Course Name: Deep Learning Specialization (by DeepLearning.AI, on Coursera) **Instructor: Dr. Andrew Ng**

Couse Summary

Deep Neural Network

	Motivations/Concepts	NN Structure and Forward Prop	Back Prop and Gradient Descent	Numpy Script	Notes	Applications
Deep Neural Network	• NN: a series of algorithms, to recognize underlying relationships in data, mimicking the way the human brain operates. • CNN (Convolutional): transforms data through filter before activation, good for image processing • RNN (recuring): interpret temporal or sequential information	Training set: $\boldsymbol{X-n_x \times m}$; $\boldsymbol{Y-1 \times m}$; NN structure: $\boldsymbol{L-\#layers}$, *L0* is input layer Parameters: $\boldsymbol{W^{[l]}-n_l \times n_{l-1}}$; $\boldsymbol{b^{[l]}-n_l \times 1}$ Forward prop: Linear Representation: $Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$ Followed by activation function: $A^{[l]} = g(Z^{[l]})$ Cost Function: $J(W,b) = -\frac{1}{m}\sum[Y \odot \log(Y_{hat}) + (1-Y) \odot \log(1-Y_{hat})]$	Backprop from last L: $dZ^{[L]} = A^{[L]} - Y$ $dZ^{[l]} = dA^{[l]} \cdot g'(Z^{[l]}) = W^{[l+1]^T} dZ^{[l+1]} g'(Z^{[l]})$ $dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]^T}$; $db^{[l]} = \frac{1}{m}\sum dZ^{[l]}$ Notes: • During backprop, cache the grads dW, db of each layer • During forward prop, we need to cache the W, b, A of each layer in order to compute the backprop	*## define the NN structure and initialize parameters* parameters = initialize_parameters(layers_dims) for i in range(0, num_iterations): *# Forward propagation: compute output, cache $A^{[l]}$ and $Z^{[l]}$* AL, caches = forward_propagation(X, parameters) *# Compute cost, append to a list* cost += compute_cost(AL, Y) *# Backward propagation: cache dW, db* grads = backward_propagation(AL, caches, parameters) *# Update parameters: use gradient descent* parameters = update_parameters(parameters, grads)	• Deep Learning key factors: data, computation, algorithms • Vectorization works better in GPU • Broad casting is an important aspect of numpy matrix operations • By changing NN structure, bias/variant can be improved independently without hurting the other	Standard NN: real estate, online advertising CNN: photo tagging RNN: speech recognition, machine translation Customized Hybrid: autonomous driving
Activation Function	• Without activation function, the NN degenerated to a linear correlation between input and output (high bias)	Sigmoid: $g(z) = \frac{1}{1+e^{-z}}$ Hyperbolic tangent: $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ ReLU (rectified linear unit): $g(z) = \max(0, z)$ Leaky ReLU: $g(z) = \max(0.01 * z, z)$	Sigmoid: $g'(z) = g(z)(1-g(z))$ Hyperbolic tangent: $g'(z) = 1 - g(z)^2$ ReLU (rectified linear unit): $g'(z) = 0\ if\ z < 0;\ g'(z) = 1\ of\ z \geq 0$ Leaky ReLU: $g'(z) = 0.01\ if\ z < 0;\ g'(z) = 1\ of\ z \geq 0$	Two main changes: *# Forward propagation: activation is a list of act.Func. names used in each layer* AL, caches = forward_prop (X, parameters, activation) *# Backward propagation:* grads = backward_prop (AL, caches, parameters, activation)	• Circuit theory: If a "small" deep NN is replaced by a shallow NN, the hidden units will be exponentially increased (e.g. 1L of 2^{n-1} hidden units can represent all the possible non-linear combinations of n input units)	Typically use ReLU for all hidden layers, and use Sigmoid (or softmax) for last layer of binary (or categorical) classification
Regularization	• To solve high variant issue • If λ is sufficiently large, three benefits: 1. w is small, as if reduce # hidden units 2. act.Func. is close to center, training faster 3. weight decay due to the extra gradient term, training faster	Frobenius Norm: $\|\|W\|\|_F = \sum\sum w_{ij}^2$ Cost Function: $J(W,b) = -\frac{1}{m}\sum[Y \odot \log(Y_{hat}) + (1-Y) \odot \log(1-Y_{hat})] + \frac{\lambda}{2m}\sum_{layer}\sum\sum W^2$ Gradient Decay: $W := W - \alpha\left(backprop\ term + \frac{\lambda}{m}W\right) = W\left(1 - \frac{\alpha\lambda}{m}\right) - \alpha(backprop\ term)$ *move faster*	Backprop: $dZ^{[L]} = A^{[L]} - Y$ $dZ^{[l]} = dA^{[l]} \cdot g'(Z^{[l]}) = W^{[l+1]^T} dZ^{[l+1]} g'(Z^{[l]})$ $dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]^T} + \frac{\lambda}{m}W^{[l]}$; $db^{[l]} = \frac{1}{m}\sum dZ^{[l]}$	Forward prop does not change. Cost function, back prop, and parameter updates will need changes (λ – hyperparameter of regularization). compute_cost_with_regularization (AL, Y, lambda) backward_prop (AL, caches, parameters, lambda) update_parameters(parameters, grads, lambda)	Other regularization methods: • Data augmentation (e.g. rotation, distortion, mirroring) • Early stopping: plot J_train and J_dev v.s. iteration and find the elbow (W is initialized small, the ||w|| is still small while early stopping)	Computer vision: input size is so big, never have enough data, always tend to over fitting
Dropout Regularization	• To solve high variant issue • Randomly choose the node to eliminate for forward/back prop during each iteration on gradient descent • Intuition: shrink weights (don't rely heavily on any feature)	Forward prop: $Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$ $A^{[l]} = g(Z^{[l]}) \odot D^{[l]}/keep_{prob}$ *where $D^{[l]}$ is to randomly eliminate some nodes* • We need to cache A, Z, and D of each layer • Divide/keep_prob, is to bump up the unit values so that to keep the magnitude of output values	Backprop: $dZ^{[L]} = A^{[L]} - Y$ $dZ^{[l]} = dA^{[l]} \cdot g'(Z^{[l]}) = W^{[l+1]^T} dZ^{[l+1]} \odot D^{[l]} \odot g'(Z^{[l]})$ $dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]^T}$; $db^{[l]} = \frac{1}{m}\sum dZ^{[l]}$	Forward prop and back prop will change forward_prop_with_dropout (X, parameters, keep_prob) backward_prop_with_dropout (AL, caches, parameters, keep_prob) *where keep_prob is the probability to keep a node* $D^{[l]} = \text{np.random.rand}(A^{[l]}.shape[0], A^{[l]}.shape[1]) < keep\_prob$	• Drop out parameter D needs to be randomly initialized in each iteration • Keep_prob value can vary by layers • Cost function may not decrease after each iteration, validate full NN before turn on "drop off"	

