Coursera deeplearning.ai - Deep Learning Specialization by Andrew NG et al.

Notes by *Ricardo Santos*

Neural Networks and Deep Learning

- The amount of data plus the ability to compute larger neural networks, allowed models to become much better.
- Change from sigmoid to ReLU made learning faster.

Logistic regression as a neural network

- Small scale example of a neural network with a single neuron (and still useful as a classifier).
- Side note:
  - d stands for "derivative" where there is only a single variable. ∂ stands for "partial derivative" where there are multiple variables.

Neural networks

- Typically the input layer is not counted as a layer when counting the layers of a neural network, only the hidden layers and the output layer are considered. The output layer is NOT a hidden layer.
- $A^{[l]}$ is the activation for layer l, $W^{[l]}$ is the weights matrix, and $b^{[l]}$ is the bias term. $g(z)$ is the chosen activation function.
- The learning rate α scales the size of the gradient descent steps, and therefore determines how fast it converges (though values too high can actually make it not converge).
- Regarding notation $a_j^{[l](i)\{k\}}$ means the activation (could be any other parameter) on the j^{th} unit/neuron of the l^{th} network layer, for the i^{th} example/data point of the k^{th} minibatch (minibatches are discussed later on).

Forward Propagation for layer l (vectorized):

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^l$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

Backward Propagation for layer l (vectorized):

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]}), \text{ note that } * \text{is the element-wise multiplication}$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]}(A^{[l-1]})^\intercal$$
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepdims = True)$$
$$dA^{[l-1]} = (W^{[l]})^\intercal dZ^{[l]}$$

Parameter Update for layer l (vectorized):

$$W^{[l]} = W^{[l]} - \alpha \cdot dW^{[l]}, \text{ where } \alpha \text{ is the learning rate}$$
$$b^{[l]} = b^{[l]} - \alpha \cdot db^{[l]}, \text{ where } \alpha \text{ is the learning rate}$$

Activation functions

- Sigmoid (or logistic function) - usually used for the output layer only (becuase it outputs either 1 or 0), and not for the hidden layers because its derivative can be close to 1 for values of z further away from the origin.

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

- Tanh is better than sigmoid for hidden layers (though it is an offset sigmoid) only because it is defined between -1 and 1, which means that for the mean of zero, the tanh will be close to 0 as well, which can be useful for computation.

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

- Rectified linear unit (ReLU) - $ReLU(z) = max(0, z)$ - is the de facto standard for linear units nowadays (most commonly used), its derivative is easy to calculate (not defined for z = 0, but we can work around that with a convention on what 0 means), and does not suffer from slow convergence of gradient descent due to derivate being close to 0.

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

- Leaky ReLU - $max(0.01 * z, z)$ - Similar to the ReLU, but instead of being zero for values of z lower than 0, it actually has a small positive slope in that section, so that its derivate is not 0, making it easier for the optimizer (e.g. gradient descent) to converge. In reality normal RelU are still the standard though.

$$LReLU(z) = \begin{cases} 0.01z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases},\ LReLU'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Why do we need some sort of activation function anyway?

- If we had non, the activation would be linear, and all layers would have linear activation function, resulting in the NN activation being itself a linear activation function, which negates the usefulness of the hidden layers. There must always be a hidden layer.

Random initialization

- Two neurons are considered symmetric if they are doing the exact same computation.
- We avoid that by using random initialization, otherwise all neurons/units will keep on being symetric trough out all backprop iterations.
- b (bias) doesn't require random initialization like the weights W.
- W should be initialized to small random values (multiply by 0.01 for example), to allow faster convergence of gradient descent with sigmoid or tanh activation functions.

Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Training/Dev(cross-validation)/Test

- Training set is used to train the model's parameters
- Dev(cross-validation) is used to train the model's hyperparameters and check model's performance while in development.
- Test set is an unbiased set of data that was never seen by the model. Some teams may not use one of this, only a dev set instead.
- Traditionally with small data the split between these two sets would be either 70/30% (train/test(or dev)) or 60/20/20% (train/dev/test). With big data that can be something like 98/1/1%.
- if the training data is from a different distribution than the test set, then it is recommended that the dev set should belong to the same distribution as the test set.

Bias and Variance

- High bias generally means underfitting - high error on the training set
- High variance generally means overfitting - high error on the test set
- Base error is the reference (e.g. human) error for the same task, and should be compared to the model to determine high variance or high bias.
- There used to be a tradeoff between these 2 but that is not so discussed in the scope of deep learning, because we can always increase the network or/and add more data.
- high bias and high variance can occur at the same time if the model underfits some parts of the model and overfits other parts.

Solution for high bias:

- Bigger network (KEY) - does not cause high variance (with good regularization)
- Train longer
- (change NN arquitecture)

Solution for high variance:

- More training data (KEY) - does not cause high bias
- Regularization
- (change NN arquitecture)

Regularization

- **Regularization** is also called **weight decay**, because it cause weights to be smaller for higher values of lambda (the regularization parameter). It reduces **high variance**
- There is L2 and L1 regularization, L2 uses the squared of the weights, L1 uses only the norm and has the "advantage" of making the weights matrix sparse, though L2 is the most used in practice.
- There is usually no regularization of the bias term because it is just a constant.
- **Dropout** regularization consists on training the classifier with a number of neurons "switched off" at every training iteration (though not during testing). It has a similar effect to regularization, and it is possible to have different percentages of droped units/neurons for each layer, making it more flexible. The same units/neurons are droped in both forward and backward steps. The cost function with dropout does not necessarily decrease continuouly as we usually see for gradient descent.
  - **Inverted dropout** is the most common type of dropout, and it consists of scalling activations by dividing with the activation matrix with **keep_prob** (the probability of keeping units), for each layer.

- Early stoping consists on stoping training when the error of the network is the lowest for the dev(cross-validation) data set, even if it can still be decrease for the training set.
- Other regularization examples include **data augmentation** (e.g. adding to the training set flipped images of others already in it).
- Orthogonalization is the separation of the cost optimization step (e.g. gradient descent) from steps taken for not overfiting the model (e.g. regularization), in other words, optimizing model's parameters vs optimizing model hyperparameters.

Normalizing training sets

- consists of, for each feature, subtracting the mean and dividing by the variance.
  - The mean and variance obtained in the training set should be used to scale the test set as well, (we don't want to scale the training set differently).
  - Allows using higher learning rates for gradient descent.

Vanishing / Exploding gradients

- In very deep networks (depending on the activation function) weights greater than 1 can make activations exponentially larger depending on the number of layers with such weights, whereas weights smaller than 0 can make activations exponentially smaller, depending on the number of layers with such small weights (think in terms of a very deep network with linear activations as intuitive example).
  - The above is also applicable for the grandients (not just the activation/output), in the opposite direction (backword propagation), thus gradients can either explode (causing numerical instability) or become very small (with the consequence of lower layers not to being updated as well as numerical instability). Hence the designation of **Vanishing/Exploding gradients**.
  - Partial solution - make the randomly initialized weights to have a variance of $\frac{1}{n^{[l-1]}}$, though for ReLU it is usually $\frac{2}{n^{[l-1]}}$, and for tanh is $\sqrt{\frac{1}{n^{[l-1]}}}$ (Xavier initialization) though sometimes an alternative $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$ is also used.
  - tunning the variance of the weight initializations has usually just a moderate effect, except for specific cases.

Numerical approximation of gradients

- Two sided diference approximates the derivative of a function with $O(\epsilon^2)$ error therefore much better than the one sided difference that is $O(\epsilon)$ - and for ϵ smaller than 0 that that means that the two sidded difference has a much smaller error.

Gradient checking (Grad check)

- Used to check correctness of the implementation (bugs). Only to be used during debug, not during training (its slow).
- With Θ being a vector of parameters θ_i, and $d\theta[i] = \frac{\partial J}{\partial \theta_i}$, compare the "approx" derivate with the real $d\theta$ with the check:

$$\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2}$$

- Note that $||\cdot||_2$ denotes the squared root of the sum of the squared differences (that is, the norm of the vector).
- If the result is near 10^{-7} then great, if it is 10^{-5} then suspect something in your formula, if 10^{-3} the something is really wrong.
- Look for the what $d\theta[i]$ (what component) has the highest difference, to pinpoint the cause of the bug.

- Include the regularization term in the cost function when performing **grad check**
- Doesn't work with dropout (turn it off during grad check).
- Run at initialization and then again after some training.

Mini-batch gradient descent

- Applicable for large data-sets (single batch).
- Consists on running each iteration of gradient descent on smaller batches of the full data-set. May take too long per iteration.
- The cost function trends downward but not linearly in this case.
- If batch size = m then it is just batch gradient descent (run for all the examples at once). (use this for m <= 2000).
- If batch size = 1 then it is stochastic gradient descent (every example is a mini-batch). Gradient descent doesn't completely converge. Looses speedup from vectorization.
- Ideal scenario is in between the two above (may not exactly converge, but we can reduce the learning rate).

- Typical minibatch sizes are powers of two (64, 128, 256, 512) - make sure they fit in cpu/gpu memory.

Optimization algorithms - exponentially weighted moving averages

- $V_t = \beta V_{t-1} + (1-\beta)\theta_t$
- All the coefficients add up to 1.
- When $\beta = 0.9$ it takes a delay of approx 10 (think 10 days for a daily time-series data) for the contribution of a point to reduce to 1/3. The general rule (in which ϵ is 0.1 and $\beta = 1 - \epsilon$ to meet this example) is:

$$(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}$$

- To correct bias of the first few terms (compared to 0 initialization), the following formual can be used:

$$\frac{V_t}{1-\beta^t}$$

Gradient descent with momentum

- Uses exponential weighted moving averages to smooth out the derivatives dW and db, when updating W and b in each iteration. From example for dW (and similarly for db):

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

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

- Sometimes a simplified version is used that factors the $1-\beta$ term into the learning rate (that must be adjusted) instead of it being explicit, therefore:

$$V_{dW} = \beta V_{dW} + dW$$

$$\alpha adjusted = \alpha * (1-\beta)$$

- β is most comonly 0.9 (pretty robust value)
- Bias correction is not usually used for gradient descent.

