Improving Deep Neural Networks Course

Hyperparameter searching: It is impossible to guess the best set of hyperparameters settings in advance for a model. A good model can be found only through a **fast and iterative** "try and error" process in which we try to find progressively a good set of hyperparameter settings.

Model selection/evaluation: A way to compare different models trained with different hyperparameter settings is using the train/validation/test technique. First, we train the models with training set using different hyperparameter settings. Then, we evaluate the models with the validation set and select the one with the best performance. Last, to take an unbiased performance estimate of the best model we use the test set.

Training/Validation/Test split ratio: When we have a relatively small dataset the traditional split ratios are 70%/30% for training/test sets or 60%/20%/20% for training/validation/test sets. However, **in deep learning** we use a large number of examples (e.g. 1M) and different ratios can be used, e.g. 99.5%/0.25%/0.25% or 99.5%/0.4%/0.1% or 98%/1%/1% for training/validation/test sets. Also, in the machine learning world when we do not want to use a validation set then the test set plays the same role as validation set for model selection.

Mismatched Train and Validation/Test sets distributions: A **new trend in deep learning** is to train a model using many examples crawled from the Web and validate/test it on images from a specific data source. For example, try to train a porn image detector using millions of training images from Web and then validate/test it on images that have uploaded registered users to Facebook. Such a technique uses images (training and validation/test images) from different distributions and can really help to build a model that generalizes well and detects effectively images with different quality, resolution or professionalism.

Bias/Variance tradeoff and deep learning: With **deep learning there is less tradeoff on bias/variance**. When we use a very complex model (e.g. deep NN architecture) it is possible to fit well both the train set (low bias) and at the same time the validation set (low variance) as long as the model is trained with a large number of examples. The worst case is to have a model with large loss both in training and validation sets (high bias and high variance). Last, we still can have models with high bias / low variance and vice versa (e.g. cases of underfitting / overfitting).

Regularization to prevent overfitting (high variance)

Both the loss regularization terms, dropout and early stopping (including other regularization techniques) are based on the assumption that when the weights of a model have small values then the model is simpler.

Using regularization terms in loss functions: By reducing the values of the trainable parameters of a model we can reduce overfitting. So, in general we use **L1** and/or **L2** regularization terms in loss functions to achieve it. Also, usually we do not perform regularization on the bias term trainable parameters cause they take a very small percentage of the total trainable parameters. So, the regularization terms operate mostly on the weights of the model. Specifically, in NNs we use a regularization term (similar to L2) that uses the sum of all **Frobenius norms** for each weight kernel matrix. The Frobenius norm of a matrix calculates the sum of all squared elements of it. Keep note that L2 regularization is same as gradient descent with **weight decay**.

Intuitions why regularization reduces overfitting in NNs: 1) If the regularization λ is large then both the trainable weights and neuron inputs will be small (around zero). Consequently, when the

neuron inputs are small and around zero the activation function – let's assume tanh – is mostly linear. So, using regularization the deep NN will use layers of roughly linear functions instead of complex non-linear functions and we know that a deep NN that uses linear layers is actually a linear model. For that reason the model will be more simple and overfitting will be reduced. **2)** When reducing the weight parameters and forcing them to be close to zero (with large λ regularization parameter) then many of the neurons calculate simpler functions since some of the input terms are roughly discarded (have low impact).

Dropout regularization: Instead of using the same NN architecture for learning each different example while training, dropout skips randomly from specified layers some neurons with their related ingoing/outgoing weights. So, this technique uses various random simpler versions of the NN to learn the different training examples. When the training is completed and we want to use the NN all the neurons are used with their latest weight values. We usually use a low keep probability in layers with many trainable parameters that might tend to overfit more and high keep probability (or no dropout at all) in layers with few parameters. The most well-known implementation of dropout is "**inverted dropout**" and can be used with vectorized implementation of forward/backprop computations without any changes in test mode. The dropout technique spreads out the weights and enforces neurons not to rely on a specific feature. More specifically, a neuron cannot rely on a specific input feature cause input features are dropped out in different iterations. So, the weight values of a neuron are spread across all inputs. This has the effect of shrinking the weights (similar to L2 regularization).

