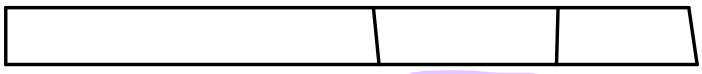
Week 1: Practical aspects of deep learning

Setting up your machine learning application

Train / Dev / Test sets

- as stated before, applied ML is a highly iterative process. We must make decisions on things like the # layers, # of hidden units, learning rates, and which activation functions to utilize, among others.
  - ↳ experiment → refine → iterate
- People working in applications such as vision, speech, NLP, structured data (search), often try to switch into another domain of the aforementioned. Typically, intuitions learned in the first domain, do not carry over into the new applications (such as a specific combination of hyperparameter tuning).
  - ↳ therefore, experimentation in order is key to optimize the models. (importance of iterativity)

- Data:



→ test set

training set

hold-out / cross validation / development set "dev"

- the work flow is to train multiple models on the training set, test with the dev set and after finding the best model we evaluate it with the test set.

- the final set is done to test that the model is producing an unbiased estimate.

- previously, the best practice was
  - 70/30%, train/test split
  - 60/20/20, train/dev/test split

* these ratios are reasonable if we have 100 to 10,000 examples in our datasets.

- In the modern big data era where we might have a million examples in a dataset, the trend is that the dev/test sets are becoming a much smaller % of the total.
  - the dev set is only used to determine which of the models is doing better, therefore, we don't need 20% of the dataset for comparison in huge datasets. 10,000 examples might be enough for test and dev sets in each.
    - 98/1/1 for 1 million examples.

- mismatched train/test distributions
  - example:
    - train set: high quality pictures of cats from websites online
    - dev/test set: Cat pictured from users using an app (maybe they're blurrier, less resolution)

* we have to make sure that the composition

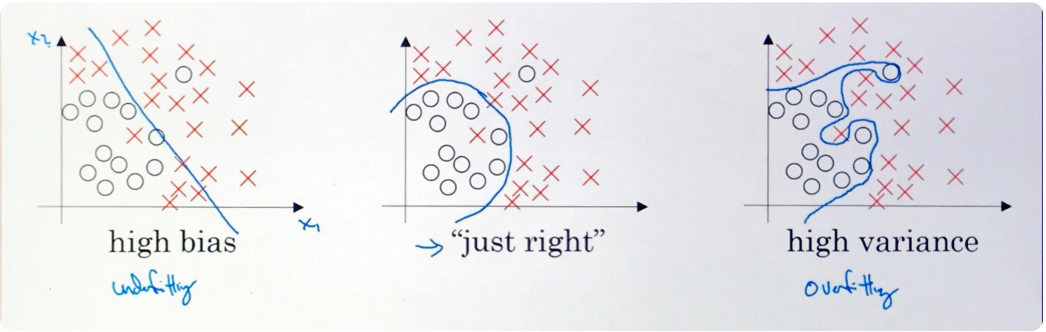
of our sets are based on the same distribution to accurately validate the performance of the model in its real-world application.

↳ this ensures better models.

- Sometimes, it might be okay to not have a test set.
  - ↳ but you won't have an unbiased estimate of performance.
    - ↳ sometimes you don't need one...
    - ↳ we might be overfitting to the test set if we are not careful as we are selecting our hyperparameters based on its results. (creating a biased model)

Bias / variance

- In the deep learning era, there's less of a (bias vs. variance trade off.



- In the high bias example, we are underfitting at the expense of having a lot of errors in our prediction cause we are trying to generalize too broadly

— In the high variance situation, we are fitting too closely to the training data. This makes our model perform worse in dev/test/application because our model is very constrained and does not generalize well.

