods:

- Data Augmentation: Transform data in a way that makes sense for that data set. Could rotate, crop, reflect, alter perspective, etc. Classic example of what not to do is rotating 6 or 9 when trying to do number recognition.

- Early Stopping: Plot train and dev set errors over iterations. Stop training when the dev set error starts to increase as the train error continues to decrease.

Setting Up Your Optimization Problem

Normalizing Inputs

Calculate mean and variance of train data, then apply $\frac{x-\mu}{\sigma^2}$ transformation to train, dev, and test data sets.

$\mu = \frac{1}{m}\sum x^{(i)}$

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

Normalize inputs because we want our parameter search space to be as symmetric as possible. If we don't normalize, we could get a shallow cost function, leading to smaller gradients and slower learning.

Vanishing and Exploding Gradients

When gradients are too small or too large, activations (and gradients) will be decreased/increased exponentially as a function of number of layers. If $W > I$ (identity matrix), the activation and gradients will explode. If $W < I$, the activation and gradients will vanish. We need to be careful about how we initialize our weights.

Weight Initialization

Xavier Initialization: Use with tanh activation. Multiply random $[0, 1]$ weights for layer l by $\sqrt{\frac{1}{n^{[l-1]}}}$.

He Initialization: Use with ReLU activation. Multiply random $[0, 1]$ weights for layer l by $\sqrt{\frac{2}{n^{[l-1]}}}$.

Gradient Approximation and Checking

Can be useful to numerically approximate gradients using $\frac{f(\theta+\epsilon)-f(\theta-\epsilon)}{2\epsilon}$ to ensure that algorithm is working as expected while debugging. Don't use during training. Doesn't work with dropout.

To check gradients, reshape $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$ into one long vector θ. Approximate gradient using $f(\theta) = J(\theta)$. Subtract $\epsilon = 10^{-7}$ from each parameter. Then check $\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2}$. If result is $\approx 10^{-7}$ that's great, should be worried if $\approx 10^{-7}$.

Optimization Algorithms

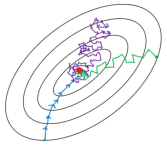
Mini-Batch Gradient Descent

Training NN with a large data set can be slow. Sometimes memory isn't large enough to fit all data in at once, or using the entire data set for a single iteration of gradient descent would take longer than it's worth. We can split our m training examples into mini-batches of a specified size (usually a power of 2) and small enough that it fits into CPU/GPU. We denoted our mini-batch number by $t = 1, 2, \ldots, \lceil\frac{m}{\text{mini-batch size}}\rceil$, and use $X^{\{t\}}$, $Y^{\{t\}}$, and $J^{\{t\}}$ to denote the examples, labels, and cost for mini-batch t. One pass through the entire data set is called an epoch.

Batch vs. Mini-Batch vs. Stochastic Gradient Descent

- Stochastic gradient descent has mini-batch size of 1 (purple). Often too noisy regarding cost minimization and lose speedup from vectorization.

- Batch gradient descent has mini-batch size of m (blue). Sometimes too long per iteration.



- Mini-batch gradient descent has mini-batch size in $(1, m)$ (green). Faster learning while maintaining some advantage from vectorization. Doesn't always converge, but can reduce learning rate when getting close.

Exponentially Weighted Averages

We calculate exponentially weighted averages across time points t where $v_0 = 0$ and $v_t = \beta v_{t-1} + (1-\beta)\theta(t)$. Then, v_t is the average over approximately the last $\frac{1}{1-\beta}$ time points. If we want an average over time points, then exponentially weighted averages are better than sliding window averages because they require less memory. However, starting with $v_0 = 0$ biases our average. To correct for this, we divide our calculation of v_t by $1 - \beta^t$ yielding $v_t = \frac{\beta v_{t-1} + (1-\beta)\theta(t)}{1-\beta^t}$.

Gradient Descent with Momentum

Momentum helps the cost function to go to the minimum point in a more fast and consistent way.

$V_{dW} = \beta * V_{dW} + (1-\beta)dW$

$V_{db} = \beta * V_{db} + (1-\beta)db$

$W := W - \alpha V_{dW}$

$b := b - \alpha V_{db}$

In practice, don't use bias correction because it's resolved in ≈ 10 iterations. Usually use $\beta = 0.9$.