DropConnect regularization: This regularization technique is like Dropout but is applied only on the weights of neurons. Instead of skipping totally a neuron and all of its ingoing/outgoing weights we keep all neurons in the NN but skip some of their ingoing/outgoing weights. The side effect of the DropConnect is the same as Dropout. The NN is regularized since various simpler versions of it are trained on different examples.

Early stopping: In the beginning of a model training the weights have small random initial values. While training in case the model starts to overfit the weights start also to increase. This technique stops training to the point where the evaluation loss starts to increase (due to overfit). The best way to reduce overfitting is not to stop training as soon as validation error starts to increase but to use dropout and/or L1/L2 regularization to reduce the complexity of the model (especially L1/L2 regularization uses a formalized optimization loss function for the solution).

Why normalizing inputs? 1) When we normalize the input features to have values close to zero and zero mean then we help the learning algorithm to converge faster and easier to the solution. In logistic/linear regression the cost function is a bowl shaped function. When we normalize the input features the bowl shaped function is more round and gradient descent can fast converge to the global minimum. However, when there are many differences in feature scales the trainable parameters tend also to have differences in scale and the bowl shaped function is more like an ellipsis/elongated and gradient descent might oscillate until it find the global minimum. Also, we might need to use a smaller learning rate. **2)** It helps to avoid the gradient vanishing problem. **3)** All features are treated equally. So, it is impossible for some features to dominate other.

What are the problems of vanishing/exploding gradients: These two problems happen usually when training deep NNs with many layers. In "vanishing gradients" problem the gradients become very small for the earlier layers and in "exploding gradients" problem the gradients become very large (also in earlier layers). Both of these two problems make the training of the NN difficult cause very small or very large changes happen to weights and either the NN stops learning or deviates from the optimal solution.

When vanishing/exploding gradients problems can happen? The gradients can explode or vanish because they are propagated back with the chain rule, due multiplication operations. In backpropagation multiplications use layer weight matrixes and activation functions derivatives (too simplified). So, when a weight matrix in a layer has values < 1 or > 1 and is multiplied with the propagated gradients it will make the gradients for the earlier layers smaller or larger respectively. Also, depending on the input signal a neuron got in the forward propagation, its gradient might either vanish or explode (e.g. tanh/sigmoid vanishes with very large positive/negative inputs) and this can also make the gradients for the earlier layers smaller or larger respectively. So, the causes of gradient vanishing/exploding are either because the gradient of neurons can vanish/explode or the weights of layers are small/large. To partially solve the problem of vanishing gradients we can both use better activation functions that do not saturate on large inputs (e.g. ReLU, Leaky ReLU) and also initialize the weights of the layers in such a way so that net input signals will not be very large/small. The problem of exploding gradients can be solved partially with gradients clipping.

Weights initialization to face the vanishing gradients problem: By just normalizing randomly the weights of a NN and the input features cannot face the problem of vanishing gradients. For example, in case the input features (or activations from previous layer) are many then the operation of W^Tx (w1*x2 + w2*x2 + … + wn*xn) will calculate a big positive or negative net input signal and lead to the saturation of the neurons that use it (assuming that tanh or sigmoid activation function are used). We know that when neurons saturate because of large inputs then the gradients of their activation function vanish to zero. One way to partially solve this problem is to use modern weight initialization methods such as Glorot/Xavier, He, Lecun, etc. These methods initialize randomly with a Gaussian distribution the weights of the NN layers so that the variance of the weights of each layer is inversely proportional to the connecting input/output layer sizes. In other words when the no. of activations in the input or output layer increases then the variance of the weights decreases so that the calculated value of W^Tx remain small. The input features, weights and even layer activations should be normalized to have a zero mean and small variance to take advantage of the gradient of the activation functions (e.g. tanh and sigmoid are centered around zero).