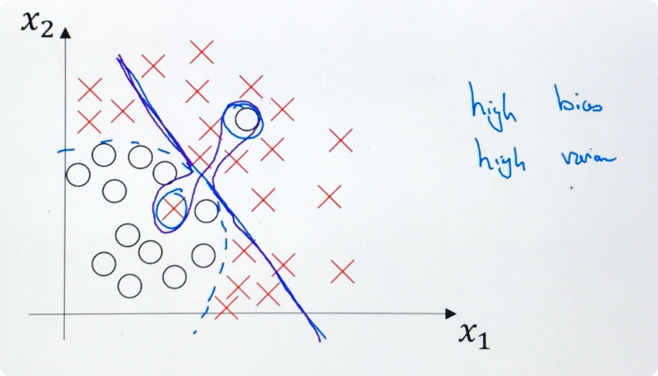
- In the previous example, we could visualize the boundary as we were drawing in a 2D plane. In higher dimensions, different metrics are required to perceive the bias/variance in our model.

human reference metric: ~ 0%

high bias & high variance

train set error	1%	15%	15%	0.5%
dev set error	11%	16%	30%	1%
	high variance	high bias (underfitting)		low bias and low variance

* this analysis is based on the fact that the Optimal (Bayes) error is near 0%.
  - this analysis is also assuming that both sets are taken from the same distribution.



- the high bias comes from the linear parts not containing the true quadratic behaviour of the data (underfitting the true function).

- the high variance comes from the high flexibility to fit these two outlier examples in the middle (as shown by the purple line)

* these effects are more common/possible in high dimensionalities.

Basic Recipe for machine Learning

- Systematic reduction when we have a high bias/variance.
- High bias (Based on training set performance)
  - ↳ try a bigger network with more layers or hidden units
  - ↳ train the model longer or use better more advanced optimization algorithms.
  - ↳ find a neural network architecture which is better suited for the problem. (this might work, but it's not guaranteed)

* with a big enough network, fitting the training set is generally possible if it's a problem that's generally solvable

- If an image is very blurry it might be impossible to fit it.
- If bayes error is not too high on the problem.

- once we have fit the data to the training set, we might ask if we have a high variance problem (by using the dev set).
  - to solve this,
    i. we might get more data (if possible)
    ii. try regularization to reduce overfitting.
    iii. try a more appropriate neural network architecture

- if we have a high bias problem, getting more training data will generally not help.
  - try the appropriate techniques mentioned above based on what problem you actually have.

- Back in the pre-deep learning era, we used to call it a bias/variance tradeoff because as we lowered one, the other one would vary inversely (increase).
  - back then, we didn't have many tools that just affected one of the two without impacting the other.
    - bigger network and more data serves as

these independent knobs to impact one without hurting the other.

- this is why neural nets really shine at supervised learning tasks.

* if we have well regularized data, training bigger networks doesn't hurt performance.
  - the only downside of the increase is a rise in computational time.

Regularization

- if we have a situation where overfitting is happening (high variance), regularization might be a way to solve it. Another way is to get more training data... which sometimes might not be possible.

- Developing these ideas with logistic regression,
  - min $J(w,b) = \frac{1}{m} \cdot \sum_{i=1}^{m} L(\hat{y}^{(i)}, y)$
    $+ \frac{\lambda}{2m} \|w\|_2^2$ ← regularization parameter

  - where the norm of w squared ($\|w\|_2^2$) = $\sum_{j=1}^{n_x} w_j^2 = w^T w$ (a square euclidean norm) ← of the parameter vector w

  * this is called L2 regularization because we are using the euclidian norm
  * this method of regularization is more popular than L1.

- Why do we regularize the parameter w, but not b in this expression. (such as $\frac{\lambda}{2m} \cdot b^2$)
  - In practice this can be done but it's omitted because w encompasses almost all the parameters (in high dimension) while b is only one parameter. (therefore this one parameter won't make much of a difference in practicality).

- L_1 regularization:
  - instead we add $+ \frac{\lambda}{2m} \sum_{i=1}^{n_x} |w| = \frac{\lambda}{2m} \cdot \|w\|_1$,
  - if we use L_1 regularization, w will end up being sparse. This means that the w vector will have a lot of zeros in it.