RMSprop

- The method consists on updating W and b on each iteration with dW or db divided by the root mean square of the exponential moving average of dW or db (the square is element-wise):

$$S_{dW} = \beta S_{dW} + (1 - \beta) dW^2$$

$$W = W - \alpha \frac{dW}{\sqrt{S_{dW}}}$$

- Implementations add a small ϵ to the denominator to avoid divisions by 0.
- The objective is to have smaller/slower updates of db (derivative of the bias term or vertical direction) and higher/faster updates of dW (derivativee of the weights or horizontal direction), to improve convergence speed. dW is a large matrix, therefore RMS of dW is much larger than RMS of db
- Allows using a higher alpha
- Originally proposed by Geoffrey Hinton in a Coursera course!

Adam (adaptive moment estimation) optimization

- Momentum + RMS prop together, both with bias correction!

$$W = W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \epsilon}$$

$$b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$$

- There are two β parameters:
  - β_1 is the momentum parameter and is usually 0.9
  - β_2 is the RMSprop parameter and is usually 0.999
  - ϵ is usually 10^{-8}

Learning rate decay (lower on the list of hyper-parameters to try)

- Have a slower learning rate as gradient descent approaches convergence.

$$\alpha = \frac{1}{1 + decay\_rate * epoch\_num} * \alpha_0$$

- Alternatives:
  - Exponential decay: $\alpha = 0.95^{epoch\_num}$
  - Or: $\alpha = \frac{k}{\sqrt{epoch\_num}}$
  - Discrete stair case, manual decay, etc.

Local optima and saddle points

- Most points of zero gradient are saddle points, not local optima!
  - Plateau's in saddle points slow down learning.
  - Local optima are pretty rare in comparison/unlikely to get stuck in them.

Hyperparameter tuning process

- Learning rate is the most important to tune. Followed by momentum term (0.9), mini batch size and number of hidden units. Finally the number of layers and learning rate decay. Lastly beta1, beta2 and epsilon.
- Choose the hyperparameter value combinations at random, (don't use a grid) because of the high number of hyperparameters nowadays (cube/hyperdimentional space), doesn't compensate test all values/combinations.
- Coarse to fine tuning - first coarse changes of the hyperparameters, then fine tune them.
- Using an appropriate scale for the hyperparameters.
  - One possibility is to samples values at random within an intended range.
  - Using log scales to sample parameter values to try (for example, applicable for the learning rate).
- Two possible approaches for hyperparameter search:
  - Panda approach - nurse only one model and change the parameters gradually and check improvements - requires less hardware, might not be the most efficient method.
  - Caviar approach - run multiple models with different parameters in parallel - if you have the computing power for it.

Batch normalization

- Normalize not just the inputs but the activation inputs to the next layer, subtracting the mean and dividing by the variance (mean 0, variance 1).
- Normaly Z is normalized, before the activation function, thought some literature suggests normalizing after the activation function.
- New parameters gamma (multiplied by Z) and beta (added to Z after the multiplication) are introduced, and learned in the forward backward propagation. This is to prevent all neurons from having activations with mean 0 and variance 1 which is not desirable:

$$Z^{(i)}_{norm} = \frac{z(i) - \mu}{\sqrt{\sigma^2 + \epsilon}}, \text{ where } \mu = \frac{1}{m}\sum_i Z(i), \text{ and } \sigma^2 = \frac{1}{m}\sum_i (Z(i) - \mu)^2$$

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

- The bias parameter "b" in the calculation of Z is no longer needed, because the mean of Z is being subtracted, cancelling out any effect from adding "b". The new parameter beta effectively becomes the new bias term.
- About **Covariate shift** - Data distribution changes with inputs (e.g. over time, with new batches, etc), you need to retrain your network normaly...
- Batch normalization makes the process of learning easier by reducing the variability of the inputs presented to each layer (which now have similar variance and mean), therefore reducing the **covariate shift**, and that is especially important for deeper layers, where inputs could change significantly as a net effect of all the other changes in the network.
- It also has a slight regularization effect with mini-batch, due the "noise" introduced by the calculations of the mean and variance only for that mini-batch only, which has an effect similar to that of dropout.
- At test time there is no μ and σ^2, so these are computed based on an exponetially weighted average of these two parameters obtained during training (with mini-batches, though I think it works on a single batch too).

Multi-class classification

- The output layer must have as many neurons/units as there are classes (maybe one of these can be a catch all for points that aren't classified). Sum of all outputs must be 1, since these are probabilities of X being classified in any of these classes (likelyhood).

- **Softmax** activation function is used as the output activation function:

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

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{C} t_j}, \text{ where C is the number of classes and output units}$$

- **Softmax** is in contrast with **Hardmax**, where the networks output will be a binary vector with all 0s except for the position corresponding to the max value of $Z^{[L]}$.
- **Softmax** generalization of logistic regression to more than two classes. For two classes can be simplified/reduced to logistic regression.
- Loss function

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j log(\hat{y}_j), \text{ assuming a binary vector } y$$

$$\text{Cost function: } J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^i, y^i), \text{ backprop: } \frac{\partial J}{\partial z} = \hat{y} - y$$

Deep learning frameworks

Choose deep learning framework based on ease of programming, running speed, trully open (open source with good governance). A few:

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lsasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

TensorFlow

Example, minimize the simple cost function:

$$J(w) = w^2 - 10w + 25$$

Code:

```
import numpy as np
import tensorflow as tf
```

```
coefficients = np.array([[1.,],[-10.],[25.]]) # run 3
# coefficients = np.array([[1.,],[-20.],[100.]]) # run 3.1

w = tf.Variable(0,dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1]) # run 3 - placeholders are added later
# cost = tf.add(tf.add(w**2, tf.multiply(-10.,w)), 25) # run 1 - ugly version
# cost = w**2 - 10*w + 25 # run 2 - pretty version
cost = x[0][0] * w ** 2 + x[1][0] * w + x[2][0] # run 3 - with placeholder variables
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w)) # output should be 0

# session.run(train) # run 1 and 2
session.run(train, feed_dict={x:coefficients}) # run 3 - note x is being specified
print(session.run(w)) # output should be 0.1 for runs 1 and 2, 0.2 for run 4

for i in range(1000):
    # session.run(train) # run 1 and 2
    session.run(train, feed_dict={x:coefficients}) # run 3 - note x is being specified
# outpput should be 4.9999, close to the solution 5
print(session.run(w))
```

Notes on this TensorFlow examples:

- The cost function specifies the computation graph.
- Tensor flow knows how to calculate the derivatives, no need to specify backprop updates!
- Placeholder specified values that will be supplied later to the computation graph.
- Easy to swap gradient descent with an Adam optimizer with a single line code change.

Structuring machine learning projects

Orthogonalization

- Separating the hyperparameter tunning into different "dimensions". "Using different knobs" for each objective/tunning/assumption in ML:
  - Fit training set well on cost function: Bigger network, use Adam
  - Fit dev set well on cost function: Regularization, Bigger training set
  - Fit test set well on cost function: Bigger dev set
  - Perform well on real world: Change dev set, Change cost cost function
- A bad example of a "knob" is early stopping, because it simultaneously affects the fit of the training set and the dev set, therefore is not a targeted "knob".

Single number evaluation metric

There is usually a trade-off between:

- Precision - the percentage of images identified as cats that actually are cats.

$$\frac{true\_positives}{true\_positives + false\_positives}$$

- Recall - the percentage of cats correctly classified out of all cat images

$$\frac{true\_positives}{true\_positives + false\_negatives}$$

Combine the 2 metrics into a single number with the F1 Score:

- Average of precision P and recall R

$F1Score = \frac{2}{\frac{1}{P} + \frac{1}{R}}$, this is the harmonic mean of P and R

Satisficing and Optimization metric

- Satisfacing metric - one that only needs to be better than a fixed threshold (e.g. running time)
- Optimizing metric - one that we want to optimize (e.g. accuracy)

Training/dev/test sets

- Dev and Test set must (or at least should) come from the same distribution! Dev and test set should reflect the data that you expect to get in future, and consider important to do well on.
  - The traditional splits (70/30 or 60/20/20) are no longer useful if your have a large datasets (e.g. 1 million records). In that case use a 98/1/1 split.
  - Not having a test set might be OK (sometimes there is even only a training set, though that is not recommended).
- Metrics that evaluate classifiers should reflect what is expected from a classifier, for example, measuring an algorithm just by classification error does not take into account the fact that some of the false positives that it classifies are actually porn images instead of cats, in which cased a revised (e.g. weighted version) the metric should be used instead.
  - Use orthogonalization, first place the target (the right metric), only then try to improve the performance according to that metric.
- If doing well on your metric + dev/test set does not correspond to doing well on your application, change your metric and/or dev/test set.

- Do NOT run for too long with an appropriate dev set or applicable metric.

Comparing to human-level performance

- Performance of an algorithm can surpass **human-level performance**, but even then they will be bound by the **Bayes optimal error** (or **Bayes error**), which is the best possible error value that can be achieved.
- There are two reasons why performance slows down when human-level performance is surpassed, because humans are already close to "Bayes optimal error" in some cases.
- So long as ML is worse than humans, you can:
  - Get labeled data from humans
  - Gain insight from manual error analysis:
  - Better analysis of bias/variance.

Avoidable bias

- Focus on improving bias - if human performance is much better than performance on the training set.
- Focus on improving variance - if human performance is close to performance on the training set, but performance on the dev set is significantly lower as training performance.
- The difference between training error and human error is the **avoidable bias**

Understanding human-level performance

- **Human-level** error as a proxy for **Bayes error** - we use the best possible human-level error when trying to approximate Bayes error.
  - This matters when the avoidable bias and the variance are already low.