Gradient checking issues: 1) Do not use it in training but only in debug mode cause it is computational expensive (we call the loss function for each weight twice). **2)** In case of error research on value differences between gradient approximations and backprop gradients. **3)** In gradient checking use the same loss that was used in training. **4)** Gradient checking cannot work with dropout cause the loss function is different on each iteration (random neurons are dropped out). We have to do gradient checking with dropout disabled.

Advanced Optimization Algorithms

Batch / Mini-batch / Stochastic Gradient Descent

In order to find a good deep learning model we need to be able to **fast** try various experimental ideas so that we can do model selection and select the one that performs best on the evaluation data. However, deep learning models need to be trained on a **large** no. of examples and with Batch Gradient Descent (BGD) this process can be really **slow (time complexity problem)**. For example, with m=50M examples in BGD we need to calculate forward-propagation / loss-calculation / back-propagation for all the examples with a vectorized implementation in each gradient descent iteration step. Also, when m is enormously large it might be impossible to support a vectorized implementation of forward-propagation / loss-calculation / back-propagation because of **memory limitations (space complexity problem)** either in CPU or GPU.

Solution: To be able to train deep learning models fast with a large no. of examples we can use Mini-Batch Gradient Descent (MBGD). This gradient descent variation splits the training data to no_batches=ceil(m/batch_size) batches of batch_size size (except the last batch if m%batch_size != 0) and performs no_batches gradient descent iteration steps for each batch using a vectorized implementation. So, in order to complete a training epoch we need to take multiple steps of gradient descent (in comparison to BGD which needs only one iteration step). This problem solves both the space and time complexity problems. The batch size is a hyperparameter that needs to be fine-tuned. Usually it is preferred to be a power of two value (e.g. 64, 128, 256, 512, 1024) and it is very important for a batch to be able to fit in CPU or GPU card so that can be processed fast.

BGD (or when batch size = m in MBGD)

- all the training examples are processed in a single gradient descent step
- on each iteration loss should be decreased
- **slow iteration speed**
- **needs less iterations**
- **supports vectorized implementation**
- **memory limitation**
- converges to the global minimum
- the gradient decent steps are stable toward the minimum
- prefer when using a small dataset (m <= 2000 examples)

SGD (or when batch size = 1 in MBGD)

- on each gradient descent step a single training example is processed
- loss may not decreased on each iteration
- **fast iteration speed**
- **needs more iterations**
- **no support of vectorized implementation**
- **no memory limitation**
- oscillates to solutions around minimum
- the gradient decent steps are noisy
- prefer on big datasets (usually is used on online learning systems)

MBGD (it gets the best from SGD and BGD)

- on each gradient descent step a batch of training examples is processed
- loss may not decreased on each iteration
- **medium iteration speed**
- **iterations needed something between BGD and SGD**
- **supports vectorized implementation**
- **no memory limitation**
- oscillates to solutions around minimum (more closer than SGD does)
- the gradient decent steps are noisy (less noisy than SGD)
- prefer on big datasets

Advanced Optimization Algorithms (Momentum, Adam, RMSprop)

There are various advanced gradient descent optimization algorithms (e.g. RMSprop, Adam, Momentum, etc) that can converge faster and/or be able to find a solution more close to the global minimum (in comparison with MBGD). A common feature all these algorithms share is that they

are based on the mechanism of **exponential weighted (moving) averages (EWA)** which reduces the oscillations while searching thus search more steadily for the global minimum.

> **EWA** is a method that can be used to generate a smoother version of a signal by calculating the weighted moving average using the current value and previous values. The method is parametrized with a **beta** parameter. When we increase the parameter **beta** then the previous values are taken account more (so we are averaging over more values) than the current value. This has the side effects of producing a more smoother signal which is shifted a bit to the right (there is a small latency in the new signal cause it adapts more slowly to the changes of the original one). Also, when we decrease the parameter **beta** then the previous values are taken account less than the current value and this has the side effects of producing a more noisy signal which is more susceptible to outliers. However, it adapts faster to the changes of the original signal.