RMSprop

Also helps the cost function to go to the minimum point in a more fast and consistent way. With RMSprop you can increase your learning rate.

$S_{dW} = \beta_2 * V_{dW} + (1-\beta_2)(dW * dW)$

$S_{db} = \beta_2 * V_{db} + (1-\beta_2)(db * db)$

$W := W - \alpha\frac{dW}{\sqrt{S_{dW}+\epsilon}}$ where $epsilon = 10^{-8}$

$b := b - \alpha\frac{db}{\sqrt{S_{db}+\epsilon}}$ where $epsilon = 10^{-8}$

Adam Optimizer

Adaptive Moment Estimation combined ideas from momentum and RMSprop. It tends to work very well.

$V_{dW} = \beta * V_{dW} + (1-\beta_1)dW$

$V_{dW}^{cor} = \frac{V_{dW}}{1-\beta_1^t}$

$V_{db} = \beta * V_{db} + (1-\beta_1)db$

$V_{db}^{cor} = \frac{V_{db}}{1-\beta_1^t}$

$S_{dW} = \beta_2 * V_{dW} + (1-\beta_2)(dW * dW)$

$S_{dW}^{cor} = \frac{S_{dW}}{1-\beta_2^t}$

$S_{db} = \beta_2 * V_{db} + (1-\beta_2)(db * db)$

$S_{db}^{cor} = \frac{S_{db}}{1-\beta_2^t}$

$W := W - \alpha\frac{V_{db}^{cor}}{\sqrt{S_{dW}^{cor}+\epsilon}}$ where $epsilon = 10^{-8}$

$b := b - \alpha\frac{V_{db}^{cor}}{\sqrt{S_{db}^{cor}+\epsilon}}$ where $epsilon = 10^{-8}$

Hyperparameters usually $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and α needs to be tuned.

Learning Rate Decay

Idea is to slowly reduce learning rate across iterations so the oscillations near the optimum are smaller. Some techniques that are applied across epochs are:

- $\alpha = \frac{\alpha_0}{1+\text{decay\_rate}*\text{epoch\_num}}$
- $\alpha = 0.95^{\text{epoch\_num}}\alpha_0$
- $\alpha = \frac{k}{\sqrt{\text{epoch\_num}}}\alpha_0$ or $\alpha = \frac{k}{\sqrt{k}}\alpha_0$
- Discretely halving α at end of each epoch
- Manual decay only if traiming a small number of models

Problem of 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 is much more likely to get to the saddle point. Sometimes plateaus can make learning slow when gradient is close to zero for a long time, but small random perturbations can help along with algorithms like momentum, RMSprop or Adam.

Hyperparameter Tuning

- Most important to tune is α. Then mini-batch size, number of hidden units, and β. Then number of layers and learning decay rate. We almost never tune Adam optimizer parameters.

- Try random values of parameters – don't use a grid. Hard to know which parameters will be most important for your problem and what their values should be.

- Can use a coarse to fine sampling scheme. When you find some hyperparameters values that give you a better performance, zoom into a smaller region around these values and sample more densely within this space.

- Make sure to choose an appropriate scale to sample parameters from. The sensitivity of results to mall changes in parameter value is not uniform over the interval of possible values. Ex: α should be sampled on a logarithmic scale of $10^{\text{rand\_num}}$ for $[10^{-4}, 1]$. For exponentially weighted averages β use $1 - 10^{\text{rand\_num}}$.

- Like a panda, you can babysit your model while training if you have a lot of data and not a lot of computational resources. Or like caviar, you can train many models in parallel (preferred).

Batch Normalization

Batch normalization can speed up learning, as previously discussed. It is usually applied to mini-batches at every layer of the neural network. Once we have normalized $Z^{[l]}$, we can transform the data to a new distribution defined by the learnable parameters γ and β: $\tilde{Z}^{[l]} = \gamma Z^{[l]}_{\text{norm}} + \beta^{[l]}$. There are some changes to propagation.

Forward Propagation:

$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ ← drop $b^{[l]}$ because replaced by $\beta^{[l]}$

Calculate $Z^{[l]}_{\text{norm}}$

$\tilde{Z}^{[l]} = \gamma Z^{[l]}_{\text{norm}} + \beta^{[l]}$

$A^{[l]} = g^{[l]}(\tilde{Z}^{[l]})$

Backward Propagation:

Only update parameters $W^{[l]}$, $\gamma^{[l]}$, and $\beta^{[l]}$.