Mini-Batch Gradient Descent	• To speed up the vector computation when data set size m is very large
	• In each ***epoch*** (a single pass through the training set), loop over each mini batch
		Mini-batch size: $\boldsymbol{n_t}$
		(Stochastic gradient descent: $n_t = 1$, this loses the speed up from vectorization)
		Forward prop:
		$Z^{[l]\{t\}} = W^{[l]} \cdot A^{[l-1]\{t\}} + b^{[l]}$
		$A^{[l]\{t\}} = g(Z^{[l]\{t\}})$
			Backprop:
			$dZ^{[L]\{t\}} = A^{[L]\{t\}} - Y^{\{t\}}$
			$dZ^{[l]\{t\}} = dA^{[l]\{t\}} \cdot g'(Z^{[l]\{t\}}) = W^{[l+1]^T} dZ^{[l+1]\{t\}} \odot g'(Z^{[l]\{t\}})$
			$dW^{[l]} = \frac{1}{m} dZ^{[l]\{t\}} \cdot A^{[l-1]\{t\}^T}$; $db^{[l]} = \frac{1}{m}\sum dZ^{[l]\{t\}}$
				parameters = initialize_parameters(layers_dims)
				for i in range(0, num_iterations):
				shuffle training set sequence
				for j in range(0, m/n_t):
				AL, caches = forward_prop (X[:, j*n_t:(j+1)*n_t], parameters)
				cost += compute_cost(AL, Y[:, j*n_t:(j+1)*n_t])
				grads = backward_prop (AL, caches, parameters)
				parameters = update_parameters(parameters, grads)
					• For batch-GD, one epoch takes one gradient step, while min-batch GD allows to take m/n_t gradient steps
					• Mini-batch GD does not guarantee reduce of cost function in each iteration
						Use Mini-Batch to speed up the gradient descent in order to speed up the vector computation when data set size m is very large