In the context of EWA-based gradient descent optimization methods the **beta** is a hyperparameter and specifies how much of the gradients from previous iterations will affect the gradient calculation of the current iteration. So, each step is not independent from the previous ones. We actually calculate the EWA of the gradients and use that to update the model weights at each iteration step. So, it is like we are performing gradient descent using an accumulated **momentum** taken from the last iterations (e.g. with **beta**=0.9 for last the 10 iterations). This leads to a more stable optimization (not too susceptible to outliers) that doesn't make many oscillations and thus converges faster. Also, with these methods since we reduce the oscillations we are able to increase the learning rate to converge even faster and not diverge.

Momentum Gradient Descent

Although, stochastic and mini-batch GD algorithms converge eventually in a solution around the minimum the gradients taken on each step oscillate too much and point in various directions. There is too much noise while converging to the minimum. So, the path taken in the contours view seems like a zig-zag line instead of a straight-like line. One way to smooth the path taken is to calculate on each iteration step the EWA of gradients and use that to update the weights. Momentum Gradient Descent is the simplest form of Gradient Descent with EWA.

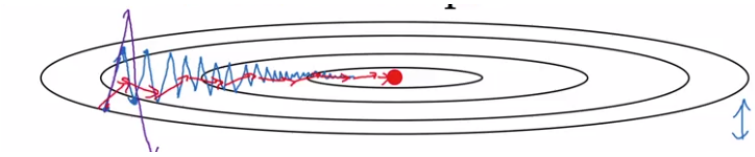
On iteration t:

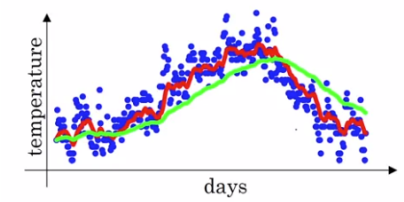
Compute dW, db on the current mini-batch

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

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

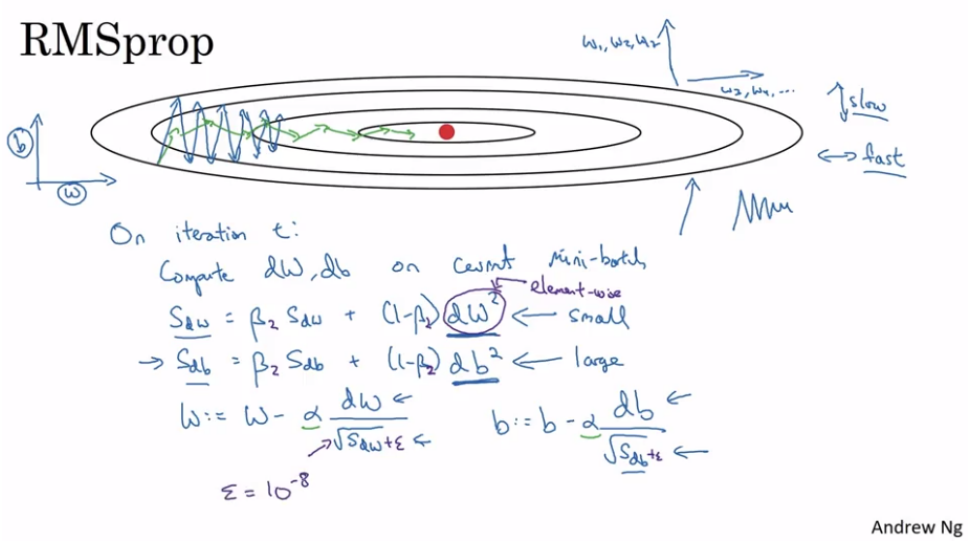
$$W = W - \alpha v_{dW},\ b = b - \alpha v_{db}$$





