

Course 3, week 2: Structuring ML Projects

hyperparameter tuning

Tuning process

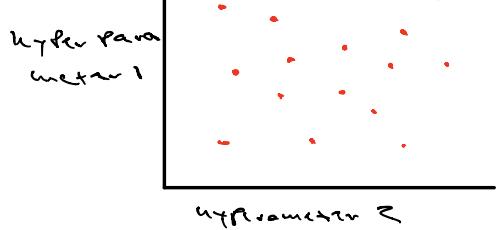
- $\alpha_1, B_1, B_2, \epsilon_1$ (→ layers), → hidden units, learning rate decay, mini-batch size, B

important second importance third importance never tuned

$$B_1 = 0.7, B_2 = 0.999, \epsilon = 10^{-8}$$

- how to test different hyperparameters

(using random sampling instead of grid search)



trying out hyperparameters on

a randomly chosen set in the hyperparameter search space

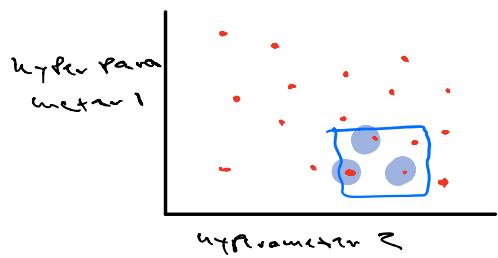
This enables us to see what hyperparameters are the most useful to tune for the problem (which ones to focus on)

↳ for example, if hyperparam 1 was α and hyperparam 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 in different values of hyperparameters for each (n is the # of models trained)

↳ in the example, we are searching over 2 hyperparameters, while in reality we will be searching in a higher 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.

Affordance Search to find high affordance

- Sampling at random doesn't mean sampling uniformly at random over the range of valid values.

- Let's for example picture the # of layers between 50 and 100:

$$\leftarrow n^{cl} = 50, \dots, 100$$



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

- Day we are searching for $\lambda = 0.0001, \dots, 1$



If we are searching 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).

We can change our number line to be logarithmic

use to achieve a better search methodology.



In Python we implement this by doing:

$r = -4 + np.random.rand()$ ~r will be a random #

$d = 10^r$ ~making this between -4 and 0
between $10^{-4} \dots 10^0$

• hyperparameters for exponentially weighted averages $\omega \sim 10^r$
 \sim values

Let's say we want to search between $\beta = 0.9$ and $0.999 \sim \text{last } 1000$

This can be achieved by sampling over 10^r
which would make the range $0.1, \dots, 0.001$
and then applying the method described above

$$\beta = 1 - 10^r$$

Faders 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 faders approach (as faders for a lot of effort on the survival of one child)

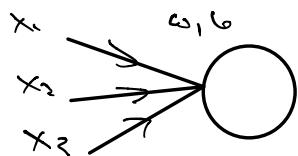
Training multiple models at a time is known as the **Covariation Strategy**, (e.g. train a lot of models and don't pay too much attention to any one of them, but note that multiple will slow down).

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

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

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

$$\hookrightarrow x = x - \mu$$

$$\hookrightarrow \sigma^2 = \frac{1}{n} \sum_{i=1}^n x^{(i)2} \sim \text{element wise variance}$$

$$\hookrightarrow x = x / \sigma^2 \sim \text{normalizing the}$$

dataset in terms of variance

- How do we extrapolate this towards a deeper model
 - ↳ We are trying to normalize $a^{(1)}, a^{(2)}, a^{(3)}, \dots, a^{(L)}$ to make the training of the intermediate weights and biases more effective (can we normalize $a^{(2)}$ so as to train $\mathbf{w}^{(3)}, b^{(3)}$ faster? — this is what batch normalization does...)

↳ We are going to normalize $\mathbf{z}^{(2)}$ (post activation (m))

• Implementing Batch Norm:

↳ Given some intermediate values in NN $\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}$

- ↳ $\mu = \frac{1}{m} \sum_{i=1}^m z^{(i)}$
- ↳ $\sigma^2 = \frac{1}{m} \sum_{i=1}^m (z^{(i)} - \mu)^2$
- ↳ $z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$

$m = \# \text{ of elements}$ in a mini-batch

ϵ added for numerical stability in case $\sigma^2 = 0$

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

* Every component of \mathbf{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,

$$z^{(i)} = \gamma \cdot z_{\text{norm}}^{(i)} + \beta \quad \begin{matrix} \gamma, \beta \text{ are learnable parameters} \\ \gamma \text{ beta of the model} \end{matrix}$$