Exceeding human-level peformance

- If an algorithm surpasses human-level performance it is not clear what the avoidable bias is, and therefore we don't know if we should focus on improving bias or variance. Examples are:
  - Online advertising, Product recommendations, Logistics, Loan approval - all structured data problems, NOT natural perception problems.
  - Speech recognition, some image recognition, medical ECG and cancer analysis, etc.

Improving your model performance

- Improve **avoidable bias**
  - Train bigger model
  - Train longer/better optimization algorithms (momentum, RMSProp, Adam)
  - NN architecture/hyperparameters search (RNN, CNN, etc.)
- Improve variance
  - More data
  - Regularization (L2, dropout, data augmentation)
  - NN architecture/hyperparameters search (RNN, CNN, etc.)

Error analysis

- Finding the most promising way to reduce error, by manually going through examples and find out what the misclassified examples refer to (e.g. dogs, when trying to classify cats).
  - Depending on the percentage of these case, chose the most prominent cases to focus on. If there are multiple examples, each team can focus on a different case.
- Incorrectly labeled examples
  - Deep Learning is robust to random errors in the training set - not worth fixing manually.
  - The frequency of mislabeled examples due to incorrect labeling should be calculated similarly other mislabeled example analysis mentioned above.
- Correcting incorrect dev/test set examples
  - Whatever is done to the dev set we should also do to the test set so that they continue to come from the same distribution. The training set might come from a slightly different distribution (discussed later).
  - Consider examining the (mislabeled) examples that the algorithm got right, not just those that it misclassified.

Build your first system quickly, then iterate

- Includes setting up dev/test set and metric.
- Allows you to identify areas where the algorithm is not performing well

- Use Bias/Variance analysis and error analysis to prioritize next steps.

Training and testing on different distributions

Suppose we have 200k examples from a distribution A (high resolution cat images from webpages) and only 10k images from distribution B (lower resolution cat images from a mobile app). It also happens that the images that we will need to classify are from distribution B.

- OPTION 1 (BAD): Mix A and B, randomized the images, use a training/dev/test split of 205k/ 2.5k/2.5k images. This is bad because distribution B is the target, and there will be only a low number of the dev/test set in this case.
- OPTION 2 (GOOD): use 200k from A as training examples and split the 10k from B into a dev and test sets. This is much better becuse dev and test are used to set the "target" distribution (B in this case).

Variance (difference between error in the training and dev sets) might be higher (assuming OPTION 2) due to the different distributions of the two sets. A **training-dev** set using the same distribution as that of the training data (A), can be created to find out if:

- The error is due to overfitting the training set (variance), if the training-dev error is close to the dev error,
- or if it is due to the model not generalizing well to distribution B (**data mismatch** problem) if training-dev error is close to the training error.

Note the 4 type of errors:

- avoidable bias: $human\_level - training\_set\_error$
- variance: $training\_set\_error - training\_dev\_set\_error$
- data mismatch: $training\_dev\_set\_error - dev\_error$
- degree of overfitting to dev: $dev\_error - test\_error$

Addressing data mismatch

- Carry out manual error analysis and try to understand the differences between the training and dev/test sets.
- Make training data more similar (**artificial data synthesis**), or collect more data similar to dev/test sets.
  - Make sure that **artificial data synthesis** is varied enough to avoid overfitting to one dimension or a sub-set domain of the synthesized data.

Transfer learning

Consists on re-using a previously trained model (e.g. in general image recognition if your task is to make radiology diagnosis). This usually requires replacing the last layer of the classifier with a new layer (or layers) with weights and bias randomly initialized. Then there are two options:

- Retrain only the last layer if we don't have enough data in the new (e.g. radiology) data-set.
- Or, if there is sufficient data, retrain entire model, in which case:
  - The initial phase of training (e.g. on image recognition) is called **pre-training**.
  - Training on the new data set (e.g. radiology) is called **fine-tuning**.
- Transfer learning (from models A to B) makes sense if:
  - Task A and B have the same input X
  - You have a lot more data for Task A (e.g. image recognition) than Task B (e.g. radiology).

– Low level features from A could be helpful for learning B.

Multi-task learning

Learning to identify multiple objects at the same time (e.g. automatic car driving). Loss function must be a vector $\hat{y}^{(i)}$ where each entry corresponds to one object (for example). One cost function for this can be:

$$J = \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{c} \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

Where c is the number of object classes/types and $\mathcal{L}$ is the logistic loss function.

- Unlike Softmax, one example (e.g. image) can have multiple labels.
- $\hat{y}$ can "unknown" values in certain positions. To calculate $\mathcal{L}$ in that case, we can simply exclude those terms from the inner sum of the cost function.

Multi-task learning makes sense when:

- 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. If we focus on a single target task, they we need to make sure that the other tasks have a sufficiently large number of examples.
- Can train a big enough enough neural network to do well on all tasks with better performance (e.g. computer vision or object detection).

Transfer learning is much more used currently (2017-12-15) in practice.

End-to-end learning

A single classifier can replace multiple stages of processing in previous approaches (eg. speech processing).

- For small data sets the traditional pipeline approach works better, and for large amounts of data neural networks work better.

Pros of end-to-end learning:

- Let the data speak (features are learned, not forced).
- Less hand-designing of components needed.

Cons of end-to-end learning:

- May need large amount of data.
- Excludes potentially useful hand-designed components.

Key question: do you have sufficient data to learn a function of the complexity needed to map x to y?

If not: * It is also possible to use Deep Learning (DL) to learn individual components. * Carefully chose $X \to Y$ to learn depending on what tasks you can get data for.

Sidenote: **Motion Planning** - area of robotics, decide the next move.

Convolutional Neural Networks

Computer vision

Problems tackled: * Image Classification * Object detection * Neural Style Transfer * ...

Convolution: Application of a **kernel** or **filter** to a matrix. Denoted by operator $*$.

- Typical example is edge detection in images (e.g. the Sobel or Scharr algorithms/filters)
- Size of the result of convolving an image of size $n \times n$ with a filter of size $f \times f$ is $(n-f+1) \times (n-f+1)$.
- **Strided convolution** - use a different step than 1 (e.g. 2) when convolving. Size of the output (for each dimension) becomes $\frac{n+2p-f}{s} + 1$ where s is the stride.
- **Padding** - adding an extra border of pixels set to 0, to avoid loosing information. If $p = 1$ corresponding to a 1 pixel padding border, then the resulting convolved image is $(n+2p-f+1) \times (n+2p-f+1)$.

- If we need the resulting image to be of a specific (or the original) size, then we need to solve for p the equation $\frac{n+2p-f}{s} + 1 = t$ where t is the target size (of each dimension).
  - **Valid** padding means that no padding is used.
  - **Same** padding means a padding such that the size of the output image is the same as the input image, so $p = \frac{f-1}{2}$ (from solving $\frac{n+2p-f}{s} + 1 = n$ and assuming a stride of 1).
- By convention f is always odd (so it has a central position and we can use the math above, but for not other specific reason).
- In math text books convolution as a mirroring step where the filter is flipped in the x and y axis. So technically the convolution used in Deep Learning is called "cross-correlation", but in Deep Learning we still just call it convolution by convention.
  - The flip is used in signal processing to ensure the associativity property of convolution $(A * B) * C = A * (B * C)$, not required in deep learning.
- On RGB images the filter is a cube $f \times f \times f$. Applying convolution is similar, but at each step we do f^3 multiplications and then we sum all the numbers to get 1 output value. The sizing formulas above apply because the number of channels is not taken into account to determine the size of the output.

- If we have multiple filters, we can apply them to the same image and then stack the multiple outputs as if they were different channels n_C, into a *volume*. The term *depth* is often used to denote the *number of channels*, though that can be more confusing.

Functions for convolution:

- Python: `conv-forward`
- TensorFlow: `tf.nn-conv2d`
- Keras: `Conv2D`

One layer of a convolutional network

- Performs the convolution of the input image across all chanels n_C, uses multiple filters, and therefore originates multiple convolution outputs.
- A bias term b is then added to the output of the convolution of each filter W and we get the equivalent of the term Z in a normal neural network.

- We then apply an activation function such as ReLU Z (on all chanels), and we finally stack the channels to create a "cube" that becomes the network layer's output.

Notation and volumes:

- $f^{[l]}$ = filter size of layer l
- $p^{[l]}$ = padding of layer l
- $s^{[l]}$ = stride of layer l
- $n_C^{[l]}$ = number of filters/channels of layer l
- Input size: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$
- Output size: $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
- Output volume *h*eight: $n_H^{[l]} = \lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$

- Output volume *w*idth: $n_W^{[l]} = \lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor$
- Each filter volume: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$
- Activations volume: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
- Activations volume for M examples: $A^{[l]} \rightarrow M \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]} = M \times a^{[l]}$
- Weights volume: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$
- Bias size/number: $n_C^{[l]}$

Convolutional network

- Stack of layers as above (CONV layers)
- The last layer is a logistic or sigmoide, as required.
- There are also pooling (POOL) layers and fully connected (FC) layers.
- Sometimes POOL are not counted as layers since they don't have parameters to optimize.
  - Here we will assume that 1 layers is a combination of CONV + POOL

Pooling layers

- Max pooling - define regions of size f (filter size) and step through them according to a stride s, and create an output matrix containing only the max value of each reagion. This normally reduces the size of the representation.

- The intuition for using max pooling (except from wide acceptance) is that it expresses that if a feature exits in a region, we will express it more evidently.
- Size of the output will be calculated by the same ouput volume formulas above.
- Computations are made independently for each channel.
- Average pooling is also possible but less used, except deep in a neural network to colapse a representation, reducing an image size.

- There are no parameters to optimize. Only the hyperparameters f, s and the type of pooling (max or average).
- Usually we don't use padding (though there are exceptions).