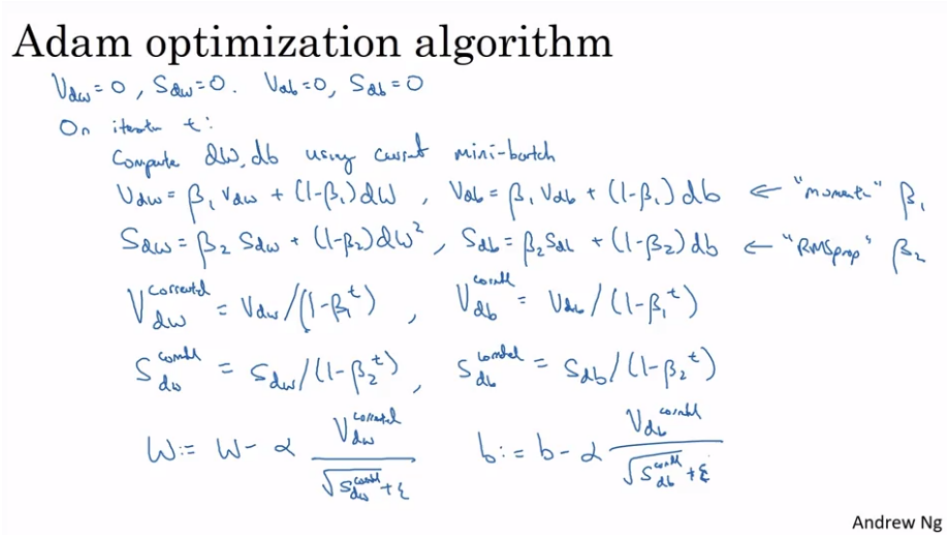
RMSprop (Root Mean Square Prop) – well tested and recommended in deep learning

Similar with Gradient Descent with momentum, RMSprop method is used to reduce any oscillations and smooth the path we take to find the minimum. The key difference of this method is that it makes large changes in weights that lead to a more "straight line" search towards the minimum and eliminates any changes in weights that can make the search noisy (path with oscillations). The way RMSprop works is to calculate on each iteration step the EWA of the **squared** gradients and use it to update the weights as follows:



Adam – well tested and recommended in deep learning

It is just a combination of RMSprop and GD with Momentum and we use two different beta hyperparameters, **beta1** for GD momentum and **beta2** for RMSprop (with default values beta1=0.9, beta2=0.999 and epsilon=10^{-8}). We also perform bias correction in EWA of RMSprop and GD momentum. Here is the combination of the algorithms:



Learning rate decay formulas

- a = a_0 / (1 + decay_rate * epoch_or_iteration)
- a = a_less_than_one_value^{epoch_or_iteration} * a_0

- $a = a_0 * (k / sqrt (epoch\_or\_iteration))$
- discrete staircase (100 steps a_0, next 100 steps $a_0/2$, next 100 steps $a_0/4$, etc)

Optimization problems in deep learning

- *It is unlikely to get stuck to local optima in deep learning*

  When optimizing traditional low-dimensional functions any zero-gradient points found are usually **local optima** (local minimum or local maximum). However, in deep learning in which we optimize high-dimensional cost functions any zero-gradient points found are usually **saddle points** which can act both as local minimum and local maximum at the same time. It is very unlikely in a space with e.g. 20000 dimensions (parameters of a cost function) to appear zero-gradient points which behave only as local minimum or local maximum. Usually, the zero-gradient points in high-dimensional spaces are saddle points which behave for some dimensions as local minimum and for some other as local maximum. So, it is unlikely to stuck in a local optima.

- *Problem of plateaus in cost functions*

  Surface areas of the cost function can exist that behave like plateaus. These areas can really slow down the optimization/learning cause they have very small gradient (near zero) for a long time. In order to descent more fast plateau areas we can use optimization algorithms that introduce acceleration, momentum, etc. Some candidates are: Adam, RMSprop, GD with momentum.

Systematic process to converge to a good setting for hyperparameters

Deep learning uses various hyperparameters (e.g. learning rate, momentum term, regularization weight, b1/b2/epsilon parameters in Adam/RMSprop optimizers, number of layers, number of hidden units, learning rate decay, batch size, etc).

