Course 2, week 3 : Tuning, BN, Programming Frameworks

Hyper-parameter tuning

Tuning process

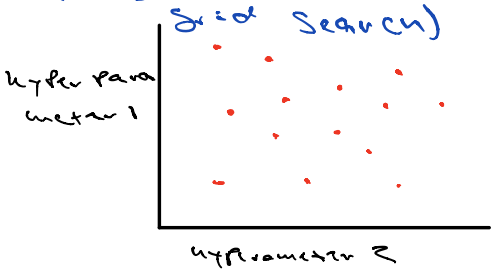
- α, β_1, β_2, ε, # layers, # hidden units, learning rate decay, mini-batch size, β

important | Second important | third important | never tuned

$\beta_1 : 0.9$, $\beta_2 : 0.999$, $\varepsilon : 10^{-8}$

- How to test different hyperparameters

(using random sampling instead of grid search)



↳ try out hyperparameters on a randomly chosen set in the hyperparameter search space

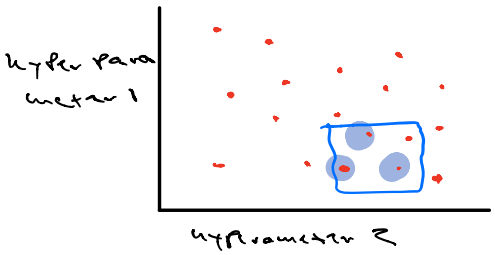
↳ this enables us to see what hyperparameters are the most useful to tune for the problem (which ones to focus on)

↳ for example, if hyper#1 was α and hyper#2 was ε, we would see that the value of α is way more important towards learning than ε

↳ By randomly sampling we get to try n different values of hyperparameters for each (where n is the # of models trained)

↳ in the example, we are searching over 2 hyperparameters, while in reality we will be searching in a high dimensional hyperparameter space.

- Coarse to fine



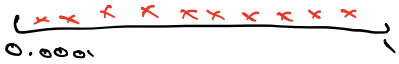
Once we find good hyperparameters in a broad search space, we might zoom into a smaller region and then sample within this space.

Appropriate Scale to pick hyperparameter

- Sampling at random doesn't mean sampling uniformly at random over the range of valid values.
- Instead for example picking the # of layers between 50 and 100:
  - $n^{[l]} = 50, \ldots, 100$

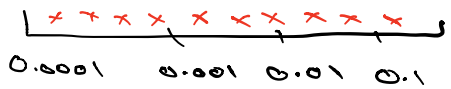
* this is a reasonable case in which we would search the space uniformly at random, but this search methodology is not true for every hyperparameter type.

- Say we are searching for $\alpha = 0.0001, \ldots, 1$



  - If we are sampling uniformly at random, 90% of the resources will be spent searching between 0.1 and 1, while only 10% of searches will be done between 0 and 0.1 (but we really want to explore this space in higher detail).
    - Use can change our number line to be logarithm

- Use to achieve a better search methodology.



- In Python we implement this by doing:

`r = -4 * np.random.rand()` ~ r will be a random # between -4 and 0

`a = 10**r` ~ making this between 10^{-4} ... 10^{0}

- Hyperparameters for exponentially weighted averages (say 10 values)
  - Lets say we want to search between $\beta = 0.9$ and 0.999 ~ last 1000
  - This can be achieved by sampling over $1-\beta$ which would make the range $0.1, \ldots, 0.001$ and then applying the method described above.
    - $\beta = 1 - 10^{r}$

Pandas vs. Caviar

- Intuitions about deep learning techniques may or may not transfer from one area, such as CV, to another, NLP.
  - This transfer is generally not applicable to hyperparameters
- To find the adequate parameters, we can take on two techniques: Babysitting one model or training many models in parallel.
  - The babysitting approach is called the Pandas approach (as Pandas put a lot of effort on the survival of one child)

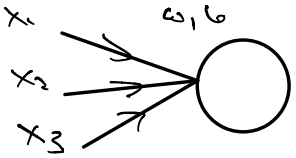
- training multiple models at a time is known as the caviar strategy (as in you get a lot of eggs and don't pay too much attention to any one of them, but hope that multiple will do well.