Fully connected layers

- These are layers that take the output of CONV or CONV + POOL layers, and take each individual pixel as an input (instead of a volume with channels/filters) by colapsing the volume into a single vector with equivalent size.

- A fully connected layers is a norma neural network layers (e.g. with ReLU activation) that takes this colapsed output of the previous layer.
- FC layers have significantly more parameters than CONV.

Why convolutions:

- Even for a small image, normal NN would use very high number of parameters per image.
- **Paramater sharing** - A feature detector (such as a vertical edge detector) that's 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 (the size of the filters).

Historical/Classic examples

- LeNet - 5
  - INPUT -> CONV -> POOL -> CONV -> POOL -> FC -> FC -> OUPUT
  - Used Sigmoid/Tanh instead of ReLU.
  - Average pooling (not max pooling)
  - Includes not linearities after the pooling.
  - different filters for each channel (not fully understood unless reading the paper)
  - "Gradient-based learning applied to document recognition"
- AlexNet
  - INPUT (227x227x3) -> CONV -> POOL -> CONV -> POOL -> CONV -> CONV -> CONV -> POOL -> FC -> FC -> SoftMax -> OUTPUT
  - Similar to LeNet but much bigger
  - Uses max pooling
  - most convolutions are "same".
  - Use ReLU activation
  - Multiple GPUs
  - Local Response Normalization -> normalizes each position in a volume across all channels of the volume (not very used in practice).
  - "ImageNet classification with deep convolutional neural networks" -> good/easy paper to read.
  - ~60 million parameters
- VGG - 16
  - INPUT (224x224x3) -> CONV (64) -> POOL -> CONV (128) -> POOL -> CONV (256) -> POOL -> CONV (512) -> POOL -> CONV (512) -> POOL -> FC (4096) -> FC (4096) - SoftMax (1000)
  - All CONV = 3x3 filter, s = 1, same
  - All MAX-POOL = 2x2, s = 2
  - ~128 million parameters
  - "Very deep convolutional networks for large-scale image recognition"

ResNet

- "Deep residual networks for image recognition"
- Add "shortcut"/"skip connection" to the network, where activation of past layers are included when calculating the activation of layers in future:

$$a^{[l+2]} = g(z^{l+2} + a^{[l]})$$

- These are residual blocks. A ResNet will have a sequence of these blocks.
- This allows training much deeper networks because the training error always goes down with an increase in the number of layers (which is not the case with traditional CONV nets)
  - "Learning the indentity function is easy" - This means that it doesn't hurt to add to layers even if they do not contribute to the activation of previous layers. if $W^{[l+2]} = 0$ and $b^{[l+2]} = 0$ then $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$
  - This means that a larger network will at least do as good as a smaller network, never worse.
  - During training of a traditional deep convolutional network we might see the magnitude (or norm) of the gradient for the earlier layers decrease to zero very rapidly as training proceeds. In ResNets the "shortcut"/"skip connection" allows the gradient to be directly backpropagated to earlier layers, thus solving the problem **vanishing gradients**.

- For the opperation $z^{l+2} + a^{[l]}$ to be possible, "same" convolutions must be used. Alternatively there could be a W_s matrix such that $z^{l+2} + W_s a^{[l]}$ makes the dimensions match. This matrix could also add padding, or it could be just additional parameters to learn.
- Another similar (mentioned to be more "powerful") design allows for skiping 3 layers instead of 2.

Networks in Networks / 1x1 Convolution

- Useful only when the input has multiple channels.
  - Like having a fully connected layer for each pixel in the input.
- Useful to shrink the number of channels in a volume. Whereas pooling only reduces height and width.

Inception network (aka GoogleNet)

- Instead of choosing which filters to use, we used them all and the network chooses what's important!
- in a single layer applies several types of filters, including
  - 1x1 convolutions
  - several "same" convolutions with different filter sizes (and possible stride sizes)
  - MAX-POOL directly from the input, in which case **padding must be added before the MAX-POOL** to ensure that the height and width of the output is the same as the input - this is an exceptional case, since padding is not usually added where doing POOL.
- The output will have a lot of channels!
- To reduce the computation cost of convolutions on many channels, 1x1 convolutions with a lower number of channels are used as an intermediate step or "bottleneck" for convolutions with larger number of channels (reducing the number of operations by a factor of 10, compared with doing the larger convolution on the input directly).
  - This doesn't seem to work performance compared to doing convolving the larger filters directly on the input volume.
- Similarly, after the MAX-POOL step, a 1x1 CONV is used to reduce the number of channels in its output.

- SO 1x1 CONV are used as intermediate (or final in the case of MAX POOL) steps in each layer either as bottlenecks are to reduce the number of channels at the output of MAX-POOL.
- One layer containing these 1x1 CONV, the larger CONV and the MAX-POOL (in parallel) is called an "Inception block" of which there are many in a network.
- There is a final FC and SoftMax layer.
- Side-branches (feature of inception networks) - Take some hidden layer and use FC and then SoftMax to predict a label. This ensure that even in hidden layers are not too bad to predict the output cost. These have a regularizing effect in the network.

- Name is a referent to the "Inception" and the paper includes the citation "we need to go deeper" (in a URL?).

Using open source implementations and transfer learning

- Get examples on GIT
- Use transfer learning/pre-trained models.
- Example: *Imagenet* (1000 output classes).
- Use my own Softmax output layer:
  - Freeze the parameters in the other layers (e.g. trainableParameter=0 or freeze=1, depending on framework).
  - (with small data) Pre-compute the output of the network before the new SoftMax layer across training examples and save them to disk (so you don't have to compute these on every epoch of the shallow training). Then train only the shallow SoftMax model.

  - (with more data) Include a few of the last few layers in the training, not just SoftMax. We may want to retain or not the weights of some of the unfrozen layers (used them as initialization) and add just some additional extra training there. Alternatively the last unfrozen layers could be randomly initialize, re-designed, etc.. The more data you have, the less layers you freeze.
- With a lot of data (and CPU power) just reuse the architecture and retrain the whole network.

Data Augmentation

- Having more data helps in the majority of computer vision tasks (not the same for other areas of ML).
- Methods (from existing images):
  - Mirroring
  - Random cropping (works as long as the crops are reasonable)
  - Rotation
  - Shearing
  - Local warping
  - Color shifting - change the RGB colour balance according to some probability distribution. (it is possible to use PCA to choose RGB - check AlexNet paper for "PCA color augmentation")
- A common way of implementing augmentation is to have one (or multiple) CPU thread to do the agumentation/distortion, and then run the network training in parallel in another thread or GPU.

State of computer vision

little data ← object detection - image recognition - speech recognition → **lots of data**

Two sources of knowledge:

- Labeled data (x,y)
- Hand engineered features/network architecture/other components

Often for computer vision there is not as much data as needed. So hand engineered features are more needed.

- Even though the amount of data increased dramatically in the last few years there is still a lot of hand engineering, specialized components and hyperparameters for historical reasons.

Tips for doing well on benchmarks/winning competitions

(Andrew wouldn't use these in production systems)

- Ensembling - train several networks independently and average their output. (typical 3-15 networks, lots of CPU time) - usually not used in production.
- Multi-crop at test time (data augmentation) - run classifier on multiple versions of test images and average the results - more used in benchmarks rather than production systems.
- Use open source code, implementations from published literature, and if possible the trained models themselves.

Classification with localization

- Identify the object and localize it in the picture.

- Add 4 more outputs (in addition to the object classes) that are the bounding box coordinates b_x, b_y, b_h, b_w, that is the bounding box center coordinates and its height and width respectively.
- Add another output P_c that indicates if there is an object in the image.

- if P_c is 0 (no object identified) then the loss function must take that into consideration, basically just calculating $(\hat{Y}_1 - Y_1)^2$ and ignoring the remaining components (this examples uses squared error, but it could be a log likelyhood, etc.)

Landmark detection

- Basically use a conv network to output the coordinates of the points of interest (e.g. find the corners of the eyes in a person's picture). It can be a high number of points of interest though.

- Landmark labels must be consistent between training/test examples.

Object detection

- First build a conv net that identifies a certain class(es) of objects, and train it only on pictures where the positive examples include a single object (e.g. a single car).
- To identify multiple cars, use a sliding window approach (with windows of varying sizes, over multiple passes), and run each crop via the conv net.
  - This is extremely inefficient because we don't know the stride or the size of the crops, and we need to evaluate a high number of crops.
- Fortunately there is a solution, read on!

Convolutional implementation of sliding windows

First a useful tool: Turning FC layers into convolutional layers:

- Basically just use a filter with the same heigh and width as the input volume (no padding and or stride used), e.g. 5x5 in this example. The number of channels/filters can then correspond the number of units that we would have in a FC layer (e.g. 400 in this example). The output will be a 1x1x400 volume, which is equivalent to having a fully connected layer with 400 units! Note that if the input volume has 16 channels, then the filter volume will be 5x5x16x400.

And now the convolutional implementation for sliding windows:

- The objective is to implement a convolutional network that in a single pass calculates the equivalent of the convolution of a number of sliding windows of a certain size and with a certain stride. This can be done with a filter and a MAX POOL layer.

- MAX-POOL size/stride determines the stride of the sliding window - but this stride is still fixed!
- Filter sizes determine the size of the window - but these filter sizes are still fixed!
- Each data-point
- So we would still need multiple convolutions with multiple filter sizes and max pool strides/sizes.

YOLO algorithm (You Only Look Once):

- Efficient algorithm, convolutional implementation, runs very fast.
- First divide the input image into a grid (say 19x19). We want to use a conv net for each volume in a grid, or better do one single conv net pass that performs the equivalent of that. Say that for each grid division we want to have an output vector $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$ (assuming 3 classes), they we want to obtain an output volume of dimensions 19x19x8 (8 is the dimension of the vector for each grid cell)
  - This can be achieved in a single convolution by setting appropriate filter and max pool sizes/strides.
- YOLO considers that the grid cells that have an object are those that contain the center coordinates of the bounding box for the object, b_x and b_y.