Regularization Effect of Batch Norm: 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. This has a slight regularization effect. Using bigger size of the mini-batch you are reducing noise and therefore regularization effect. Don't rely on batch normalization as a form of regularization. It's intended for normalization of hidden units, activations and therefore speeding up learning.

Batch Norm 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 weighted average across the mini-batches. We will use the estimated values of the mean and variance to test.

Multi-Class Classification

There is a generalization of logistic regression called softmax regression that is used for multi-class classification. In this case, C = number of classes, the range of classes is $[0, C-1]$, and $N_y = C$. The equations for softmax activation are:

$t = e^{Z^{[l]}}$ where $t \in \mathbb{R}^{c \times m}$

$A^{[l]} = \frac{e^{Z^{[l]}}}{\sum_i^{C-1} t_{(i)}}$ ← scale to 1 by example

Loss and Cost Function

$L(y, \hat{y}) = -\sum_{j=0}^{C-1} y_j \log(\hat{y}_j)$

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

Introduction to ML Strategy

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, we want to tune the knobs of our system to improve each of these accuracies one at a time:

- Fit train set well on cost function (near human level performance if possible).

- Fit dev set well on cost function.

- Fit test set well on cost function.

- erforms well in real world.

Setting Up Your Goal

Single Number Evaluation Metric

- Its better and faster to set a single number evaluation metric for your project before you start it.

- Precision = TP / (TP + FP) and recall = TP / (TP + FN), but better to combine these metrics in to F1 score = 2 / [(1/P) + (1/R)].

- Its hard sometimes to get a single number evaluation metric. So we can solve that by choosing a single optimizing metric and decide that other metrics are satisficing.

- If doing well on your metric and not doing well in your application, change your metric and/or dev and test sets.

Train, Dev, and 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 the target you want to aim at.

Comparing to 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.

Bayes Optimal Error: Bayes Optimal Error is the theoretical best possible error we can achieve. Usually we use human error as a proxy for Bayes optimal error because humans are fairly good at a lot of tasks. As long as your model is worse than humans, you can: get labeled data from humans, gain insight from manual error analysis, and better analyze bias and variance.

Avoidable Bias: Defined as the error between train error and human error. Methods for improving avoidable bias are the same as for improving bias.

Mismatches Training and Dev/Test Data

If train data is from a different distribution than the dev and test sets, then we can get data mismatch as a source of differences in accuracy between the train and dev sets.

Learning from Multiple Tasks

Transfer learning makes sense if:

- Task A and B have the same input x.
- You have a lot more data for Task A than Task B.
- Low level features from A could be helpful for learning B.

Hyperparameters for transfer learning are: layers to keep, layers to add, and parameters to freeze or re-tune.

End-to-End Deep Learning

Pros of end-to-end deep learning:

- Let the data speak. By having a pure machine 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).

Applying 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?
- Use ML/DL to learn some individual components.
- 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.

Convolutional Neural Networks

- Parameter sharing: A feature detector (such as a vertical edge detector) thats useful in one part of the image is probably useful in another part of the image.
- Sparsity of connections: In each layer, each output value depends only on a small number of inputs.

Padding: Can be valid or same. Valid means no padding. Same means output will have same n as input. To accomplish this, we choose $p = \frac{f-1}{2}$.

Dimensions:

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = stride

$n_C^{[l]}$ = number of filters

Each filter is: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$

Activations are: $a^{[l]} = n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

or: $A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

Weights are: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$

Biases are: $n_C^{[l]}$

Input is: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$

Output is: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

$n_H^{[l]} = \lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$

$\text{num}_p arams = (\text{f}^{[l]} * f^{[l]} * n_C^{[l-1]} + 1) * n_C^{[l]}$

1×1 Convolutions: Allow you to shrink or expand the number of channels in an image (like pooling does for n_W and n_H). Takes element-wise product of a pixel's channel. Sometimes called a network in network. Can be used as a bottleneck to reduce number of multiplication operations.

Inception Network: Allows you to apply many different types of filters instead of trying to choose between them. Must use same padding for convolutions.