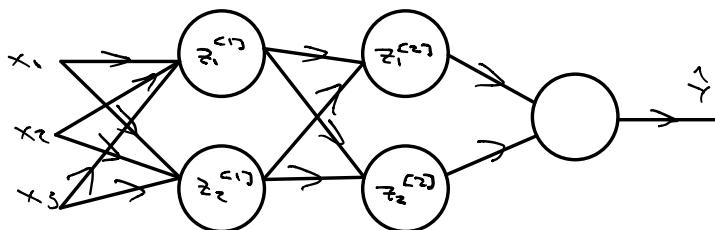
- The effect of gamma and beta is to set the mean of $\tilde{\mathbf{z}}$ and variance to whatever we want it to be (what is most optimum given gradient descent).

Hence if $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$, the equation would reverse the effects of normalization and

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

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

Fitting batch norm into a neural network



Matrixized:

$$x \xrightarrow{\substack{W^{(1)}, b^{(1)} \\ \text{Batch norm (BN)}}} z^{(1)} \xrightarrow{\gamma^{(1)}, \beta^{(1)}} \hat{z}^{(1)} \xrightarrow{\substack{W^{(2)}, b^{(2)} \\ \text{Batch norm (BN)}}} z^{(2)} \xrightarrow{\gamma^{(2)}, \beta^{(2)}} \hat{z}^{(2)} \dots$$

Parameters: $W^{(l)}, b^{(l)}, \gamma^{(l)}, \beta^{(l)}$

↳ this Beta needs nothing to do with the previous momentum param
(this one is learned and the other is given)

* the batch norm happens between computing \hat{z} and α

- Beta and gamma are learned just like the weights and biases (through Gradient descent, adam, RMSprop, 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,

$$x^{(1)} \xrightarrow{\substack{W^{(1)}, b^{(1)} \\ \gamma^{(1)}, \beta^{(1)}}} z^{(1)} \xrightarrow{\text{BN}} \hat{z}^{(1)} \xrightarrow{\alpha^{(1)}} \hat{z}^{(1)} \xrightarrow{\substack{W^{(2)}, b^{(2)} \\ \gamma^{(2)}, \beta^{(2)}}} z^{(2)} \xrightarrow{\text{BN}} \hat{z}^{(2)} \dots$$

We control the mean and variance on the mini-batch, not the whole train set.

*Clarification: $\tilde{z}^{[l]} = \omega^{[l]}.a^{[l-1]} + b^{[l]}$

↳ 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).

↳ Since $b^{[l]}$ can be set to zero

$$\tilde{z}^{[l]} = \omega^{[l]}.a^{[l-1]}$$

↳ $\tilde{z}^{[l]}$

$$\tilde{z}^{[l]}_{\text{norm}} = \tilde{f}^{[l]}. \tilde{z}^{[l]} + \beta^{[l]}$$

↳ $\tilde{f}^{[l]}$ ends up playing the role of $b^{[l]}$

- Implementing Gradient Descent (using mini-batch)

↳ for $t=1 \dots n$ mini-batches

↳ Compute forward pass on X^{t+3}

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

↳ Use backprop to compute $d\omega^{[l]}$, $d\beta^{[l]}$, $d\tilde{f}^{[l]}$

↳ update parameters

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

Why does Batch norm work?

- keeps gradients in the same range $-1 \dots 1$, instead of different numbers having different scale ranges.
- additionally \Rightarrow means 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 π to f mapping, if the distribution of x changes we might need to retrain the learning algorithm.

↳ take 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 later help with predicting y in later 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 **covariate shift phenomenon**.

↳ through batch norm, we can reduce the amount the distribution of these from layer shifts around (making the inputs to the later layers more stable).

↳ it ensures that even if the exact values

of $\mathbf{z}_1^{(l)}$ and $\mathbf{z}_2^{(l)}$ change, that the mean and variance of the previous layer 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 $\mathbf{z}^{(l)}$ within that mini-batch. This acts similarly to dropout as noise is added to each layer's activation having an overall 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 legitimate 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.