- λ (~ lambda) = regularization parameter (independent)
  - this is set using the development set

- Because the cost function is,
  - $\frac{1}{m} \cdot \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \cdot \|w\|_2^2$, there's a constant tug of war between optimizing the w's and making them small enough to not increase the cost function (since λ tend to find suitable tradeoff)

- L_2 in a neural network ~ (Frobenius Norm)
  - $J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m} \cdot \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \cdot \sum_{l=1}^{L} \|w^{[l]}\|^2$
  - $\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$

$i=1$ $j=1$

- Where, rows "i" should be the # of neurons in the current layer ($n^{[l]}$)
- and columns "j" of the weights of the matrix should be equal to the neurons in the previous layer $n^{[l-1]}$

- how do we implement gradient descent with this?
  - $dw^{[l]} = \ldots + \frac{\lambda}{m} \cdot w^{[l]}$
  - $w^{[l]} = w^{[l]} - \alpha \cdot dw^{[l]}$

* L2 normalization is also called weight decay because for each update of w we are subtracting by a $\alpha \cdot \lambda \, w^{[l]}$ term additional to what we used to get from Backprop.

Why does Regularization Reduce overfitting

- why is it that shrinking the frobenius norm will cause less overfitting? (intuition)
  - if we set lambda large, we will make the majority of weights close to zero as we are trying to minimize the cost function. (J)
    - this will zero out a lot of the impact of many hidden units in a neural net
    - meaning that the resulting neural network can be simplified into a smaller equivalent neural network stacked multiple layers deep.
    - this will take us from the overfitting

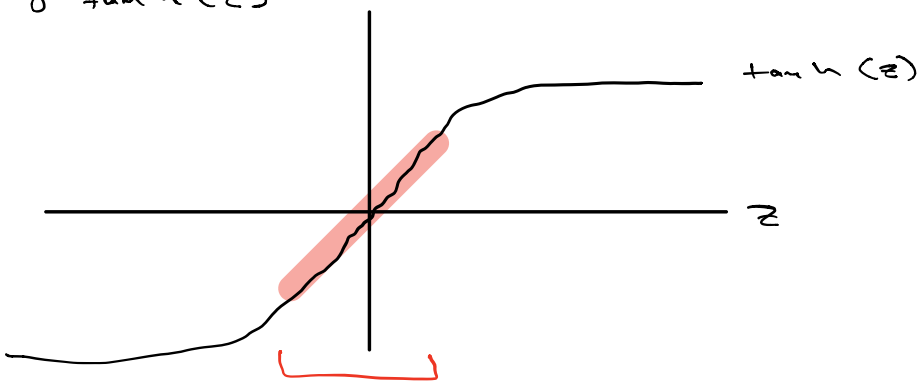
case towards a more general (simplistic) model (such as higher bias).

↳ the intermediate between these two is what we desire.

* ↳ by zeroing most hidden unit we are minimizing the individual effects of each unit and giving more weighting to the model as a whole.

↳ making it less prone to overfitting.

using $\tanh(z)$:



↳ so long as we keep the z value small, we are using the linear regime of the tanh function.

↳ if lambda is large, $W^{[2]}$ will be small, since $z^{[2]} = w^{[2]} a^{[2-1]} + b^{[2]}$ will also be small.

↳ hence every layer will be closely linear.

If every layer is linear (as seen previously), even a deep network will only be able to compute a linear function.

↳ we will not be able to fit huge

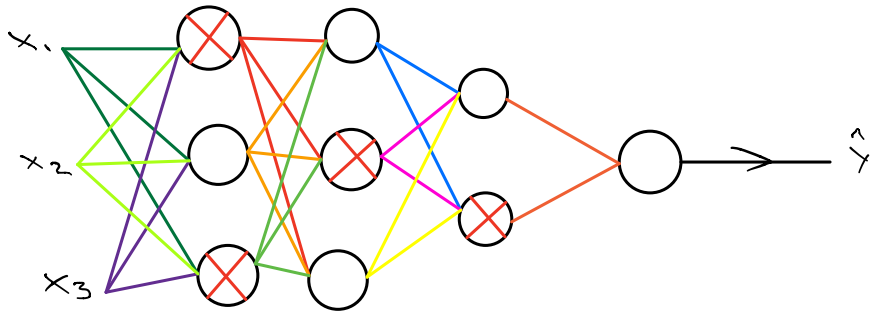
complicated decisions that lead to high variance / overfitting.

