2025-07-22T21-28-52
Tags: DLS

DLS C2 week 1

Train/dev/test sets

Dataset	Purpose
Train	Fit the model
Dev	Compare models and tune hyperparameters
Test	Final unbiased performance estimate

The dev set is also known as hold-out cross validation set.

Data splitting workflow:

1. Train models on training set.
2. Evaluate and select best-performing models via the dev set.
3. Evaluate the best model on the test set to get an unbiased estimate.

Common splits:

- 60/20/10 for up to 10,000 examples (previous era best train/test split)
- 98/01/01 for up to 1 million examples (big data era)

For examples more than a million examples, the common splits are:

- 99.5/0.25/0.25
- 99.5/0.40/0.10

The larger the dataset the smaller the dev and test percentages are.

Dev and test sets must come from the same distribution—improvements on performance are measured on the dev set.

Train set may have a different distribution due to DL algorithms having a huge hunger for training data.

It's acceptable to have no test set if there's no need for an unbiased estimate. In this case, you only have train/dev sets.

Remember, the dev set can't be a true test set as it doesn't estimate an unbiased estimate due to overfitting via optimization.

overfitting

1. Weights are updated to minimize the cost J.
2. If the model is fitting the train data well, updates lead to larger w values.
3. If training continues for too long, the model *overfits* by fitting noise (large w values) in the train data.

Bias/variance

Train set error	• Indicates bias.
Dev set error	• Combined with train error, indicates variance.

	Description	Performance	Visual example
High Bias	underfitting, model too simple	Poor on train and test	Straight line trying to fit a curved dataset
High Variance	overfitting, model too complex	Good on train, poor on test	Wiggly line fitting all points but fails to generalize

Condition	Train Error	Dev Error	Characteristics	Example
High variance	low	high	Good fit, poor generalization	01%/11%
High bias	high	similar	Poor fit even on training	15%/16%
High bias AND high variance	high	even higher	Poor fit, poor generalization (worst case)	15%/30%
Low bias AND low variance	low	low	Great fit, great generalization (ideal case)	0.5%/01%

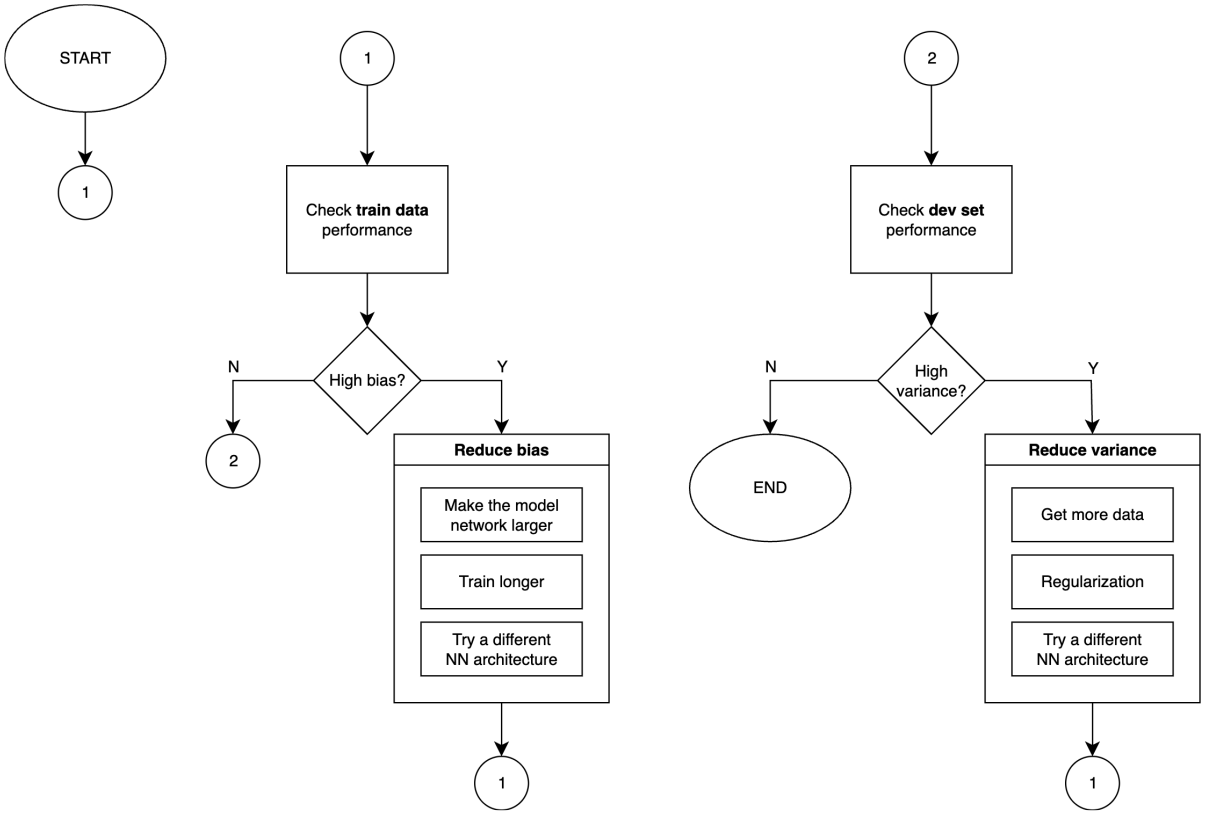
When analyzing bias and variance, assume that the optimal (Bayes) error is $\approx 0\%$. As humans can classify nearly perfectly.