Resideual Networks:

Skips over layers and adds in activation from first layer in block to z of last layer in block. Adding layers to a ResNet doesn't hurt because it is easy for residual blocks to learn the identity function, and it still gives them the chance to learn a more complex function if it will boost performance. Assumes $a^{[l+2]}$ and $a^{[l]}$ have the same dimension, so it's common to use same padding on convolutions, but if not you can add W_s to change the shape of $a^{[l]}$. Here $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$.

Generative Adversarial Networks

$$J^{(D)} = \underbrace{-\frac{1}{m_{real}} \sum_{i=1}^{m_{real}} y_{real}^{(i)} \cdot \log(D(x^{(i)}))}_{\text{cross-entropy 1: "D should correctly label real data as 1"}} \underbrace{-\frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} (1 - y_{gen}^{(i)}) \cdot \log(1 - D(G(z^{(i)})))}_{\text{cross-entropy 2: "D should correctly label generated data as 0"}}$$

$$J^{(G)} = -\frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log(D(G(z^{(i)}))) \quad \text{"G should try to fool D: by minimizing this"}$$

Python Examples

Logistic Regression

```
def sigmoid(z):
  return 1 / (1 + np.exp(-(z)))
def initialize_with_zeros(dim):
  w = np.zeros((dim,1))
  b = 0
  return w, b
def propagate(w, b, X, Y):
  m = X.shape[1]
  A = sigmoid(np.dot(w.T, X) + b)
  cost = -(1/m) * np.sum(Y * np.log(A) + (1 - Y) *
      ↪  np.log(1 - A))
  dw = (1/m) * np.dot(X, (A - Y).T)
  db = (1/m) * np.sum(A - Y)
  cost = np.squeeze(cost)
  grads = {"dw": dw, "db": db}
  return grads, cost
def optimize(w, b, X, Y, num_iterations,
      ↪ learning_rate, print_cost = False):
  costs = []
  for i in range(num_iterations):
    grads, cost = propagate(w, b, X, Y)
    dw = grads["dw"]
    db = grads["db"]
    w = w - learning_rate*dw
    b = b - learning_rate*db
    if i % 100 == 0:
      costs.append(cost)
    if print_cost and i % 100 == 0:
      print ("Cost␣after␣iteration␣%i:␣%f" %(i,
      ↪ cost))
  params = {"w": w, "b": b}
  grads = {"dw": dw, "db": db}
  return params, grads, costs
def predict(w, b, X):
  m = X.shape[1]
  Y_prediction = np.zeros((1,m))
  w = w.reshape(X.shape[0], 1)
  A = sigmoid(np.dot(w.T, X) + b)
  for i in range(A.shape[1]):
    Y_prediction[0,i] = 1 if A[0,i] > 0.5 else 0
  assert(Y_prediction.shape == (1, m))
  return Y_prediction
def model(X_train, Y_train, X_test, Y_test,
      ↪ num_iterations = 2000, learning_rate =
      ↪ 0.5, print_cost = False):
  w, b = initialize_with_zeros(X_train.shape[0])
  parameters, grads, costs = optimize(w, b,
      ↪ X_train, Y_train, num_iterations,
      ↪ learning_rate, print_cost = print_cost)
  w = parameters["w"]
  b = parameters["b"]
  Y_prediction_test = predict(w, b, X_test)
  Y_prediction_train = predict(w, b, X_train)
  print("train␣accuracy:␣{}␣\%".format(100 - np.
      ↪ mean(np.abs(Y_prediction_train - Y_train)
      ↪ ) * 100))
  print("test␣accuracy:␣{}␣\%".format(100 - np.
      ↪ mean(np.abs(Y_prediction_test - Y_test))
      ↪ * 100))
  d = {"costs": costs,
    "Y_prediction_test": Y_prediction_test,
    "Y_prediction_train" : Y_prediction_train,
    "w" : w,
    "b" : b,
    "learning_rate" : learning_rate,
    "num_iterations": num_iterations}
  return d
```

