- ## Hyperparameters Tuning

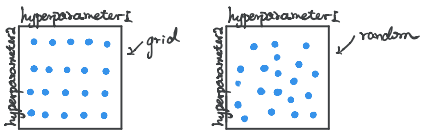
The following will show how to systematically organize hyperparameters tuning process. Depending on the algorithms, different sets of hyperparameters need to be tuned: α, β, β_1, β_2, ε, # layers, # hidden units, learning rate decay, mini-batch size etc.

(most important; second important; third important; rarely change, 0.9, 0.999, 10^{-8})

- How to search different values of a range of hyperparameters:

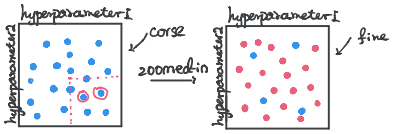
Random sampling:

Suppose we only tune 2 hyperparameters,



Traditionally, we create grid for hyperparameters to try out value systematically, but if we don't know which hyperparameter is more important and we can only try a few different values (e.g. 4 or 5 in 20 models), it's not very cost-effective. If we randomly sample values, we can try more different values in the same amount of models.

Corse to fine sampling:



If some points in a specific area perform better than the others, we can zoom in that area and sample (randomly) more density.

- How to pick the scale of hyperparameters:

At linear scale:

- # hidden units $n^{[l]}$: 50, ... 100
- # layer L: 2, 3, 4, ...

However, for other hyperparameters, they are more sensitive in one ranges than in other ranges (e.g. 0.999 → 0.9995 in β has way more impact on the model than 0.9 → 0.9005 in β). So we can't scale linearly anymore.

At log scale:

- learning rate α: 0.00..1, ... , 1 ⇒ log α ranges from $a\log 10$ to $b\log 10$

then $r = -b * np.random.rand()$ ⇒ sample α in log scale

$\alpha = 10^r$

- β: 0.9 .... 0.999

instead of sampling β directly, we sample $1-\beta$

$1-\beta$: 0.1 ... 0.001 ⇒ similar to α, $\log(1-\beta)$ ranges from $a\log 10$ to $b\log 10$.

then $r = -b * np.random.rand()$

$\beta = 1 - 10^r$

- ## Batch Normalisation

Batch normalisation enable hyperparameter search easier and makes the NN much more robust. What batch normalisation does is to normalise $z^{[l]}$ for any hidden layer to train $w^{[l]}$ and $b^{[l]}$ much faster (There are debates about whether we should normalise z or A but normalising z is done much more frequently so here I will present normalising z).

How to implement:

Given some intermediate values in NN in layer l $z^{[l]}: z^{[l](1)} \dots z^{[l](m)}$

$$\text{BN}\begin{cases} \mu = \frac{1}{m}\sum_i z^{(i)} \\ \sigma^2 = \frac{1}{m}\sum_i (z^{(i)}-\mu)^2 \\ z^{(i)}_{norm} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\varepsilon}} \\ \tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta \end{cases}$$

element-wise multiplication and addition, so different values for different zs in layer l.

→ trainable parameters: γ and β enable z to have different mean and variance (other than 0 and 1 in the normalised version), i.e. they control the range of value of z in the algorithm.

note: If $\begin{cases}\gamma = \sqrt{\sigma^2+\varepsilon} \\ \beta = \mu\end{cases}$ ⇒ $\tilde{z}^{(i)} = z^{(i)}$ (inverting the normalisation)

Adding Batch Norm to NN:

$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{BN}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow A^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow A^{[2]} = g^{[2]}(\tilde{z}^{[2]}) \longrightarrow \dots \longrightarrow A^{[L]}$$

$\tilde{z}^{[l]}$: same dimension as $z^{[l]}$

then for each iteration:

- compute forward prop on X (in each hidden layer, replace $z^{[l]}$ with $\tilde{z}^{[l]}$.
- use back prop to compute dw, $d\beta$, $d\gamma$
- update w, β, γ (using gradient descent, momentum, RMS, Adam etc.)

but in practise we will use programming framework so we won't need to implement these details ourselves. (tf.nn.batch-normalization)

note: since in the normalization step, the mean is subtracted and then later added back in, so effectively the parameter b is cancelled out. Therefore we can omit b, i.e. $z^{[l]} = w^{[l]} A^{[l]}$.

Working with mini-batches:

$$X^{\{1\}} \xrightarrow{w^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow A^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow A^{[2]} = g^{[2]}(\tilde{z}^{[2]}) \longrightarrow \cdots \longrightarrow A^{[L]}$$

$$X^{\{2\}} \xrightarrow{w^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow A^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow A^{[2]} = g^{[2]}(\tilde{z}^{[2]}) \longrightarrow \cdots \longrightarrow A^{[L]}$$

$X^{\{3\}}$ ...

⋮

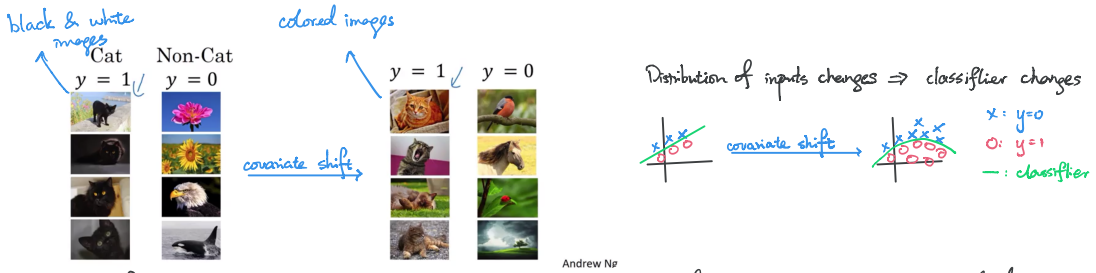
for t=1 ... minibatch_num:

- compute forward prop on X (in each hidden layer, replace $z^{[l]}$ with $\tilde{z}^{[l]}$.
- use back prop to compute dw, $d\beta$, $d\gamma$
- update w, β, γ (using gradient descent, momentum, RMS, Adam etc.)