Priority order (just a suggestion):

1. learning rate (very sensitive and most important hyperparameter)
2. momentum term (default 0.9), batch size (power of 2 sizes), no. of hidden units
3. no. of layers, learning rate decay, regularization weight
4. b1/b2/e (usually use default values, b1=0.9, b2=0.999, $e=10^{-8}$)

How do we select a good set of hyperparameters?

In the past, the **grid search** was very popular and used to search for a good setting of hyperparameters (e.g. using a 2D grid of values for C and σ in non-linear SVM with Gaussian kernel). The grid search tries all possible value combinations of hyperparameters and selects the combination that performs best. For example, using [1,2,3,4,5] values for hyperparameter1 and [10,20,30,40,50] values for hyperparameter2 the grid search will try all possible 25 value combinations. **However, in deep learning where the no. of hyperparameters is large this kind of search is not very exploratory.** The reason is because it is difficult to know in advance which hyperparameters are the most important for a specific problem. For example, if in a 5x5 grid the hyperparameter2 isn't so important (e.g. epsilon) then our set of 25 different hyperparameter settings isn't rich for hyperparameter1 (only 5 different values are used).

Solution: Random sampling instead of Grid search

Hyperparameter Space Exploration – first attempt: Instead of using a grid of values, searching randomly by sampling the hyperparameters space is more exploratory. So, if we sample randomly 25 settings from the hyperparameter space then although the hyperparamter2 isn't so important then our different values for hyperparameter1 would be 25 instead of 5. In the more general case if we have **n** hyperparameters we should randomly sample from the **n**-dimensional space.

Hyperparameter Space Exploitation – second attempt: In case we have already found a good hyperparameters setting by randomly exploring the hyperparameter space we can further spend our resources to perform a more fine search locally around the good setting to find an even better solution.

How do we sample random values from a range of values for a hyperparameter?

In general, we can simply sample uniformly a hyperparameter value from a valid [min, max] range. However, there are some cases where sampling will not spread uniformly and we might need to search on a different scale. The problem with sampling values at random from a range pops up when the min, max range limits are of two different orders of magnitude.

- **Case 1: No need to search on different scale:** When searching for hyperparameters such as #hidden_units or #hidden_layers and the range of values is e.g. **[50-100]** or **[2-8]** respectively, we can immediately uniformly sample values from the ranges.

- **Case 2: Sample on log scale:** When searching with ranges that min, max limits have different orders of magnitude, e.g **[0.0001-1]** or **[10, 10000000]** and uniformly sample values then we do not search uniformly the whole range of values. To fix this we need to search on the log scale as:

  1) new_from_min = log10(range_min)
  2) new_from_max = log10(range_max)
  3) r = sample uniformly from [min(new_min, new_max), max(new_min, new_max)]
  4) hyperparameter sample = 10^r

- **Tricky case when min, max limits are too close:** When searching for a hyperparameter such as beta from **EWA** using a range such as **[0.9-0.999]** and uniformly sampling values then we do not search uniformly the whole range of values. To fix this we calculate the 1-b range as [1-min, 1-max]=[0.1, 0.001] and then use the log scale in the new range to sample a random value as:

  1) new_from_min = log10(1-range_min)
  2) new_from_max = log10(1-range_max)
  3) r = sample uniformly from [min(new_min, new_max), max(new_min, new_max)]
  4) hyperparameter sample = $1-10^r$

Pandas vs Caviar approaches to hyperparameter searching process

When we have low computational power/resources, we prefer the "**Pandas**" approach in which we train/test a single baby model and gradually fine-tune it manually by changing its hyperparameters until we reach a desired performance.

When we have high computational power/resources we prefer the "**Caviar**" approach in which we can train/test in parallel multiple models by random sampling the hyperparameter space. At the end of all tries we select the best models and perform a finer search around these solutions.

Using Batch Normalization in NNs

For the same reasons we standardize the input data features of X to avoid gradient vanishing and help to learn the parameters faster, we can use batch normalization in the hidden layers of a NN. So, batch normalization helps to train a deep NN and learn the parameters faster. Furthermore, batch normalization helps to reduce the covariate-shift problem, a major problem in ML.