Basic recipe for ML

Bias and variance have different solutions:

- If high bias, make the network larger.
- If high variance, add more data or regularize.

For high variance, the best action (if possible) is to get more data.
But this won't help if there's high bias.



In the previous era of ML, reducing bias increases variance and vice versa.

But today, there are less strict tradeoffs thanks to:

- larger models,
- bigger datasets, and
- proper regularization.

The usual cost of bigger network is mostly computational if regularized properly.

There is a slight bias increase when regularizing but minimal if network is large enough.

Regularization

Regularization prevents overfitting by adding a penalty to the loss function $\mathcal{L}$.

Regularization is a way to reduce high variance when getting more data is not feasible.

While regularization may not always reduce the training cost, it often leads to reduces the validation cost.

Tuning lambda

If λ is too small it has no effect.
If λ is too large it causes underfitting.

λ is the regularization parameter. You set this using the dev set.

Use `lambd` instead of `lambda` in Python as the latter is a reserved keyword.

Cost functions

For logistic regression: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||w||_2^2$

For neural networks: $J(w^{[1]}, w^{[1]}, \dots, w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{n}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}||w^{[l]}||_F^2$

L2 regularization

L2 regularization adds squared values of the weights to $\mathcal{L}$.

L2 regularization is also known as *weight decay* as it gradually shrinks weights.

Euclidean norm: $||w||_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^\intercal w$

Frobenius norm: $||w^{[l]}||_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}\left(w_{ij}^{[l]}\right)^2$

- The reason behind the summations: The shape of w is $(n^{[l]}, n^{[l-1]})$.

Use L2 regularization (ridge) when:

- you want to retain all features but reduce their weights.
- dealing with multicollinearity or noisy data.

It is more common.

Update rule in gradient descent changes:

$$
\begin{aligned}
dw^{[l]} &= (\text{from backprop}) + \frac{\lambda}{m}w^{[l]} \\
w^{[l]} &:= w^{[l]} - \alpha \cdot dw^{[l]} \\
&:= w^{[l]} - \alpha\big[(\text{from backprop}) + \frac{\lambda}{m}w^{[l]}\big] \\
&:= w^{[l]} - \frac{\alpha\lambda}{m}w^{[l]} - \alpha \cdot (\text{from backprop}) \\
&:= \big(1 - \frac{\alpha\lambda}{m}\big)w^{[l]} - \alpha \cdot (\text{from backprop})
\end{aligned}
$$

$\big(1 - \frac{\alpha\lambda}{m}\big)$ is the cause of the *weight decay*.

L1 regularization

L1 regularization adds the absolute values of the weights to $\mathcal{L}$.

L1 regularization ends up making the w vector have a lot of zeros.

L1 norm: $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w| = \frac{\lambda}{2m}||w||_1$

Use L1 regularization (lasso) when:

- fewer features, when many are irrelevent.
- feature selection is important.
- compressing models.

It is less common.

Why regularization reduces overfitting

L2 regularization adds a penalty term $\frac{\lambda}{2m}||W||_F^2$ to the cost function to discourage large weights.

How regularization reduces overfitting:

1. Large λ forces W to be close to zero.
2. Small W keep z to be relatively small (ignoring b).
3. If z is small, then activations—like $\tanh(z)$—act almost linear.
4. The network becomes simpler and less nonlinear.
5. A simple network behaving like a linear model.
6. A linear model is less prone to variance, leading to less overfitting.

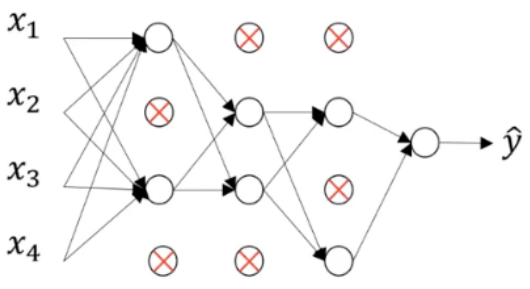
Remember when plotting the cost J, include the regularization term in the cost function.