Why does Batch Norm work:

- It normalizes inputs and makes the cost function bowl-like, which helps speed up the training.
- It also makes weights in deeper layer of the NN, more robust to changes to weights in earlier layers of the NN, by make each layer learn independently. The following will explain more about it:

  If the model has learned how X maps to Y, then when the distribution of X changes, we might need to retrain the algorithm to make correct predictions.



What it means for NN is that, if the input change, parameters in earlier layers change, which also leads to changes in later layers, i.e. from the perspective of the l^{th} layer, the hidden unit values change constantly and thus suffering from the problem of covariate shift.

What batch norm does is to reduce the amount that the distribution of these hidden unit values shift around, because parameters β and γ control the mean and variance. So even if the values of hidden units in earlier layers change when parameters update, the mean and variance will remain the same. Thus, these values will become more stable and the later layers won't have to adapt as much if the early layers change, i.e. each layer can learn independantly of other layers. This has the effect of speeding up of learning in the whole network.

Batch Norm at test time

Since at test time, we need to process the examples one at a time, so we won't have a mini batch of examples the same size as the training process. So we need a different way of coming up with μ and σ^2. In typical implementations of batch norm, what we usually do is to estimate these using an exponentially weighted average where the average is across the mini batches:

suppose we have mini-batches $X^{\{1\}}$, $X^{\{2\}}$, $X^{\{3\}}$, ...

for each batch, we compute the mean and variance: $\mu^{\{1\}[l]}$, $\sigma^{2\{1\}[l]}$, $\mu^{\{2\}[l]}$, $\sigma^{2\{2\}[l]}$ ...

then the exponentially weighted average of these means and variances become the estimates for the mean and variance of zs for layer l, i.e. $\mu^{[l]}$, $\sigma^{2[l]}$

finally, $z_{norm} = \frac{z - \mu^{[l]}}{\sqrt{\sigma^{2[l]} + \varepsilon}}$ and $\tilde{z}^{(i)} = \gamma z_{norm} + \beta$ (γ, β: learned in training process) for the test sample.

- **Multi-class classification**

The generalization of logistic regression called Softmax Regression, which makes predictions to recognize one of multiple classes. The Softmax regression has different number of units in the output layer to that of binary regression, which is equal to the number of classes:

let C = # classes, $n^{[l]} = C$

Then $A^{[l]} = \begin{bmatrix} a_1^{[l](m)} \\ a_2^{[l](m)} \\ \vdots \\ a_{n^{[l]}}^{[l](m)} \end{bmatrix}$, which output the probability of being each class, i.e. $A^{[l]} = \begin{bmatrix} P_r(\text{class } 1) \\ P_r(\text{class } 2) \\ \vdots \\ P_r(\text{class } C) \end{bmatrix}$

Since these are probabilities, $\sum_{i=1}^{n^{[l]}} a_i^{[l](m)} = 1$.

The Softmax regression also has different activation function for the output layer:

Suppose we have $z^{[l]}$, which is $n^{[l]} \times 1$

Then $A^{[l]} = \frac{e^{z^{[l]}}}{e^{z^{[l]}} \text{ summed up vertically}}$ → element-wise division

i.e. suppose $z^{[l](m)} = \begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \\ \vdots \\ z_{n^{[l]}}^{[l]} \end{bmatrix}$, then $e^{z^{[l](m)}} = \begin{bmatrix} e^{z_1^{[l]}} \\ e^{z_2^{[l]}} \\ \vdots \\ e^{z_{n^{[l]}}^{[l]}} \end{bmatrix}$, and finally $a^{[l](m)} = \begin{bmatrix} \frac{e^{z_1^{[l]}}}{\sum_i e^{z_i^{[l]}}} \\ \vdots \\ \frac{e^{z_{n^{[l]}}^{[l]}}}{\sum_i e^{z_i^{[l]}}} \end{bmatrix}$ ← normalise across different outputs so they sum to 1

The name softmax is in contrast to hardmax, which maps probabilities to only 0 and 1, e.g. $\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$

If $C = 2$, softmax regression is reduced to binary regression.

Training the Softmax classifier:

loss function: $\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{n^{[l]}} y_j \log \hat{y}_j$ → turns out to be a number because $y_i \log \hat{y}_i$ will become 0 in other $n^{[l]}-1$ classes, e.g. if $y^{(i)} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$, $\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -y_i \log \hat{y}_i$. Minimising it is equivalent to maximising $\hat{y}_i$.

cost function: $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

gradient descent: $dz^{[l]} = \frac{\partial J}{\partial z^{[l]}} = \hat{y} - y$ → in practise, we only need to specify forward prop and the programming framework will compute backprop.

- **Deep learning frameworks**

There are many deep learning frameworks that can help us apply deep learning more easily.

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow → we will show how to use tensorflow in another notebook.
- Theano
- Torch

Choosing deep learning frameworks
- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)