* to debug gradient descent, it is important to plot this new definition of the cost function ($J(..)$) with regularization vs the # of iterations. Not the previous definition.

↳ to see it decrease across iterations.

Dropout Regularization

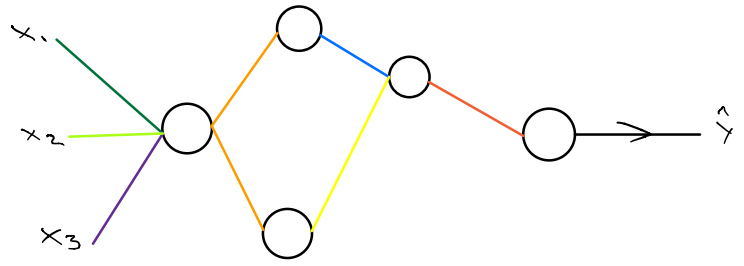
- with dropout we set a probability of eliminating a node in the neural network at each iteration of training in order to reduce overfitting.



- imagine that for each layer, there's a 0.5 chance of keeping a node. The resulting smaller/diminished network trained in the specific iteration of each forpagation looks like the resulting network below.

↳ on a different iteration, the nodes that get eliminated might be different as there's a

there's a random probability of each specific one dissapearing (0.5) in any given iteration.



- On each set of examples we would train the net under a different set of reduced nodes.

Implementing dropout ("Inverted dropout") ($l=3$, specific layer)

- dropout vector for layer 3,
  - $d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\_prob$
    - keep_prob is a number (such as 0.8) and it is the probability that a given hidden unit will be kept.
      - 0.2 chance of eliminating any one hidden unit in this example (for layer $l=3$)
    - d3 will be a matrix where there's a 80% chance that a node will be one and a 20% chance the node will be zero for a given iteration.
  - $a3 = np.multiply(a3, d3)$ * element-wise multiplication
    - the corresponding elements get zeroed out.
  - $a3 /= keep\_prob$
    - we scale up our activations by the given

keep_prob constant.

- take $l=3$ to have 50 units. Given the probability (0.8), 10 units will shut off. When computing $z^{[4]}$ for the next layer, ($z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$), we must scale $a^{[3]}$ by the keep_prob in order to not reduce the original expected value of $z^{[4]}$

  ↳ this line is known as the inverted dropout technique

* no matter what we set keep_prob to, inverted dropout ensures that the expected value of the next layer remains the same (because of the scaling)

  ↳ this makes testing easier as we cancel out the scaling problem.

- this technique is the most common implementation of dropout.
- using dp means that for different training examples we zero out different units. Given the random nature, we also zero out different units on different passes (iterations) of the training set.

  ↳ the vector zeroes out both in forward prop and back prop.

- at test time we do not implement dropout (we go back to business as usual). Because of the scaling this works and the expected values of the activations across layers don't change.

Understanding Dropout

- using dropout randomly knocks out units in a neural network making it as if we are working with a smaller neural network (which should have a regularizing effect in theory)
- Single unit Expressive: a next stage unit doesn't rely too heavily on any one feature as these can go away at random throughout the training step. This has the effect of spreading out the weights among the inputs.
  - Ensures the square norm of the weights of the individuals are minimized (reducing overfitting)
    - dropout has a similar effect to L2 regularization

Reminder: in general, the # of neurons of the previous layer = # columns and the # of neurons of the current layer is = to # of rows in the layer's weight matrix (Present layer, Previous layer)

- dropout can be done on the input layer, but it's often not done in practice.
- if we're worried about some layers overfitting more than others, we can set them to a lower keep_prob and let dropout do its thing
  - this tactic gives you more hyperparameters to tune when doing cross validation.

- Implementation tips:
  - many of the first successful implementations of dropout happened in computer vision ~ # of pixels
    - the input sizes are large, but we never have enough training data
- don't use dropout if don't not much overfitting
* major downside to dropout is that the cost function becomes not well defined
  - on every iteration we are randomly killing off different nodes. this makes it harder to check the actual performance of J in gradient descent.