- How do you encode bounding boxes (that may go beyond the grid cell)?
  - b_x and b_y are between 0 and 1 and are specified relative to the bounding box start point in the upper left corner (0,0).
  - b_h and b_w are specified as a proportion in relation to the grid cell size, and therefore can be greater than 1.

  - Other encodings are possible.
- Problem: what if there are multiple objects in the same bounding box?
  - Addressed later
- Problem: how to decide what cell has the center of the bounding box?
- Better check the YOLO paper (hard to read): "You Only Look Once: Unified real-time object detection"

Intersection over Union (IoU) - to evaluate object localization

- Given 2 overlapping bounding boxes, we define:

$$IoU = \frac{size\ of\ intersection}{size\ of\ union}$$

- We say that a bounding box is "correct" if $IoU \geq 0.5$. This is just a convention. 0.5 may be too low if we are more demanding (but never lower than 0.5).

Non-max suppression

- Multiple bounding box detections are possible because it is possible that several grid cells (in YOLO) have a high probability of having the coordinates of the center of the object.
- Turns out that P_c can be the *Probability of the class* (instead of 1 or 0)
- First of all we discard any bounding boxes with $P_c \leq 0.6$
- Then we choose the highest probability bounding box first, and then iteratively eliminate the overlapping bounding boxes with a $IoU \geq 0.5$, then select the next highest P_c, eliminate overlapping boxes with $IoU \geq 0.5$, and so on, while there are still boxes remaining.

Anchor boxes

- With anchor boxes with increase the number of output parameters to encode multiple possible object detection. E.g. with two anchor boxes (and for 3 classes as before) the output vector is $y = [P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$, notice that all parameters are doubled, one of each anchor box.

- Previously: Each object in training image is assigned to grid cell that contains that object's midpoint.

- With anchor boxes: Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

- What if you have 2 anchor boxes but 3 objects? The algorithm doesn't work well in this case.
- If there are 2 objects but both have the same anchor box shape the algorithm won't work well either.
- It is not clear if the objects in different anchor boxes need to have different classes, I assume not.
- One way to choose anchor box shapes is by doing K-Means clustering over the anchor boxes of the training set.

YOLO algorithm (second take)

- If we have a grid of 3x3, 3 classes and 2 anchor boxes, our output volume will be $3 \times 3 \times (5 + 3) \times 2$, where 5 comes from the components P_c, b_x, b_y, b_h, b_w.

- With 2 bounding boxes for each grid cell, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class, use non-max suppression to generate final predictions.

Region proposals

- Optional because Andrew uses it less often. He thinks like approaches that require only 1 step like YOLO, instead of 2 like R-CNN is more elegant.
- **R-CNN** (R as in regions) Select only a few regions/windows that seem relevant in the image
  - To find the regions, a **segmentation algorithm** is used, to find "blobs" that are candidates to be enclosed by bounding boxes.
  - Output label + bounding box (possibly a better bounding box)
  - Historically slow.
  - Fast R-CNN: Propose regions. Use convolution implementation of sliding windows to classify all the proposed windows.
  - Faster R-CNN: Use convolutional network to propose regions. - still slower than YOLO.

Face recognition

- Related to **liveness detection** - (live vs non-live human)
- Validation (1 image to 1 ID) much easier problem than:
- Recognition 1:K persons - requires a database with the recognized individuals.
- **One-shot learning**: Learning from 1 example to recognize the person again.
- Learning a **similarity function**:
  - $d(img1, img2)$ = degree of difference between images
  - if $d(img1, img2) \leq \tau$ same, otherwise different.

Siamese network

- Paper "DeepFace closing the gap to human level performance"
- Idea: run each photo trough a convolutional neural network and use the output as an "encoding" of the picture $x^{(1)}, ..., x^{(m)}$ as $f(x^{(1)}), ..., f(x^{(m)})$.
- Then calculate the distance between two encodings that is the norm between the difference of the two encodings:

$$d(x^{(1)}, x^{(2)}) = ||f(x^{(1)}) - f(x^{(1)})||_2^2$$

The Triplet Loss function

- Learning objective - Given a image A (*Anchor*), we want to find a function d such that d is small for another image of the same person P (*Positive*) and at the same time higher by an α **margin** when compared with an image of a different person N (*Negative*). In other words:

$$d(A, P) - d(A, N) + \alpha \leq 0$$

- Loss function - Given 3 images A, P, N (*triplet*) we define the loss as:

$$\mathcal{L}(A, P, N) = \max(d(A, P) - d(A, N) + \alpha, 0)$$

- We do need a data-set where we have multiple pictures of the same person (suggested 10 pictures per person on average).
  - Choosing the triplets randomly, then it makes it too easy for the training.
  - We should instead choose pictures where the negative is as close as possible as the positive, as these will be the best triplets to train on.
  - Paper: "A unified embedding for face recognition and clustering"

Face Verification and Binary classification

- Example of **siamese networks**, with a logistic regression unit for the output. That unit implements:

$$\hat{y} = \sigma(\sum_{k=1}^{N_f} w_i |f(x^{(i)})_k - f(x^{(j)})_k| + b)$$

- In this formula, $x^(j)$ and $x^(i)$ are training examples, w_i and b are the parameters of the logistic unit, N_f is the number of input features in each vector input to the logistic unit, k is one single feature (pair of nodes from each of the siamese networks), and $f()$ is the function calculated by each of of the siamese networks.
- In some variations the χ^2 formula is used instead of the absolute value of the difference:

$$\chi^2 = \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$

- We don't need to store the original images in a production verification system. Only the encodings of the images (e.g. for each individual).
- This type of pre-computation works also with triplet encoding.
- The siamese network is trained using matching and mismatching pairs of pictures.

Neural style transfer

- Given a Content image C (a photo) and a style image S (eg. a painting) generate a new version of the original content reflecting the style in the painting, the "Generated image" G.

What are deep ConvNets learning - visualizing what a deep network is learning

- Book/paper: "Visualizing and understanding convolutional networks"
- Pick a unit in layer 1, and find the nine image patches that maximize the unit's activation.
  - Repeat for other units.
- Deeper units learn increasingly more complex features.

Cost function for neural style transfer

- Paper: "A neural algorithm of artistic style".

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

- So basically the two components measure the difference between the content image to the generated and the style image to the generated image, where α and β are hyperparameters.
- Algorithm:
  1. Initialize G randomly (e.g. G: 100x100x3 randomly initiated)
  2. Use gradient descent to minimize $J(G)$ and update G (where α is the learning rate):

$$G = G - \frac{\alpha}{2G} J(G)$$

Content Cost Function

- Choose a layer l somewhere in the middle of the layers of the network (neither too shallow or too deep).
- Use pre-trained ConvNet. (E.g., VGG networks) to measure how similar the content and generated images are.
- let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images.

- if $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content.
- Then calculate the L2 norm (element-wise sum of squared differences) of the activation vectors (activations are unrolled into a vector), with:

$$J_{content} = \frac{1}{2}||a^{[l](C)} - a^{[l](G)}||^2$$

- Finally, when we are generalizing in a $n_H \times n_W \times n_C$ volume we can use a more appropriate normalization constant:

$$J_{content}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{[l](C)} - a^{[l](G)})^2$$

Style cost function

- Using l's activation to measure style, we define style as correlation between activations across channels (really the unormalised cross variance).
- Let $a^{[}l]_{i,j,k}$ = activation at (i, j, k).
  - The style matrix ("Gram matrix" in algebra) is $G^{[}l]$ and is $n_c^{[l]} \times n_c^{[l]}$. k and k' are coordinates of the style matrix.
- We calculate style matrices for the style image and the generated image:

$$G_{kk'}^{[l](S)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l](S)} \times a_{ijk'}^{[l](S)}$$

$$G_{kk'}^{[l](G)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l](G)} \times a_{ijk'}^{[l](G)}$$

- Finally the style cost function can now be defined by the Frobinius norm (sum of squares of the element-wise differences) multiplied by a normalization constant (the fraction portion):

$$J_{style}^{[l]}(S, G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} ||G^{[l](S)} - G^{[l](S)}||_F^2$$

$$J_{style}^{[l]}(S, G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})^2} \sum_k \sum_{k'} (G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)})$$

- Finally style transfer is better if all the layers are used. So we get the final style cost function (with additional hyperparameter vector λ):

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J_{style}(S, G)$$

1D and 3D generalizations

- Everything that we've learned on ConvNets for 2D data is also applicable for 1D data (e.g. sound, ECG data) as well as 3D data (color images are already 3D data) such has CT scans with multiple body slices.

Sequence models

Why sequence models

- Speech recognition
- Music generation
- Sentiment classification
- DNA sequence analysis
- Machine translation
- Video activity recognition
- Name entity recognition

Notation

Motivating example:

```
x: Harry Potter and Hermione Granger invented a new spell.
y: 1     1      0   1        1       0        0 0    0
```

Elements in each position (9 positions in this cases) - t stands for temporal btw - are represented as:

$$x^{<1>}, x^{<2>}, ..., x^{<t>}, ..., x^{<9>}$$
$$y^{<1>}, y^{<2>}, ..., y^{<t>}, ..., y^{<9>}$$

- $x^{(i)<t>}$ is the t sequence element of example i
- $y^{(i)<t>}$ is the t sequence element of ouput i
- $T_x^{(i)}$ is the total number of sequence elements of training example i (9 in this case).
- $T_y^{(i)}$ is the total number of sequence elements of ouput example i (9 in this case).

Search companies use word dictionaries (**Vocabularies**) of 1 million or more words, while many commercial applications use dictionaries of sizes between 30k and 50k words, in some cases up to 100k words.

One possible word representation:

- One-Hot vectors, where each word is a binary vector of the size of the whole vocabulary.
- More to come...