Deep Neural Network

```
def initialize_parameters_zeros(layers_dims):
  parameters = {}
  L = len(layers_dims)
  for l in range(1, L):
    parameters['W' + str(l)] = np.zeros((
      ↪ layers_dims[l], layers_dims[l-1]))
    parameters['b' + str(l)] = np.zeros((
      ↪ layers_dims[l], 1))
  return parameters
def initialize_parameters_random(layers_dims):
  parameters = {}
  L = len(layers_dims)
  for l in range(1, L):
    parameters['W' + str(l)] = np.random.randn(
      ↪ layers_dims[l],layers_dims[l-1]) * 10
    parameters['b' + str(l)] = np.zeros((
      ↪ layers_dims[l], 1))
  return parameters
def initialize_parameters_he(layers_dims):
  parameters = {}
  L = len(layers_dims) - 1
  for l in range(1, L + 1):
    parameters['W' + str(l)] = np.random.randn(
      ↪ layers_dims[l],layers_dims[l-1]) * np.
      ↪ sqrt(2/layers_dims[l-1])
    parameters['b' + str(l)] = np.zeros((
      ↪ layers_dims[l], 1))
  return parameters
def linear_forward(A, W, b):
  Z = np.dot(W, A) + b
  cache = (A, W, b)
  return Z, cache
def linear_activation_forward(A_prev, W, b,
      ↪ activation):
  if activation == "sigmoid":
    Z, linear_cache = linear_forward(A_prev, W, b)
    A, activation_cache = sigmoid(Z)
  elif activation == "relu":
    Z, linear_cache = linear_forward(A_prev, W, b)
    A, activation_cache = relu(Z)
  cache = (linear_cache, activation_cache)
  return A, cache
def L_model_forward(X, parameters):
  caches = []
  A = X
  L = len(parameters) // 2
  for l in range(1, L):
    A_prev = A
    A, cache = linear_activation_forward(A_prev,
      ↪ parameters["W" + str(l)], parameters["b"
      ↪ + str(l)], "relu")
    caches.append(cache)
  AL, cache = linear_activation_forward(A,
      ↪ parameters["W" + str(L)], parameters["b"
      ↪ + str(L)], "sigmoid")
  caches.append(cache)
  return AL, caches
def compute_cost(AL, Y):
  m = Y.shape[1]
  cost = -(1/m) * np.sum(Y * np.log(AL) + (1-Y) *
      ↪ np.log(1-AL))
  cost = np.squeeze(cost)
  return cost
def linear_backward(dZ, cache):
  A_prev, W, b = cache
  m = A_prev.shape[1]
  dW = (1/m) * np.dot(dZ, A_prev.T)
  db = (1/m) * np.sum(dZ, axis=1, keepdims=True)
  dA_prev = np.dot(W.T, dZ)
  return dA_prev, dW, db
def linear_activation_backward(dA, cache,
      ↪ activation):
  linear_cache, activation_cache = cache
  if activation == "relu":
    dZ = relu_backward(dA, activation_cache)
    dA_prev, dW, db = linear_backward(dZ,
      ↪ linear_cache)
  elif activation == "sigmoid":
    dZ = sigmoid_backward(dA, activation_cache)
    dA_prev, dW, db = linear_backward(dZ,
      ↪ linear_cache)
  return dA_prev, dW, db
def L_model_backward(AL, Y, caches):
  grads = {}
  L = len(caches) # the number of layers
  m = AL.shape[1]
  Y = Y.reshape(AL.shape)
  dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 -
      ↪  AL))
  current_cache = caches[L-1]
  grads["dA" + str(L-1)], grads["dW" + str(L)],
      ↪ grads["db" + str(L)] =
```

```
    ↪ linear_activation_backward(dAL,