* the way to choose between these two approaches is a function of how many computational resources you have. If you have enough compute to train a lot of models in parallel, the caviar approach is optimal. In other applications, where the models are massive (such as in CV), the pandas approach might be a bit more realistic.

Batch Normalization

Normalizing Activations in a network

- Batch normalization makes the hyperparameter search problem much easier. It also makes the neural network more robust to the choice of hyperparameters selected. This enables the training of deep networks.



* as discussed before, normalizing the inputs can speed up learning

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

$$x = x - \mu$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$$ ← element wise squaring

$$x = x / \sigma^2$$ ← normalizing the dataset in terms of variance

- how do we extrapolate this towards a Deeper model
  - we are trying to normalize $a^{[1]}, a^{[2]}, a^{[3]}, \ldots, a^{[L]}$ to make the training of the intermediary weights and biases more effective (can we normalize $a^{[2]}$ s.o. as to train $w^{[3]}, b^{[3]}$ faster ? — this is what batch normalization does...)
    - we are going to normalize $z^{[2]}$ (post activation fnc)

- Implementing Batch Norm:
  - Given some intermediate values in NN $z^{(1)}, \ldots, z^{(m)}$
    - $\mu = \frac{1}{m} \sum_{i=1}^{m} z^{(i)}$
    - $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (z^{(i)} - \mu)^2$
    - $z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ ← added for numerical stability in case $\sigma^2 = 0$

this is all done for the inputs of a particular layer $z^{[l]}$

m = # of examples in a mini-batch

* every component of z will have a mean = 0 and a variance = 1. It doesn't always make sense for hidden unit to take on the Gaussian-like shape.

Hence we do the transformation,

- $\tilde{z}^{(i)} = \gamma \cdot z^{(i)}_{norm} + \beta$

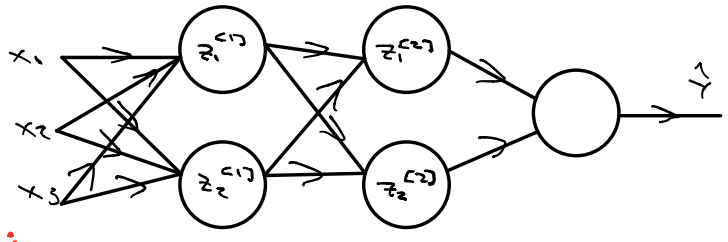
γ Gamma, β Beta — these are learnable parameters of the model

- the effect of gamma and beta is to set the mean of $\tilde{z}$ and variance to whatever we want it to be (what is most optimum given gradient descent).
  - hence if $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, the equation would reverse the effects of normalization and

then $\tilde{z}^{(i)} = z^{(i)}$.

Hence, overall, we are normalizing the mean and variance across layers to have some fixed mean and variance that we set.

Fitting Batch norm into a neural network



Parameters:

$$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow a^{[2]} \dots$$

↳ Parameters: $W^{[1]}, b^{[1]}, \beta^{[1]}, \gamma^{[1]}$

↳ this Beta has nothing to do with the hyperparam momentum Beta (this one is learned and the other is given)

* The Batch norm happens between computing z and a

- Beta and gamma are learned just like the weights and biases (through gradient descent, adam, RMS prop, momentum).
- typically in programming frameworks, batch norms can be applied through a method,
  - ↳ tensorflow: tf.nn.batch_normalization

- in practice, batch norm is applied with mini-batches,

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

We compute the mean and variances on the mini_batch, not the whole train set.

* Clarification: $z^{[l]} = w^{[l]} \cdot a^{[l-1]} + b^{[l]}$

  └ When doing batch norm, $b^{[l]}$ will be eliminated as a parameter as when we subtract the mean (μ) this will remove the bias (as the bias is an added constant to all the examples).