Exponentially Weighted Average	• A moving average technic • The beginning few data points may need ***bias correction***	Equation: $V_0 = 0,\ V_t = \beta V_{t-1} + (1-\beta)\theta_t$ Where Vt is approx. the average over $\frac{1}{(1-\beta)}$ Bias correction: $\frac{V_t}{1-\beta^t}$	Derivation: $V_t = (1-\beta)\theta_t + \beta(1-\beta)\theta_{t-1} + \beta^2(1-\beta)\theta_{t-2} + \cdots$ $\lim(1-\epsilon)^{1/\epsilon} = \frac{1}{e}$	--	--	The hyperparameters (high priority first): *# Learning rate: α* *# Momentum term: β* *# layers, mini-batch size* *# hidden units, Learning rate decay* *# Adam terms: β1, β2, ϵ (0.9, 0.999, 10^-8)* Tunning Process: *# try random values instead of grid* *# course to fine* *# choose appropriate scale (e.g. log)*
Adam Algorithm	• Act at moving average of gradient descent, to allow larger learning rate • Include two parts: momentum term, RMSprop (root-mean squared)	Momentum term: $V_t = \beta V_{t-1} + (1-\beta)\theta_t$ RMS-prop term: $S_t = \beta S_{t-1} + (1-\beta)\theta_t^2$ • Forward prop does not change • Typically choose: $\beta_1 = 0.9; \beta_2 = 0.999; \epsilon = 10^{-8}$	Back prop: $V_{dW^{[l]}} = \beta_1 V_{dW^{[l]}} + (1-\beta_1) dW^{[l]}$; $V_{dW^{[l]}}^{corrected} = \frac{V_{dW^{[l]}}}{1-\beta_1^t}$ $S_{dW^{[l]}} = \beta_2 S_{dW^{[l]}} + (1-\beta_2) dW^{[l]^2}$; $S_{dW^{[l]}}^{corrected} = \frac{S_{dW^{[l]}}}{1-\beta_2^t}$ $W^{[l]} := W^{[l]} - \alpha \frac{V_{dW^{[l]}}^{corrected}}{\sqrt{S_{dW^{[l]}}^{corrected}} + \epsilon}$; $b^{[l]} := b^{[l]} - \alpha \frac{V_{db^{[l]}}^{corrected}}{\sqrt{S_{db^{[l]}}^{corrected}} + \epsilon}$	• Follow mini-batch gradient descent steps • In function "backward_prop", instead of cache dW, db, cache V_{dw}, S_{dw}, V_{db}, S_{db} • "update_parameters" function also needs update	• The adam algo is to smooth out the gradient descent in mini-batch, since the cost function may show zig-zag	
Learning Rate Decay	• Slow down the learning rate when mini-batch GD approaches local min	Equation: $\alpha = \frac{1}{1+decay-rate*epoch\#}\alpha_0$	--	--	• In high dimensional optimization, it's easy to get stuck at saddle points. Also, Plateaus can also slow down learning • Mitigation: random initialize multi-times	
Batch Normalization	• Normalize the input $Z^{[l]}$ of each layer to speed up learning • Batch norm makes the weights in deep NN layers more robust to weights change in earlier layers. • Mini-Batch norm has slight regularization effect (intro noise in each layer)	For layer l and unit i: $Z_{norm}^{[l](i)} = \frac{Z^{(i)} - \mu^{(i)}}{\sqrt{\sigma^2 + \epsilon}}$; $\tilde{Z}^{[l](i)} = \gamma Z_{norm}^{[l](i)} + \beta$ • γ and β are learnable parameters. • In parameter cache, we need to store W, γ and β. We don't need to store b, because β act same.	• Follow mini-batch gradient descent steps	--	• γ and β will effectively shift the mean and sigma, otherwise (u=0,sigma=1) loss the benefit of Sigmoid/ReLU • Slight regularization: no node is heavily weighted than others	--
Softmax Regression	• Multi-class classification • Each unit of output Y represents the probability of class C, e.g. $P(c \mid X, W, b)$	Activation function: $Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$; $t^{[l]} = \exp(Z^{[l]})$; $A^{[l]} = \frac{t^{[l]}}{\sum_{j=1}^{n} t_j^{[l]}}$ Cost function: $L(y_{hat}, y) = -\sum_{k=1}^{C} y_k \log(y_{hat\,k})$; $J(W,b) = \frac{1}{m}\sum_{i=1}^{m}(y_{hat}^{(i)}, y^{(i)})$	Backprop: $dZ^{[L]} = A^{[L]} - Y$	--	• Softmax function can be viewed as a multi-class logistic regression problem, and each decision boundary is more linear. • Softmax is typically used for output belonging to one class each	--