    ↪ current_cache, "sigmoid")
  for l in reversed(range(L-1)):
    current_cache = caches[l]
    dA_prev_temp, dW_temp, db_temp =
    ↪ linear_activation_backward(grads["dA" +
    ↪ str(l + 1)], current_cache, "relu")
    grads["dA" + str(l)] = dA_prev_temp
    grads["dW" + str(l + 1)] = dW_temp
    grads["db" + str(l + 1)] = db_temp
  return grads
def update_parameters(parameters, grads,
    ↪ learning_rate):
  L = len(parameters) // 2 # number of layers in
    ↪ the neural network
  for l in range(L):
    parameters["W" + str(l+1)] = parameters["W" +
    ↪ str(l+1)] - learning_rate * grads["dW" +
    ↪ str(l+1)]
    parameters["b" + str(l+1)] = parameters["b" +
    ↪ str(l+1)] - learning_rate * grads["db" +
    ↪ str(l+1)]
  return parameters
def random_mini_batches(X, Y, mini_batch_size =
    ↪ 64):
  m = X.shape[1]
  mini_batches = []
  permutation = list(np.random.permutation(m))
  shuffled_X = X[:, permutation]
  shuffled_Y = Y[:, permutation].reshape((1,m))
  num_complete_minibatches = math.floor(m/
    ↪ mini_batch_size)
  for k in range(0, num_complete_minibatches):
    mini_batch_X = shuffled_X[:, k *
    ↪ mini_batch_size : (k+1) * mini_batch_size
    ↪ ]
    mini_batch_Y = shuffled_Y[:, k *
    ↪ mini_batch_size : (k+1) * mini_batch_size
    ↪ ]
    mini_batch = (mini_batch_X, mini_batch_Y)
    mini_batches.append(mini_batch)
  if m % mini_batch_size != 0:
    mini_batch_X = shuffled_X[:, (k+1) *
    ↪ mini_batch_size : m+1]
    mini_batch_Y = shuffled_Y[:, (k+1) *
    ↪ mini_batch_size : m+1]
    mini_batch = (mini_batch_X, mini_batch_Y)
    mini_batches.append(mini_batch)
  return mini_batches
def L_layer_model(X, Y, layers_dims, learning_rate
    ↪  = 0.0075, num_epochs = 10000,
    ↪ mini_batch_size = 64, initialization = "
    ↪ he", print_cost = False):
  # Implements [LINEAR->RELU]*(L-1)->LINEAR->
    ↪ SIGMOID.
  costs = []
  if initialization == "zeros":
    parameters = initialize_parameters_zeros(
    ↪ layers_dims)
  elif initialization == "random":
    parameters = initialize_parameters_random(
    ↪ layers_dims)
  elif initialization == "he":
    parameters = initialize_parameters_he(
    ↪ layers_dims)
  for i in range(num_epochs):
    minibatches = random_mini_batches(X, Y,
    ↪ mini_batch_size)
    for minibatch in minibatches:
      (minibatch_X, minibatch_Y) = minibatch
      AL, caches = L_model_forward(minibatch_X,
    ↪ parameters)
      cost = compute_cost(AL, minibatch_Y)
      grads = L_model_backward(AL, minibatch_Y,
    ↪ caches)
      parameters = update_parameters(parameters,
    ↪ grads, learning_rate)
      if print_cost and i % 100 == 0:
        print ("Cost␣after␣iteration␣%i:␣%f" %(i,
    ↪ cost))
      if print_cost and i % 100 == 0:
        costs.append(cost)
  plt.plot(np.squeeze(costs))
  plt.ylabel('cost')
  plt.xlabel('iterations␣(per␣tens)')
  plt.title("Learning␣rate␣=" + str(learning_rate)
    ↪ )
  plt.show()
  return parameters
```