Dropout regularization

Dropout is a regularization technique that randomly *drop* (set to zero) some neurons in each layer with a probability of $1 - \text{keep\_prob}$.

If $\text{keep\_prob} = 0.8$ each neuron has:

- 80% to remain, and
- 20% to be zeroed out.



Dropout regularization helps by:

- Forcing the network to rely less on specific neurons.

- Trains many smaller, different subnetworks on each iteration.

Inverted dropout

The most common implementation of dropout is the *inverted dropout*.

Inverted dropout procedure:

1. Generate a mask $d \sim \text{Bernoulli}(\text{keep\_prob})$ for each layer l.
2. Apply mask using element-wise multiplication to drop units.
3. Scale activations by dividing activations by keep_prob.

Given a layer $l = 3$:

```
import numpy as np

keep_prob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob # generate mask
a3 = np.multiply(a3, d3) # apply mask to drop units
a3 /= keep_prob # scale the activation
```

Scaling the activation via division by `keep_prob` ensures that the expected value of `a3` remains the same.

Lower keep_prob means stronger regularization.

The common values for keep_prob are:

- 0.8-0.9 for input layers.
- 0.5 for hidden layers.

Train and test time

During training, use dropout to prevent overfitting. Scale the active nodes to keep the overall output consistent.

During test time, don't use dropout at all. This is possible due to scaling the activations.

Why dropout works

Dropout is similar to L2 as it spreads weights across inputs to reduce weight norms. However, dropout is adaptive, it:

1. Randomly zeroes out neurons during training.
2. Each iteration trains a smaller sub-network.
3. Reduces reliance on specific neurons.
4. Prevents overfitting.

Typical probabilities per layer:

Layer	keep_prob	Remark
Input	0.9–1.0	Rarely drop input features
Early hidden layers	0.8–0.9	Mild regularization
Middle and deep Layers	0.5–0.7	Stronger regularization
Output	1.0	No dropout for stable output

Use dropout when:

- the model overfits.
- the model has large fully connected layers like in computer vision.

A downside of dropout is that the cost function $\mathcal{J}$ becomes non-deterministic—harder to check monotic decrease. A solution for this is to first, train with dropout off to confirm convergence.

Other regularization methods

Data augmentation

Collecting more data is expensive, apply the ff. transformations to augment data:

- Horizontal flips (not vertically)
- Random crops, rotations, and zooms
- Subtle distortions (for OCR)

Early stopping

How early stopping works:

1. During training, monitor training error (cost J) and dev set error.
2. Training error decreases monotonically as iterations increase.
3. Often, the dev set error decreases then increases due to overfitting.
4. Stop at the minimum dev set error point (early stopping).

Why early stopping works:

1. Initally, W is small due to initialization.
2. Because we stop at the point where it overfits, we keep W small like in L2 regularization.

Advantages of early stopping:

> When monitoring dev set error during one training run you pass through small, medium, and large weights w. With early stopping, you pick the iteration with lowest dev error, choosing an effective w without extensively tuning λ.

Disadvantage of early stopping:

> Early stopping combines optimization (minimizing J) and regularization (reducing variance) into one step, instead of keeping them separate. This breaks *orthogonalization* (separating optimization from variance reduction), making it harder to debug and tune hyperparameters.
>
> While computationally heavier, L2 regularization does the same thing while not breaking orthogonality.

Normalizing inputs

Normalizing corresponds to two steps:

1. Subtract the mean (move the train set until $\mu = 0$).
   - $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
   - $x := x - \mu$
2. Normalize the variances.
   - $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})^2$ (element-wise squaring)
   - $x := x / \sigma$

The simplified formula for normalizing the input: $\frac{x - \mu}{\sigma}$

Use same μ and σ from train set to normalize the test set.

What happens when the input is not normalized?

1. Features on very different scales make $J(w, b)$ elongated (e.g., $x_1 \in [1, 1000], x^2 \in [0, 1]$).
2. Due to the elongated function of $J(w, b)$, you have to use a very small learning rate α.
3. A very small learning rate α leads to a slow convergence via gradient descent.

What happens when the input is normalized?

1. The contour of the cost function J becomes more spherical.
2. A spherical countour allows gradient descent to be faster and stable.

Exploding and vanishing gradients

A problem with very deep neural networks is data vanishing and exploding gradients.

The derivatives or slopes of the deep neural networks can sometimes get:

- very, very big (explode), or
- very, very small (vanish)

Exploding gradients	• Causes excessively large parameter updates. • Makes learning unstable.	Vanishing gradients	• Causes extremly small parameter updates. • Makes learning very slow. • Might prevent convergence.

The solution to exploding and vanishing gradients is careful weight initialization using Xavier, He, and other initializations.

Weight initialization for DNN

Weight initialization applies to all layers, not only to the input layer.