Recurrent neural networks (RNN)

Why not use a standard network: * Inputs, outputs can be different lengths in different examples. * Doesn't share features learned across different positions of text.

Recurrent Neural Networks (RNN):

- The idea is to use the activation at $t-1$ as well as the input at time t:

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

That can be simplified by horizontally stacking W_{aa} and W_{ax} into W_a, and vertically stacking $a^{<t-1>}$ and $x^{<t>}$ as $[a^{<t-1>}, x^{<t>}]$:

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$

And we also have:

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

Which is usually simplified by renaming W_{ya} to W_y:

$$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

RNN backpropagation

Loss is again the cross entropy loss (standard logistic regression loss)

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} log(\hat{y}^{<t>}) - (1 - y^{<t>}) log(1 - \hat{y}^{<t>})$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

- **Backpropagation through time** - Cool name derived from the need to backpropagate for t before we propagate for $t-1$ starting from the loss above.

Different types of RNNs

Paper: "The unreasonable effectiveness of recurrent neural networks."

- Many-to-many - many inputs and many outputs, when $T_x = T_y$.
- Many-to-many - many inputs and many outputs, when $T_x != T_y$, e.g. machine translation - two parts, one encoder (e.g. from the source language) of size T_x, one decoder (e.g. to translate to the target language) of size T_y.
- Many-to-one - many inputs and only one ouput (e.g. for sentiment analysis)
- One-to-one - just for the sake of completeness - really just a standard NN.
- One-to-many - one input, multiple outputs (e.g. music generation) - note when generating sequences we usually take each output of $t-1$ and feed it as the input of t (sometimes the single input can be 0).
-

Language Model and Sequence generation

With an RNN:

- Training set: large corpus of English text (corpus is an NLP related term!)
  - From which a vocabulary is derived.
- Text is first tokenized (e.g. separated in words).
- It is generally useful to have an `<EOS>` (end of sentence) token.
- Words not in the vocabulary are replaced with the `<UNK>` token.

- In a simplistic model the size of the input vectors for each word is the size of the vocabulary.
- The output of each stage of the RNN is a vector of the size of the vocabulary, and for each word in that vocabulary it contains the conditional probability of that word, given the previous words already fed to the RNN.

Sampling novel sequences

- First feed a vector of zeros as $x^{<1>}$ to the RNN. That produces a vector $y^{<1>}$ with the probability of each word in the vocabulary, from which we can sample a few with `np.random.choice`. Then we pick a word from our sample (e.g. one which has a high probability of being the first word in a sentence such as "The"), and we feed it as the input $x^{<2>}$.
  - We obtain a vector $y^{<2>}$ corresponding to the conditional probability of each word in the vocabulary given the first chosen words. For example $P(y^{<t>}|"The")$ if the first chosen word was "The".
  - We repeat this until we have a sequence of the desired size or an `<EOS>` token is generated.
- Sometimes a `<UNK>` token can be generated, though it can be explicitly ignored.
- Note that the vocabulary can also have character level, which means that the sequence becomes the character sequence.
  - Ends up with much longer sequences, computational expensive, that don't capture relationships between words.
  - They have the advantage of not having to deal with `<UNK>` tokens.

Vanishing gradients with RNNs

- RNNs are not good at capturing long term dependencies, for example: "The cat, which already ate ..., was full." The subject "cat" can be distant from the predicate "was".
  - This is due to vanishing gradients - most common problem when training RNNs. Exploding gradient are catastrophic and usually cause NANs in the output, so at least they are easy to spot (gradient clipping can be used but that is not ideal).

Gate recurrent unit (GRU)

- Paper: "On the properties of neural machine translation: Encoder-decoder approaches"
- Paper: "Empirical evaluation of Gated Recurrent Neural Networks on Sequence Modeling"

Notation:

$$c = \text{memory cell}$$
$$c^{<t>} = a^{<t>}$$

c and a are the same in this case, but they will be different in LSMTs, hence the distinction.

Example: c remembers if "cat" was singular or plural.

At each time instant we compute a candidate replacement for $c^{<t>}$ referred to as $\tilde{c}^{<t>}$.

$$\tilde{c}^{<t>} = tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

The relevance gate term Γ_r ponders the relevance of $c^{<t-1>}$ for the new candidate replacement (researchers have converged to use this term for robustness and to extend long range dependency memorization):

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

Then we also have an update "Gate" (hence the use of the letter gamma Γ), that decides if the new candidate should replace the value of the memory gate or not.

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

This function is always very close to 0 or very close to 1, so it can be considered close to binary output. Then the decision on weather to retain C or update it with $\tilde{c}$ is made with:

$$c^{<t>} = \Gamma_u * \tilde{c} + (1 - \Gamma_u) * c^{<t-1>}$$

- The $*$ operation is element-wise multiplication.

- Therfore Γ_u, $c^{<t>}$ and $\tilde{c}^{<t>}$ all have the same dimensions.
- The Γ_u gate allows memorizing items like the "cat" for as long as needed, e.g. until the "was" token is found, in which case c can be replaced with $\tilde{c}$.
- It also solves the vanishing gradient problem for the most part.

LSTM

- Paper: "Long short-term memory" (hard read).
- More powerful than the GRU - but building on the GRU.
- Compared to the GRU:
  - It does not use Γ_r
  - $c^{<t>}$ is no longuer equivalent to $a^{<t>}$
  - Γ_u is split between Γ_u and Γ_f (the forget gate)
  - It has an explicit output gate Γ_o.
- Very good at memorizing values within long range dependencies, such as the GRU.

$$\tilde{c}^{<t>} = tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * tanh(c^{<t>})$$

In some variants there is a "peephole connection", adding $c^{<t-1>}$ to the ALL gate, where the i^{th} element of $c^{<t-1>}$ affects the i^{th} element of Γ_u,Γ_f and Γ_o.

Comparison with GRU:

- LSTMs are much older than GRUs!
- LSTMs are computationally more expensive, due to more gates.
- GRU are a simplification of the more complex LSTM model.

- No consensus over the better algorithm.
- GRU is simpler and easier to scale.
- LSTM are a standard and usually more flexible, but are loosing ground.

Bidirectional RNNs (BRNN)

- Forward probagation goes both forward and back in time, and there are separate activations for each direction for each instant t.
- BRNN work with GRU and LSTM (LSTM seems to be common).
- Able to make predictions in the middle of the sequence from future and past inputs.
- Does require the entire sequence to be input (does not work well for speech recognition in real time, for example).

Deep RNN

- RNNs can be stacked "vertically", where each layer l has the same number of time units t, and the output $y^{<t>}$ is connected as the next layer's $(l+1)$ input.
- $a^{[l]<t>}$ represents the activation of layer l at time t.
- Usually there aren't many layers - 3 is already quite a lot.

- In some architectures each output $y^{<t>}$ can be connected as the input to another neural network (e.g. densly connected).
- LSTMs and GRUs work in this architecture as well, and even BRNNs.

Updated Neural Networks layer indexing notation

- Superscript $[l]$ denotes an object associated with the l^{th} layer.
- Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.
- Superscript (i) denotes an object associated with the i^{th} example.
- Example: $x^{(i)}$ is the i^{th} training example input.
- Superscript $\langle t \rangle$ denotes an object at the t^{th} time-step.
- Example: $x^{\langle t \rangle}$ is the input x at the t^{th} time-step. $x^{(i)\langle t \rangle}$ is the input at the t^{th} timestep of example i.
- Lowerscript i denotes the i^{th} entry of a vector.
- Example: $a_i^{[l]}$ denotes the i^{th} entry of the activations in layer l.

Word Representation and embeddings

One-hot/1-hot representation + Each word is represented by a binary vector for the size of the vocabulary with a 1 at the position that corresponds to a specific word in that vocabulary. - The vectors generated do not encode semantic proximity. For example the meaning of apple and orange in the following setences when trying to find the next word: `I want a glass of orange _____` `I want a glass of apple _____`
Word embedding (featurized representation) + Each word is represented by high dimentional feature vectors that include a measure of distance between the word represented by the vector and a series of features or other terms. + Each word embedding (e.g. a column of the table below) can be named by the reference where the number is the index of the word in the vocabulary `e5391` or `e9853` for example for the first and second column respectively.For example the columns of the following table:

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1.00	1.00	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
...	...	...	...	...	...	...

- Word vectors can be compared to find close relationships between terms and this helps generalize. In

the example above, if we know that the missing word is "juice", the vectors for "apple" and "orange" should be close enough that a learning algorithm can infer the next word "juice" in both cases.

- Words can then be visualized together with t-SNE (plot high dimensional spaces into 2d or 3d spaces), where the word proximity is visible.

Using word embeddings

- Is possible to use "transfer learning" to learn word embeddings from a large body of text, for example 1 billing words from the internet, and then use those embeddings with a smaller 100k word training set.
- This also allows representing words with smaller (compared to 1-hot) dense vectors of embeddings (e.g. size 300 instead of 10k). Word representation sizes are no longer restricted to the size of the vocabulary.
- Name/entity task usually use BRNN (Bidirectional RNN).
- Optinal: Continue to finetune the word embeddings with new data.
- The term *embedding* is close to *encoding*, though embedding is a *fixed* encoding, meaning that the algorithm won't recognize a new word for which there is no embedding.

Properties of word embeddings

- Paper: "Linguise regularities in continuous space word representations"
- Allow finding analogies - e.g. by subtracting the word embeddings of "Man" and "Woman" or "King" and "Queen" (from the table above) we find that the main difference between them is the "Gender".
- With word embeddings we can use equations for finding the most appropriate words (**analogy tasks**):

$$e_{man} - e_{woman} \approx e_{king} - e_w$$

$$\text{Find word } w\text{: } argmax_w sim(e_w, e_{king} - e_{man} + e_{woman})$$

- The most common similarity (sim) used is the cosine similarity:

$$sim(u, v) = \frac{u^\intercal v}{\|u\|_2 \|v\|_2}$$

- Another possible measure is the squared ditance (euclidea), though it is a distance measure, not a similarity measure, so we would need to take its negative:

$$\|u - v\|^2$$

- The difference between the cosine similarity and using the euclidean distance it is how it normalizes for the lengths of the vectors.
- Researchers seem to find 30% to 75% accuracy (only correct analogies are counted for this metric).
- t-SNE - Maps high dimensionality (e.g. 300 to 2d).