Convolutional Neural Network

```
def zero_pad(X, pad):
  return np.pad(X, ((0,0), (pad, pad), (pad, pad),
    ↪  (0,0)), 'constant')
def conv_single_step(a_slice_prev, W, b):
  return np.sum(a_slice_prev * W) + float(b)
def conv_forward(A_prev, W, b, hparameters):
  (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
  (f, f, n_C_prev, n_C) = W.shape
  stride = hparameters["stride"]
  pad = hparameters["pad"]
  n_H = int((n_H_prev - f + 2 * pad) / stride) + 1
  n_W = int((n_W_prev - f + 2 * pad) / stride) + 1
  Z = np.zeros((m, n_H, n_W, n_C))
  A_prev_pad = zero_pad(A_prev, pad)
  for i in range(m):
    a_prev_pad = A_prev_pad[i,:,:,:]
    for h in range(n_H):
      for w in range(n_W):
        for c in range(n_C):
          vert_start = h * stride
          vert_end = vert_start + f
          horiz_start = w * stride
          horiz_end = horiz_start + f
          a_slice_prev = a_prev_pad[vert_start :
    ↪ vert_end, horiz_start : horiz_end, :]
          Z[i, h, w, c] = conv_single_step(
    ↪ a_slice_prev, W[:, :, :, c], b[:, :, :, c
    ↪ ])
  assert(Z.shape == (m, n_H, n_W, n_C))
  cache = (A_prev, W, b, hparameters)
  return Z, cache
def pool_forward(A_prev, hparameters, mode = "max"
    ↪ ):
  (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
  f = hparameters["f"]
  stride = hparameters["stride"]
  n_H = int(1 + (n_H_prev - f) / stride)
  n_W = int(1 + (n_W_prev - f) / stride)
  n_C = n_C_prev
  A = np.zeros((m, n_H, n_W, n_C))
  for i in range(m):
    for h in range(n_H):
      for w in range(n_W):
        for c in range (n_C):
          vert_start = h * stride
          vert_end = vert_start + f
          horiz_start = w * stride
          horiz_end = horiz_start + f
          if mode == "max":
            A[i, h, w, c] = np.max(a_prev_slice)
          elif mode == "average":
            A[i, h, w, c] = np.mean(a_prev_slice)
  cache = (A_prev, hparameters)
  assert(A.shape == (m, n_H, n_W, n_C))
  return A, cache
def conv_backward(dZ, cache):
  (A_prev, W, b, hparameters) = cache
  (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
  (f, f, n_C_prev, n_C) = W.shape
  stride = hparameters["stride"]
  pad = hparameters["pad"]
  (m, n_H, n_W, n_C) = dZ.shape
  dA_prev = np.zeros((m, n_H_prev, n_W_prev,
    ↪ n_C_prev))
  dW = np.zeros((f, f, n_C_prev, n_C))
  db = np.zeros((1, 1, 1, n_C))
  A_prev_pad = zero_pad(A_prev, pad)
  dA_prev_pad = zero_pad(dA_prev, pad)
  for i in range(m):
    a_prev_pad = A_prev_pad[i, :, :, :]
    da_prev_pad = dA_prev_pad[i, :, :, :]
    for h in range(n_H):
      for w in range(n_W):
        for c in range(n_C):
          vert_start = h * stride
          vert_end = vert_start + f
          horiz_start = w * stride
          horiz_end = horiz_start + f
          a_slice = a_prev_pad[vert_start :
    ↪ vert_end, horiz_start : horiz_end, :]
          da_prev_pad[vert_start:vert_end,
    ↪ horiz_start:horiz_end, :] += W[:,:,:,c] *
    ↪  dZ[i, h, w, c]
          dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
          db[:,:,:,c] += dZ[i, h, w, c]
    X[pad:-pad, pad:-pad, :])
    dA_prev[i, :, :, :] = da_prev_pad[pad:-pad,
    ↪ pad:-pad, :]
  assert(dA_prev.shape == (m, n_H_prev, n_W_prev,
    ↪ n_C_prev))
  return dA_prev, dW, db
def create_mask_from_window(x):
  return (x == np.max(x))
def distribute_value(dz, shape):
  (n_H, n_W) = shape
  average = dz / (n_H * n_W)
  a = np.ones(shape) * average
  return a
  def pool_backward(dA, cache, mode = "max"):
  (A_prev, hparameters) = cache
  f = hparameters["f"]
  m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
  m, n_H, n_W, n_C = dA.shape
  dA_prev = np.zeros((m, n_H_prev, n_W_prev,
    ↪ n_C_prev))
  for i in range(m):
    a_prev = A_prev[i, :, :, :]
    for h in range(n_H):
      for w in range(n_W):
        for c in range(n_C):
          vert_start = h * stride
          vert_end = vert_start + f
          horiz_start = w * stride
          horiz_end = horiz_start + f
          if mode == "max":
            a_prev_slice = a_prev[vert_start :
    ↪ vert_end, horiz_start : horiz_end, c]
            mask = create_mask_from_window(
    ↪ a_prev_slice)
            dA_prev[i, vert_start: vert_end,
    ↪ horiz_start: horiz_end, c] += mask * dA[i
    ↪ , h, w, c]
          elif mode == "average":
            da = np.mean(dA[i, h, w, c])
            shape = (f, f)
            dA_prev[i, vert_start: vert_end,
    ↪ horiz_start: horiz_end, c] +=
    ↪ distribute_value(da, shape)
  assert(dA_prev.shape == A_prev.shape)
  return dA_prev
```