Batch normalization **standardizes the mean and variance of $z^{[l]}$ or $a^{[l]}$** of a single layer (or even more) using either all the training data in case of batch gradient descent or the current mini-batch in case of mini-batch gradient descent. Also, the mean and variance should not always necessary be $\mu=0$, $\sigma=1$. It might be better for some hidden units to use a different Gaussian distribution. So, batch normalization uses mean shifting and variance scaling parameters, called gamma and beta, which alter the distribution and are learnable by the optimizer. **When using batch normalization in a layer we can omit the bias vector (or set it to 0) for the hidden units because they are canceled out when we subtract the mean value.**

Covariate-shift problem when input data changes

The covariate-shift is difference between training data set distribution and test data set distribution. Normally we would expect them to come from the same distribution but this almost never happens. So we have to continuously update our models with latest train set, otherwise the prediction output will not be accurate. For example, let's assume we have trained a simple logistic regression cat classifier with black cats that achieves good accuracy and at some point in the production for some reason users start to test non-black cats. The model will not work well cause its weights were trained for black cats. The distribution of input has changed and we need to retrain the model with the new train data. There is strong couple between the model parameters and the input features. The mapping between $X \rightarrow Y$ needs to be retrained.

How batch normalization reduces the covariate-shift problem?

Now, assume we train a deep network and each layer tries to learn some representation of the data. Let's focus on a specific deep layer, say layer 30. The layer 30 tries to learn from its input features which are generated from the previous layers. However, while training, the input features change and the layer 30 also needs to. There is a strong coupling between the layers. **One way to create more independent layers, that are not so sensitive to input changes and learn independently is to use batch normalization. So, batch normalization reduces the problem of covariate shift problem and creates a more robust network.** Why that happens? Because batch normalization reduces the amount that the distribution of the input features changes. Even the exact values of the input features will change the mean and variance will remain the same (based on gamma and beta parameters).

How batch normalization is used at test time?

In the training phase for each mini-batch we calculate the mean and variance of the input of layers with batch normalization. At test time, we also need to normalize the inputs of the layers but for a single example. So, what mean and variance should we use for the layers? We cannot calculate the values somehow cause we have only one example. Also, we cannot

use the mean and variance from a specific mini-batch (because between different mini-batches the mean and variance are noisy). **Solution 1:** While training, for each layer we calculate the moving average (with exponential weighted averages technique) the mean and variance of the different mini-batches and at the end of the training we store the mean and variance for each layer to be used in the test time (along with the beta and gamma parameters). **Solution 2:** Use the whole training dataset with the trained network and calculate the mean and variance for each layer. Use the calculate mean and variances at test time.

Softmax regression and Softmax activation function

1. Softmax regression is a generalization of logistic regression used for classifying multiple classes (#classes > 2) using multiple linear decision boundaries. Softmax regression is a shallow model like logistic regression but uses #classes output units and the softmax activation function.

2. Each output unit models the probability that the given input belongs to the related class, $p(y=n, x; \theta)$. To model the probability for each output unit we use the softmax activation. The softmax activation function gets all inputs of the last layer and generates a vector of probabilities for all classes (the probabilities sum to 1).

3. Hardmax is the opposite of softmax and maps the input to the appropriate one-hot vector.

4. In a multi-label classification problem we can add an extra “other” class for the case or irrelevant examples (e.g. in a computer vision application: cat, dog, mouse, **other**).

Tensorflow

Idea:

1. Declare tensors for the trainable parameters as Variables (kernels, biases) and the input/output data (X/Y) as Placeholders (we should feed at each gradient descent step a different mini-batch).

2. Declare all the graph operations of our model for forward propagation and the calculation of cost function.

3. Declare the optimizer that minimizes the cost.

4. Create a session and initialize any values needed (weight, bias random initial values).

5. Through a loop run the optimizer multiple iterations.

    - The optimizer will do forward-loss-backward-update calculations.
    - At each iterations we can get results from any part of the graph.