- $x^{2,3}, x^{2,3}, x^{2,3}$
 $\downarrow \mu_{2,3}[2] \rightarrow \mu_{2,3}[2] \rightarrow \mu_{2,3}[2]$

↳ 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,

$$\text{Z}_{\text{norm}} = \frac{\bar{z} - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad \text{and} \quad \bar{z} = \gamma \cdot \text{Z}_{\text{norm}} + \delta$$

(based on these estimated parameters)

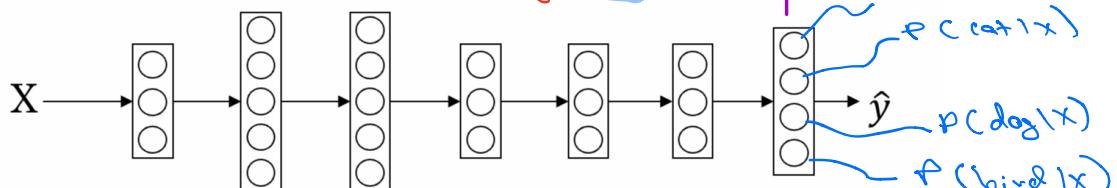
Multiclass Classification

Softmax Regression

- Softmax is used in multi-class classification problems.
- $C = \# \text{ classes}$ (0 indexed)
- We are building a neural network where the output layer has $\# \text{ units} = C$ ($\text{at } h^{(L)}$).

↳ We want the number of units in the last layer to tell us what's the probability that an example is from class i .

Probability that it is ... given X



* \hat{z} 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

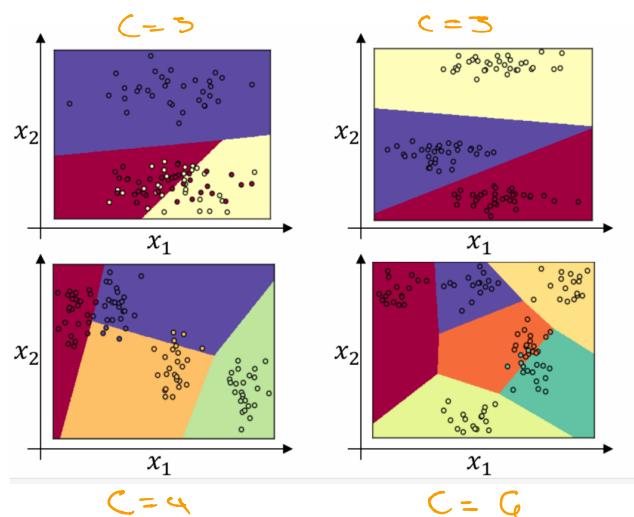
$$\hat{z}^{[L]} = W^{[L]} \cdot a^{[L-1]} + b^{[L]}$$

↳ Softmax activation function:

tentative variable $\hat{z}^{[L]} = e^{\hat{z}^{[L]}}$ ~ element wise operation $(4, 1)$

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

- Softmax Decision Boundary Example



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

Training a softmax classifier

$$C=4 \quad \hat{z}^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

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

* a hardmax would take the largest position of \hat{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 summarizes logistic regression to C classes. If $C=2$, the softmax basically reduces to logistic regression.

- Loss function to train neural network with softmax output

$$Y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \sim \text{Cat image given review ex-male}$$

$$\hat{Y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

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

$c=4$

$$L(\hat{Y}, Y) = - \sum_{j=1}^c Y_j \log(\hat{Y}_j)$$

Hence, $-Y_2 \cdot \log \hat{Y}_2$ because the rest of the $Y_j = 0$.

$Y_2 = 1$, 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

$$J(\omega^{(1)}, \omega^{(2)}, \dots) = \frac{1}{m} \cdot \sum_{i=1}^m L(\hat{Y}^{(i)}, Y^{(i)})$$

We use gradient descent once again to minimize the cost

$$Y = [\hat{Y}^{(1)} \ \hat{Y}^{(2)} \ \dots \ \hat{Y}^{(m)}]$$

$(4, m) \sim$ vectorized implementation

$$= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ of size } 4 \times m$$

* similarly \hat{Y} will be a stacked

$(4, m)$ dimensional matrix

- Gradient descent with Softmax

↳ at the next step we compute: $\hat{z}^{(l+1)} \rightarrow \hat{\alpha}^{(l+1)} = \hat{\gamma} \rightarrow \hat{z}(\hat{\gamma}, \gamma)$

↳ Backprop step: $\boxed{d\hat{z}^{(l+1)} = \hat{\gamma} - \gamma} \leftarrow (4, 1) \text{ vectors}$

$$\frac{\partial J}{\partial z^{(l+1)}}$$