Embedding matrix

- An embedding matrix E, has dimensions $n\_embeddings \times V$, where V is the size of the vocabulary (and will be referred as such going forward).
- The i^{th} row in E is e_i, e.g. e_{123}.
- If we have a 1-hot vector o_i of size $V \times 1$, then:

$$e_i = E \cdot o_i$$

- In practice, use a specialized function to lookup an embedding.
- In Keras there is an `Embedding` layer that allows retrieving the right embedding for the word more effeciently.

Learning word embeddings

- Paper: "A neural probabilistic language model"
- Over time, algorithms actually decreased in complexity!

Neural language Model

- The idea is to use E as a parameter matrix in a neural network.
  - The embedding from a window/history (a fixed number) of words are concatenated (e.g. 4 embeddings of size 300 create a vector of size 1200) and fed to a dense layer that is subsequently fed to a softmax layer. The word embeddings are treated as parameters in that they are part of the optimization process. The output of the neural network is the next word in a sequence. The input are the 1-one vectors for the words within the window.

  - Works because the algorithm tends to make words that are used together or interchangeably (in the same context) be represented closer together in the embeddings matrix, since these words tend to share the same *neighborhood*.

Context

- Either
  - 4 words on the right
  - 4 words on the right in the left (and predict the word in the middle) e.g. **CBOW** algorithm.
  - Or even simple contexts like the last 1 word

  - Better yet, the "nearby" 1 word: **skip-grams** also works well!

Word2Vec

- Paper: "Efficient estimation of word representations"
- **Skip-grams** - Given a **context** word, find a random **target** word within the neighbourhood (a window) of the context word.

- Example sentence:

```
I want a glass of orange juice to go along with my cereal.
```

Context	Target
orange	juice
orange	glass
orange	my

- Imagine a neural network to guess the target word given the context word. This is a hard problem. But the objective is not to do well on the classification problem, it is to obtain good word embeddings!
- So now we take the **context** word c (e.g. "orange") and a **target** word (e.g. "juice") t. (O is a 1-hot representation):

$$O_c \rightarrow E \rightarrow e_c \rightarrow softmax \rightarrow \hat{y}$$

$$\text{Softmax: } p(t|c) = \frac{e^{\theta_t^\intercal e_c}}{\sum_{j=1}^{V} e^{\theta_j^\intercal e_c}}$$

Where θ_t is the parameters associated with the output t, that is the chance that output t is the label.

The loss function will be:

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{V} y_i log(\hat{y}_i)$$

Even though there are parameters θ_t, the resulting embeddings E are still pretty good.

- Problem is the computational cost even for a 10k word vocabulary.
- A possible solution (in the literature) is to have a **hierachical** softmax classifier, that splits the words in the vocabulary into a binary tree and at each step tries to decide the branch to which the word should be long. This is actuall $O(log(V))$.
  - In practice the hierachical softmax classifier doesn't use a balanced tree, it instead is developed so that the most common words are on top, whereas the least frequent words are deeper.
  - This is superseded by negative sampling.

Notes on sampling context c

- Some words are extremely frequent (the, of , a, and, to, ...)
- Other are less frequent (orange, apple, durian, ...)
- So in practice the distribution P(c) is not random, it instead must balance out between more frequent and less frequent words.

Negative Sampling (Skip-Gram revised)

- Paper: "Distributed representation of words and phrases and their compositionality"
- Similar to skip-gram in performance but much faster.

`I want a glass of orange juice to go along with my cereal.`

- Similarly to skip-gram, we first choose a contex word c, then choose a random target word T and set a boolean "target?" flag to one (**positive example**).
- We then choose k random words from the vocabulary and set the "target?" flag to 0 (**negative examples**). It is not a problem if by chance the random word is in the neighborhood of the context word (such as the word "of" in the table below).
- $k \in [5, 20]$ for smaller datasets
- $k \in [2, 5]$ for larger datasets

The result is a table similar to the below:

Context	T Word	Target?
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0
...	...	...

The model uses the logistic function instead of softmax, and becomes:

$$P(y = 1|c, t) = \sigma(\theta_t^\intercal e_c)$$

$$O_c \rightarrow E \rightarrow e_c \rightarrow logistic \rightarrow \hat{y}$$

- This model must be thought of as the training a number of individual logistic regression models, as many as the size of the vocabulary.
- We train k (negative) + 1 (positive) logistic models in one iteration.
- The sampling of the negative examples as to be careful:
- Random sampling would pick up the more frequent words (of, the, and, ...) most of the time, thus under-representing less frequent words.
- Uniform sampling of each word in the vocabulary would give an even chance to each word $\frac{1}{|V|}$ (where V is the size of the vocabulary). But that would under-represent the most frequent words.
- SOLUTION - To sample words as a function of their frequency using the special ratio parameter 3/4:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{V} f(w_j)^{3/4}}$$

GloVe (global vectors for word representation)

- Paper: "GloVe: Global vectors for word representation"
- Not as used as Word2Vec or Skip-gram models, but gaining momentum due to its simplicity.

$$X_{ij} = \text{number of times } i \text{ appears in the context of } j$$

- Depending on how context and target word are defined it could happen that $X_{ij} = X_{ji}$

Model

$$\text{minimize} \sum_{i=1}^{V} \sum_{j=1}^{V} f(X_{ij})(\theta_i^\intercal e_j + b_i + b_j - log(X_{ij}))^2$$

- Where $\theta_i^\intercal e_j$ can be seen like the $\theta_t^\intercal e_c$ of the skip-gram model, though they are symmetrical and should be randomly initialized.
- b_i and b_j are bias terms.
- V is the size of the vocabulary.
- $f(X_{ij})$ is a weighting term, that:
  - Is 0 if X_{ij} is 0, hence we don't need to calculate $log(X_{ij})$
  - Compensates the inbalance due to words that are more frequent (e.g. the, is, of , a,..) than others (e.g. durian).
- In this model θ_i and θ_j are symmetric, and therefore to calculate the final embedding for each word we can take the average:

$$e_w^{(final)} = \frac{e_w + \theta_w}{2}$$

A note on the featurization view of word embeddings

- Methods used to create word embeddings matrix like Skip-Gram or GloVe do not guarantee that the embeddings are humanly interpretable. In other words, the rows of the embedding matrix will not have a humanly understandable meaning such as the rows of the motivational table that was previously used:

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1.00	1.00	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
...	...	...	...	...	...	...

- In algebraic terms it is not even possible to guarantee that the rows of the embedding matrix generated by these methods are orthogonal spaces.
- For example, in the case of GloVe, it is not possible to guarantee that there is a matrix A, given GloVe's parameters θ_i and e_j, such that:

$$(A\theta_i)^\intercal(A^{-\intercal}e_j) = \theta_i^\intercal A^\intercal A^{-\intercal} e_j = \theta_i^\intercal e_j$$

Sentiment classification

- Example 5 star reviews.
- Word embeddings can be extracted from very large data sets with 100 billion words.

Simple sentiment classification model

$$O_c \rightarrow E \rightarrow e_c \rightarrow Average \rightarrow softmax \rightarrow \hat{y}$$

- In this model the average is calculated for all the dimensions of the embedding vectors corresponding to a text's words, producing a single vector with the same dimensions of a single embedding vector.
- This embedding average vector is then fed to softmax output layer with as many units as the classes that we are trying to identify (e.g. 5 for 5 star review systems).
- It completely ignores word order, thus having bad results when order matters.

RNN for sentiment classification

For each word at position t:

$$O_c^{<t>} \rightarrow E^{<t>} \rightarrow e_c^{<t>} \rightarrow \text{RNN } a^{<t>} \rightarrow softmax \rightarrow \hat{y}$$

- We probably want to take the softmax value for the last word as the final output.

Debiasing word embeddings

- Paper: "Man is to computer programmer as woman is to homemaker? Debiasing word embeddings"

Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model. e.g.:

```
Man:Computer_Programmer as Woman:Homemaker
```

```
Father:Doctor as Mother:Nurse
```

Even if the same embeddings work in the case of:

```
Man:Woman as King:Queen
```

Addressing bias in word embeddings

1. Identify the bias direction

- For example for gender, take a few gender embedding pairs for definitional words as: $e_{he} - e_{she}$
  $e_{male} - e_{female}$
  $e_{grandfather} - e_{grandmother}$
  $e_{...} - e_{...}$

- Take the average of such embeddings, to identify the bias direction (in practice this is done with **SVD - Single Value Decomposition**, since the bias direction can be multidimensional)
- The Non-bias direction is orthogonal to the bias direction.

2. Neutralize: For every word that is not definitional (e.g. not like grandmother and grandfather), project (in the non-bias axis) to get rid of bias.
3. Equalize pairs - For example after projecting the (non-defenitional word) "babysitter" in the non-bias axis, move "grandmother" and "grandmother" so that they have the same distance from the non-bias and bias axis, hence having the same distance from the word "babysitter".

Problem: What words are definitional? * Most words are not definitional for a specific bias * It is possible to train a classifier to identify words that are definitional for a specific bias (e.g. grandmother/grandfather for gender).

This is an ongoing area of research.

Basic Sequence Models

Sequence to sequence models

- Paper: "Sequence to sequence learning with neural networks"
- Paper: "Learning phrase representations using RNN encoder-decoder for statistical machine translation"

Example: Machine translation

$$\text{encoder RNN (in language A)} \rightarrow \text{ decoder RNN (out language B)}$$

Example: Image captioning

$$\text{Conv Net (in cat picture)} \rightarrow \text{ decoder RNN (out img description)}$$

Picking the most likely sentence

continuing with machine translation example

Say we are trying to translate from French (A) to English (B) with the model:

$$\text{encoder RNN (in language A)} \rightarrow \text{ decoder RNN (out language B)}$$

- The output of the encoder module is the $a^{<0>}$ vector of the English language model.