    └ Hence $b^{[l]}$ can be set to zero

      └ $z^{[l]} = w^{[l]} \cdot a^{[l-1]}$

      └ $z^{[l]}_{norm}$

      └ $\tilde{z}^{[l]} = \gamma^{[l]} \cdot z^{[l]}_{norm} + \beta^{[l]}$

        └ $\beta^{[l]}$ ends up playing the role of $b^{[l]}$

- Implementing Gradient descent (using mini_batch)

  └ for $t = 1 \dots$ num mini batches

    └ Compute forward prop on $X^{\{t\}}$

      └ in each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

    └ Use backprop to compute $dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

    └ Update parameters

* This also works with adam, RMSprop and gradient descent with momentum.

Why does Batch norm work?

- Needs for inputs in the same range $-1 \dots 1$, instead of different inputs having different scale ranges.
- Additionally it makes weights in later layers of the neural

network more robust to changes in the weights of earlier layers.

- covariate shift refers to the idea that if you've learned a function x to y mapping, if the distribution of x changes we might need to retrain the learning algorithm.
  - ↳ ie for example a recognition algorithm trained only on black cats and then presented with different color cats at test time
    - ↳ these distributions differ.
- how does the problem of covariate shift apply to a NN?
  - ↳ looking at the learning process from the third hidden layer, the nodes get some kind of values from earlier layers to perform a transformation that will then help with predicting $\hat{y}$ in deeper layers.
    - ↳ the inputs towards these deeper layers are constantly shifting through the training process because the outputs of the prior layers are being tweaked by changing weights and biases causing a **covariance shift phenomenon**
      - ↳ through batch norm, we can reduce the amount the distribution of these hidden later shifts around (making the inputs to the later layers more stable).
        - ↳ it ensures that even if the exact values

of $z_1^{[2]}$ and $z_2^{[2]}$ change, that the mean and variance of the previous layers will change less. Weakening the coupling between previous layers and later layers.

- Batch Norm's regularization effect
  - When each mini-batch is scaled by its own mean/variance, noise is added to the values of $z^{[2]}$ within that mini-batch. This acts similarly to dropout as noise is added to each layer's activation having an owner regularization effect. By adding noise to the hidden units, we are forcing the downstream hidden units to not rely too heavily on any particular hidden unit. This effect, however, is not particularly strong. So we might add other regularization techniques to our model like dropout or L2).

* additionally, the larger the mini-batch, the smaller the noise and therefore the smaller the regularization effect.

Batch Norm at test time

- We need a different estimate of μ, σ^2 at test time to process the data (as our original estimates are mini batch dependent).

- this is done by implementing a exponentially weighted average of μ and σ^2 across the mini-batches.
- $X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$

  $\mu^{\{1\}[l]} \rightarrow \mu^{\{2\}[l]} \rightarrow \mu^{\{3\}[l]}$

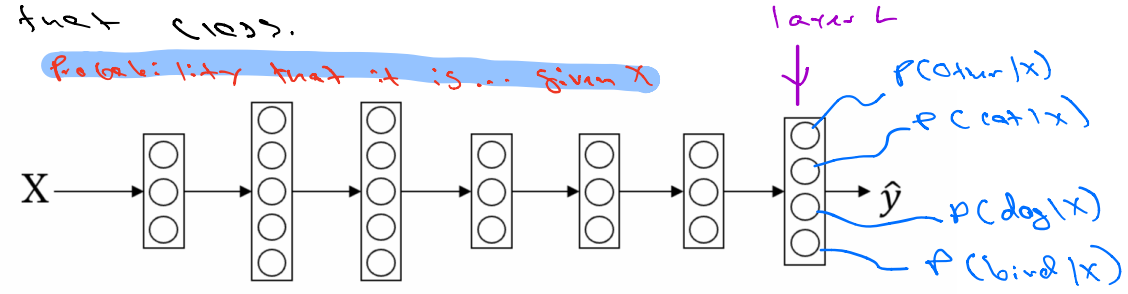
  - we use the exponentially weighted average to keep track of a the mean and variance at a particular layer.
    - at test time, we would compute,
      - $z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ and $\tilde{z} = \gamma \cdot z_{norm} + \beta$

      based on these estimated parameters

Multi-class Classification

Softmax Regression

- softmax is used in multi-class classification problems.
- C = # classes (0 indexed)
- we are building a neural network where the output layer has # units = to C (at $n^{[L]}$).
  - we want the number of units in the last layer to tell us what's the probability that an example is that class.