For a layer l: $z^{[l]} = \sum_{i=1}^{n^{[l-1]}} w_i^{[l]} a_i^{[l-1]} \ (\text{with all } b = 0)$

The problem with normal random initialization: variance grows with the number of inputs n.

If the number of inputs $n^{[l-1]}$ is large, we want $w_i^{[l}$ to be smaller to make $z^{[l]}$ not explode or vanish.

Partial solution: initialize weights with variance inversely proportial to $n^{[l-1]}$ (or $n^{[l-1]} + n^{[l]}$ for some .

Initialization	Variance (σ^2)	Activation
LeCun	$\frac{1}{n^{[l-1]}}$	SELU
Xavier (Glorot)	$\frac{2}{n^{[l-1]} + n^{[l]}}$	tanh, sigmoid
He (Kaiming)	$\frac{2}{n^{[l-1]}}$	ReLU, Leaky ReLU

Remarks:

- These initializations are only a *partial* solution (to make DNN feasible) to the exploding and vanishing gradient problem. They only slow down or reduce the growth and shrinkage through layers—which is its goal in the first place.
- These initializations are not fixed. They are treated as starting points or good defaults for weight initialization.
- Variance σ^2 can be treated as a hyperparameter by multiplying the default value by a constant factor.
- Unlike learning rate α or regularization parameters, variance σ^2 only has *modest* effects. Therefore, it has low priority for tuning.

LeCun initialization

Also known as: Fan-in Xavier initialization
Use cases: SeLU, sigmoid

$$\sigma^2 = \frac{1}{n^{[l-1]}}$$

Rationale: Keep variance σ^2 roughly constant by scaling inversely with the number of inputs n.

Xavier initialization

Also known as: Glorot initialization
Use cases: tanh, sigmoid

$$\sigma^2 = \frac{2}{n^{[l-1]} + n^{[l]}}$$

Rationale: Balance the variance of activations and gradients by considering both number of inputs $n^{[l-1]}$ and outputs $n^{[l]}$ to prevent exploding and vanishing gradients.

He initialization

Also known as: Kaiming initialization
Use cases: ReLU and its variants

$$\sigma^2 = \frac{2}{n^{[l-1]}}$$

Rationale: ReLU zeroes out about half of its inputs. Double the variance relative to $n^{[l-1]}$ to preserve the variance to mitigate vanishing gradients.

NumPy implementations

Sampling from a Gaussian random variable and scaling by the standard deviation σ sets the desired variance.

Using NumPy:

```
# LeCun
W_lecun = np.random.randn(shape) * np.sqrt(1/n_prev)

# Xavier
W_xavier = np.random.randn(shape) * np.sqrt(2/(n_prev + n))

# He
W_he = np.random.randn(shape) * np.sqrt(2/n_prev)
```

Gradient checking

Use *gradient checking* to verify correctness when implementing backpropagation.

Gradient checking relies on the numerical approximation of gradients.

One-sided difference approximation	• $f'(\theta) = \frac{f(\theta+\epsilon)-f(\theta)}{\epsilon}$ • error: $O(\epsilon)$	Two-sided difference approximation	• $f'(\theta) = \frac{f(\theta+\epsilon)-f(\theta-\epsilon)}{2\epsilon}$ • error: $O(\epsilon^2)$

The error is smaller for the two-sided diff. approx. due to $\epsilon < 1$. Making it, the more accurate option.

Grad check procedure:

1. Take all parameters $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape them into vectors.
2. Concatenate every parameter vector into a giant vector θ.
3. The cost function $J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]})$ becomes $J(\theta)$
4. Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape them into vectors.
5. Concatenate every gradient vector into a giant vector $d\theta$.
6. For each component of θ, i: $d\theta^{[i]}_{\text{approx}} = \dfrac{J(\theta_1, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \dots, \theta_i - \epsilon, \dots)}{2\epsilon} \approx d\theta[i] = \dfrac{\partial J}{\partial \theta_i}$
7. Check $\dfrac{||d\theta_{\text{approx}} - d\theta||_2}{||d\theta_{\text{approx}}||_2 + ||d\theta||_2}$
   - $\approx 10^{-7}$: likely correct
   - $\approx 10^{-5}$: okay but double-check
   - $\approx 10^{-3}$: worry, bug in backprop

Remarks:

- Don't use during training. Use only to debug.
- If grad check doesn't match, inspect which gradient component diverges.
- If regularizing, include regularization term when calculating both $d\theta_{\text{approx}}$ and $d\theta$.
- Grad check doesn't work with dropout.
  - Implement grad check first, then re-enable dropout using `keep_prob = 1.0`.
- It is not impossible that your backprop is only correct when it is near initialization. Run grad check after at random initialization and after a few training steps to confirm correctness. This is seldomly done.
- Gradient checking is slow, run it only to make sure your code is correct. Turn it off and use backprop for the actual learning process.