- The resulting model is called a **conditional language model**, that maximizes the conditional probability of an English sentence given a French sentence:

$$\arg\max P(\hat{y}^1, ..., <\hat{y}>^{<T_y>} | x^{<1>}, ..., x^{<T_x>})$$

- Why not just use "greedy search", i.e. always chose the highest probability word on the decoder stage?
  - Because the sequence of all highest probability words is not necessarily the optimal sentence to maximize $P(\hat{y}^1, \cdots, <\hat{y}>^{<T_y>}|x^{<1>}, ..., x^{<T_x>})$.
- However, the total number of English words in a sentence is exponentially large. It is a huge space, which is why **approximate search** algorithms are used to try find the sentence that maximizes P.

Beam search algorithm

1. On the decoder module, take the most likely first B words from $\hat{y}^{<1>}$.
2. For each of the first B words selected, feed $\hat{y}^{<1>}$ as the input for the next stage $t = 2$ in order to get $\hat{y}^{<2>}$, and calculate the top B combinations of $\hat{y}^{<1>}$ and $\hat{y}^{<2>}$ that maximize:

$$P(\hat{y}^{<1>}, \hat{y}^{<2>}|x) = P(\hat{y}^{<1>}|x)(\hat{y}^{<2>}|x, \hat{y}^{<1>})$$

3. So now we get 3 combinations of $\hat{y}^{<1>}$ and $\hat{y}^{<2>}$, and for each of these combinations we want to find the top B combinations of $\hat{y}^{<1>}$, $\hat{y}^{<2>}$ and $\hat{y}^{<3>}$ that maximize $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \hat{y}^{<3>}|x)$.
4. Repeat the step above until we get the top B combinations that maximize $P(\hat{y}^{<1>}, ..., \hat{y}^{<t>}|x)$, and finally select only the most likely (since we now have the entire sequence).

Note that this algorithm implies creating B copies of the network at each stage, which is however trivial (for B not too large, e.g. 3).

- If $B = 1$ then Beam search becomes a greedy algorithm, but with $B < 1$ the algorithm finds better results.

Refinements to Beam search

Length normalization

Generalizing the conditional probability used by the Beam search algorithm:

$$\arg\max_y \prod_{t=1}^{T_y} P(y^{<t>}|x, y^{<1>}, \cdots, y^{<t-1>})$$

In practice, using sum of log we get the same result, with more numerically stability (less chance of overflow):

$$\arg\max_y \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \cdots, y^{<t-1>})$$

However, this target function has the problem that long sentences have low probability, so unnaturally short sentences are preferred. To solve this one thing to do is to normalize by the number of words:

$$\arg\max_y \frac{1}{t_y^\alpha} \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \cdots, y^{<t-1>})$$

Note that using $T_y^\alpha = T_y \times \alpha$ where α is e.g. 0.7 (heuristic/hack) is just a softened replacement of T_y that has been shown to work in practice.
+ This is sometimes called normalized **log likelihood normalization**.

Choosing Beam width B

- Large B: better results, slower (high memory usage)

- Small B: worse results, faster (less memory usage)
- Typical values of B are 10 for production. 100 Would be very large for production, but in research settings usually values of 1000 to 3000 are used.

Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find the exact maximum for $\arg\max_y P(y|x)$.

Error analysis on Beam search

Given an input sequence x, a machine translation algorithm with compute a translation $\hat{y}$ while a human can compute a different translation y^*. If it is found that the machine translation is wrong (in a meaninful way), we can try to find if the cause is either:

- Case 1: $P(y^*|x) < P(\hat{y})$
  - Conclusion: Beam search is at fault - we could try to increase B for example.
- Case 2: $P(y^*|x) \leq P(\hat{y})$
  - Conclusion: RNN model is at fault

We could try to perform error analysis (looking at a batch of examples with bad translation) to try to find which of these problems is more prevalent, in order to decide what we should spend our time troubleshooting.

Note that $P(y^*|x)$ and $P(\hat{y})$ should be obtained using the same target function described in the previous sections (e.g. log likelihood normalization).

Bleu Score

- To measure accuracy of different translation of the same sentences we use the **Bleu Score**.
- Paper: "Bleu: A method for automatic evaluation of machine translation"
- Bleu stands for "Bilingual Evaluation Understudy" - In the theater world "understudy" is someone who learns and can take the role of a more senior actor if necessary.
  - In machine translation it means that we find a machine generated score that can replace human generated translation references.

Given reference human translations $R_1, \cdots, R_n$, and the machine translation MT for a given text/sentence, the Bleu score can be generally formalised as P^n, or "Precision" for "n-grams" (unigrams, bigrams, etc.) as:

$$P^n = \frac{\sum_{\text{n-grams} \in \hat{y}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-grams} \in \hat{y}} \text{Count}(\text{n-gram})}$$

Where:

- $\text{Count}_{\text{clip}}$ - is the maximum number of times that an n-gram present in the machine translation MT appears in any of the reference translations $R_1, \cdots, R_n$
- Count - is the number of times a specific n-gram is present in the machine translation MT.

Bleu details:

P_n = Bleu score on n-grams only

Combined Bleu score (using up to tetragrapns $n = 4$):

$$\text{BP} \exp(\frac{1}{4} \sum_{n=1}^{4} P_n)$$

Where BP is the **brevity penalty**, that penalizes translations that are shorter than the original:

$$\text{BP} = \begin{cases} 1 \text{ if MT\_output\_length} < \text{ reference\_output\_length} \\ \exp(1 - \text{MT\_output\_length/reference\_output\_length}) \text{ otherwise} \end{cases}$$

In practice there are multiple open source implantations. The Bleu score is widely used today.

Attention Model

- Paper: "Neural machine translation by jointly learning to align and translate."
- With traditional models the Bleu score tends to fall with the length of sequences.
- By translating long sentences in small chunks (like humans tend to do) it is possible to retain a high Bleu score independently of the size of the sentence.

BRNN (Bidirectional RNN) are normally used for machine translation where **Attention Models** are usually employed. With that in mind:

- Timesteps in the encoder section are referred to by t'
- Timesteps in the decoder section are referred to by t
- The two sets of activations of the encoder section are $a^{<t'>} = (\overrightarrow{a}^{<t'>}, \overleftarrow{a}^{<t'>})$
- $\alpha^{<t,t'>}$ = amount of "attention" used to decoder output $y^{<t>}$
- The input $C^{<t>}$ (or context) to each decoder unit/timestep is a weighted average of the "attention" and the outputs of the encoder units, thus allowing each decoder output to take into consideration the closest set of words to the translated word being generated:

$$C^{<t>} = \sum_{<t'>} \alpha^{<t,t'>} a^{t'}$$

The $\alpha^{t,t'}$ parameter is in turn the application of a softmax function to the exponentiation of another paramenter $e^{<t,t'>}$:

$$\alpha^{t,t'} = \frac{exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} exp(e^{<t,t'>})}$$

Finally, $e^{<t,t'>}$ is a parameter learned by a simple neural network (e.g. 1 dense layer) with inputs $S^{<t-1>}$ (the previous decoder state) and $a^{<t'>}$.

- The downside of this algorithm is that it runs in quadratic time, that is $T_x \times T_y$, which is asymptotically $O(n^2)$. This might be an acceptable cost, assuming that sentences are not that long.
- This technique has also been applied to other problems such as **image captioning**:
  - Paper: "Show, attend and tell: Neural image caption generation with visual attention"
- Another example is date normalization (e.g. "July 20th 1969" $\rightarrow$ 1969-07-20)

Speech recognition

- time domain and spectral features
- With deep learning phonemes are no longer needed!
- Commercial systems are now trained from 10k to 100k hours of audio!
- Attention model is applicable

CTC cost for speech recognition

- Paper: "Connectionist Temporal Classification: Labeling unsegmented sequence data with recurrent neural networks"
- CTC means "Connectionist temporal classification"

The first part of an RNN for speech recognition (normally a BRNN), uses a large number of input features and also possibly has a relatively large resolution in time, meaning that many data points are generated per time unit. Therefore the following is an example of a valid output for the model of that first part:

`ttt_h_eee___ ___qqq___ii...`

- Underscores are "blank" characters, not "spaces".
- *Basic rule*: collapse repeated characters not separated by blank

Trigger Word Detection

- Examples: "Okay Google", "Alexa", "xaudunihao" (Baidu DuerOS), "Hey Siri"
- Still a research area.

One possible solution:

- Create an RNN that inputs sound spectral features, and always outputs 0 except when it has detected the end of a trigger word in which case it outputs 1.
  - Creates an imbalanced training set, with a lot more 0s than 1s. Thus "accuracy" is not a good performance metric in this case because a classifier that always returns 0 can have more than 90% accuracy. F-score, precision and recall are more appropriate.
  - One possible hack is to make it output a series of 1s after the trigger word, not just one 1, though that doesn't solve the problem definitively.

Additional machine learning topics to check

- Best papers in the ICLR proceedings (ICML are also good)
- Books by Yoshua Bengio (his contributions to ML are everywhere!)
- Restricted Boltzmann machines
- Weigstein altorithm - unsupervised
- Neural network partitioning and consensus voting (new paper) - capsules
- Adversarial neural nets
- variational auto encoders
- Atention - allows comparing different data structures, not just vectors to vectors... Yoshua Bengio
- Checkout ImageNet - the world's most pretigious computer vision challenge
- Checkout OpenAI
- Checkout Andrej Karpathy's blog
- Andrew Ng worked in speech recognition for many years - check his work
- Deep reinforcement networks

The End of the deeplearning.ai specialization!