* $\hat{y}$ will be a (4,1) vector that gives each of the four probabilities (summing up to 1)

- the output layer is called a softmax layer

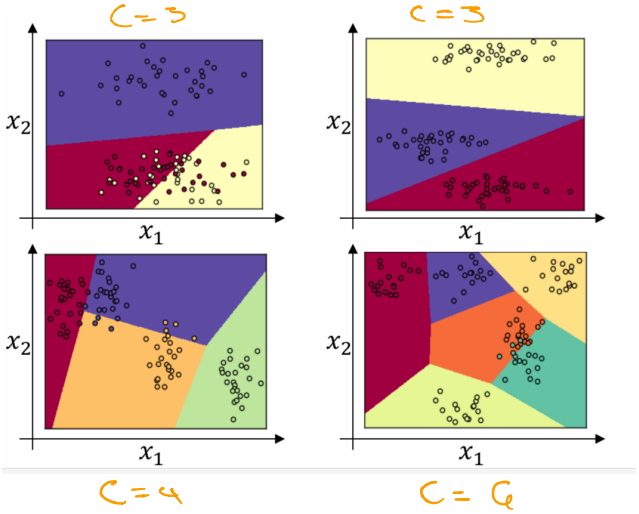
$$z^{[L]} = W^{[L]} \cdot a^{[L-1]} + b^{[L]}$$

- softmax activation function:

$$t = e^{z^{[L]}} \quad \text{(temporary variable)} \quad \sim \text{element wise operation} \quad (4,1)$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=4}^{4} t_i} \quad \sim \text{vector } t \text{, but normalized so that it sums to one} \quad (4,1)$$

- Softmax Decision Boundary Example



* this example shows a shallow neural network, hence, all decision boundaries are linear. Deeper nets can be used to train models with non-linear boundaries.

Training a softmax classifier

$C = 4$ $\quad (4,1)$

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

after softmax: $a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$

* a hardmax would take the largest position of z and add a 1 there, while zeroing out the rest, $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

In contrast, a softmax is a soft mapping where we get

probabilities of each class.

↳ the softmax regression generalizes logistic regression to C classes. If C = 2, the softmax basically reduces to logistic regression.

- Loss function to train neural network with softmax output

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$ ~ cat image given previous example

$$a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

↳ in this example the neural net didn't do really well as it gave a 20% chance of the example being a cat

$y_1 = y_3 = y_4 = 0$

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C=4} y_j \log(\hat{y}_j)$$

hence, $-y_2 \cdot \log \hat{y}_2$ because the rest of the $y_s = 0$.

$y_2 = 1$, therefore, $-\log \hat{y}_2$, the only way to reduce this loss is to make $\hat{y}_2$ as big as possible (where 1 is the max possible value for a probability) (maximum likelihood).

- Cost on entire train set

$$J(w^{[1]}, b^{[1]}, \ldots) = \frac{1}{m} \cdot \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

↳ we use gradient descent once again to minimize the cost

$$Y = [y^{(1)} \; y^{(2)} \ldots, y^{(m)}]$$

(4, m) ~ vectorized implementation

$$= \begin{bmatrix} 0 & 0 & 1 & \\ 1 & 0 & 0 & \ldots \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}$$

of outputs for example

* similarly $\hat{Y}$ will be a stacked

(4, m) dimensional matrix

- Gradient descent with Softmax

  └ at the last layer we compute: $z^{[L]} \rightarrow a^{[L]} = \hat{y} \rightarrow \mathcal{L}(\hat{y}, y)$

  └ Backprop step: $\boxed{dz^{[L]} = \hat{y} - y}$ ← (4, 1) vectors

  $\frac{\partial J}{\partial z^{[L]}}$