Summarizer: Renjie (Ryan) Chen **ren.j.chen@gmail.com**

ML Strategy

<table>
<tr><th></th><th>Motivation</th><th>Methodology</th><th>Note</th></tr>
<tr>
<td>Orthogonalization</td>
<td>solve one ML problem without affecting others</td>
<td>
<table>
<tr><th>Chain of Errors</th><th>Orthogonal Errors</th><th>Orthogonal Knobs</th></tr>
<tr><td>Human-level Error</td><td colspan="2">← as approx. of theoretical limit (Bayes optimal error)</td></tr>
<tr><td>Training Error</td><td>Avoidable Bias = training error – Human error</td><td>▪ Bigger model
▪ Longer/better optimization algo (momentum, RMSprop, Adam)
▪ NN architecture / hyperparameters searching (RNN, DNN)</td></tr>
<tr><td>Training-Dev Set Error</td><td>Variant Issue = Training-Dev set error – Training error</td><td>▪ More training data, data augmentation
▪ Regularization
▪ Revisit the NN architecture</td></tr>
<tr><td>Dev Error</td><td>Data mismatch Issue = Dev error - Training-Dev set error</td><td>▪ Error analysis on sev/test set
▪ Put part (e.g. half) of the dev/test set into training set</td></tr>
<tr><td>Test Error</td><td>Degree of overfitting to Dev Set = Test Error – Dev Error</td><td>▪ Larger Dev set</td></tr>
<tr><td>Real-word performance</td><td>Metric issue = Real Application Error – Test Error</td><td>▪ Reevaluate cost function, and dev/test set choices
▪ Add a weighted item in the cost function to penalize certain metrics</td></tr>
</table>
</td>
<td>▪ With very large data set, chose train/dev/test sets: 98% / 1% / 1%
▪ Training dev set is the same distribution as training set, but not used for training and only save for training-dev purpose.
▪ Early stopping is not recommended, because it tunes training set and dev set knobs simultaneously
▪ The goal of training set is to minimize the cost function, the dev set is used to guide the choice of model on training set based on optimization metrics (N matrix care about = 1 metric for optimization + N-1 metrics for satisfaction.)</td>
</tr>
<tr>
<td>Error Analysis</td>
<td>error analysis on mislabeled dev set examples and find the “ceiling” that could be improved</td>
<td>
<table>
<tr><th>Image</th><th>Dog</th><th>Great Cat</th><th>Blurry</th><th>Incorrectly labeled</th><th>Comments</th></tr>
<tr><td>...</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>98</td><td></td><td></td><td></td><td>✓</td><td>Labeler missed cat in background</td></tr>
<tr><td>99</td><td></td><td>✓</td><td></td><td></td><td></td></tr>
<tr><td>100</td><td></td><td></td><td></td><td>✓</td><td>Drawing of a cat; Not a real cat.</td></tr>
<tr><td>% of total</td><td>8%</td><td>43%</td><td>61%</td><td>6%</td><td></td></tr>
</table>
</td>
<td>▪ DL algorithms are quite robust to random errors in the training set; less robust to systematic error.
▪ If the magnitude will affect your decision on training set choice, then we need to correct it</td>
</tr>
<tr>
<td>Transfer Learning
Multi-task Learning</td>
<td>Low level features of the deep NN (e.g. detecting edges) helps learn the structure/nature of a general problem</td>
<td>Transfer learning: utilize a well-trained deep NN architecture, lock the lower levels, and modify the top levels for new output
Multi-task learning: use the same early layers of NN, train a single NN for multi-task would be more efficient than training separate NNs</td>
<td>▪ Transfer learning makes sense when: (1) Task A and B have the same input X; (2) Have more data for Task A than Task B; (3) Low level features from A could be helpful for learning B
▪ Multi-task learning makes sense when: (1) Training on a set of tasks that could benefit from having shared lower-level features; (2) Usually, amount of data you have for each task is quick similar; (3) Can train a big enough NN to do well on all the tasks</td>
</tr>
</table>

Summarizer: Renjie (Ryan) Chen **ren.j.chen@gmail.com**