Deep Learning Study Notes

Mahdi Hasan Shuvo

October 2025

A growing notebook of Improving Deep Neural Networks: Hyperparameter Tuning,Regularization and Optimization .

Contents

1. Train / Dev / Test Sets – Notes (Oct 2, 2025)

Why are Train/Dev/Test sets important in deep learning?
Answer:
Properly setting up training, development (dev), and test sets is crucial for building high-performance neural networks.

- Helps evaluate model performance correctly.
- Ensures fair comparison between models.
- Prevents overfitting to one dataset.

Example:
If you train on 100,000 images, you might split them into:

- 98,000 (Training), 1,000 (Dev), 1,000 (Test).

What is the typical process of model development?
Answer:
Applied ML is an iterative process:

1. Train an initial model.
2. Evaluate results.
3. Adjust architecture, hyperparameters, or data.
4. Repeat until performance improves.

Example:

- Start with learning rate = 0.01 → Model overfits.
- Try learning rate = 0.001 → Better generalization.

How should we choose dataset splits?
Answer:

1. Small datasets: 70% train, 15% dev, 15% test.
2. Large datasets: 98% train, 1% dev, 1% test.
3. Always ensure same distribution across sets.

Example:

- For 10,000 samples → 7,000 train, 1,500 dev, 1,500 test.

Why must dev and test sets come from the same distribution?
Answer:
Because mismatched distributions cause poor evaluation and misleading results.
Problems if mismatch occurs:

1. Poor generalization.
2. Overfitting to training only.
3. Biased evaluation metrics.
4. Increased error rates.

Example:

- Training set = high-resolution photos.
- Test set = low-resolution photos.
- Model accuracy drops significantly.

What is "Poor Generalization"?
Answer:
A model fails to perform well on unseen data, even if it performs well on training data.
Key Points:

- Usually caused by overfitting.
- Training accuracy $\gg$ Test accuracy.
- More common in complex models with limited data.

Example:

- Training Accuracy = 95%
- Test Accuracy = 60% → Model doesn't generalize.

How to improve a model's generalization?
Answer (Techniques):

1. Regularization (L1, L2).
2. Cross-validation.
3. Simplify the model.

4. Increase training data.
5. Data augmentation (rotation, flipping, scaling).
6. Early stopping.
7. Ensemble methods.
8. Feature selection.

Simple Analogy:

- Regularization = "Don't let the model memorize the book word by word."
- Data augmentation = "Practice the same topic in different ways."

What is Hyperparameter Tuning?
Answer:
The process of optimizing settings (hyperparameters) that are not learned during training but influence model performance.

Common Hyperparameters:

- Learning rate
- Batch size
- Number of layers
- Hidden units
- Activation functions
- Regularization strength

Tuning Methods:

- Grid Search: Try all combinations.
- Random Search: Try random combinations.
- Bayesian Optimization: Smart search using probability models.

Example:

- Learning rate = 0.01 → Diverges.
- Learning rate = 0.001 → Converges with 90% accuracy.

Hyperparameter tuning in deep learning refers to the process of optimizing the hyperparameters of a model to improve its performance. Hyperparameters are the settings or configurations that are not learned from the data during training but are set before the training process begins. They can significantly influence the model's ability to learn and generalize.

Key Points:

1. **Definition of Hyperparameters:**
   Hyperparameters include values such as the learning rate, batch size, number of layers, number of hidden units in each layer, activation functions, and regularization techniques.

2. **Importance:**
   The choice of hyperparameters can affect the model's convergence speed, accuracy, and overall performance. Proper tuning can lead to better results.

3. **Tuning Methods:**
   Grid Search: Testing a predefined set of hyperparameter values systematically.
   Random Search: Randomly sampling hyperparameter values from a specified range.
   Bayesian Optimization: Using probabilistic models to find the best hyperparameters by exploring the search space more intelligently.

4. **Validation Set:**
   A separate validation set is often used to evaluate the performance of different hyperparameter configurations, helping to avoid overfitting to the training data.

5. **Iterative Process:**
   Hyperparameter tuning is typically an iterative process, where you may need to try multiple combinations and refine your choices based on the results.

By effectively tuning hyperparameters, you can enhance the performance of deep learning models, leading to better predictions and more reliable outcomes.

Key Takeaways:

- Always split data properly (train/dev/test).
- Ensure same distribution across sets.
- Watch out for poor generalization (overfitting).
- Improve models with regularization, more data, and smart techniques.
- Hyperparameter tuning is iterative and essential.

2. Bias / Variance – Notes (Oct 2, 2025)

What do Bias and Variance mean in machine learning?

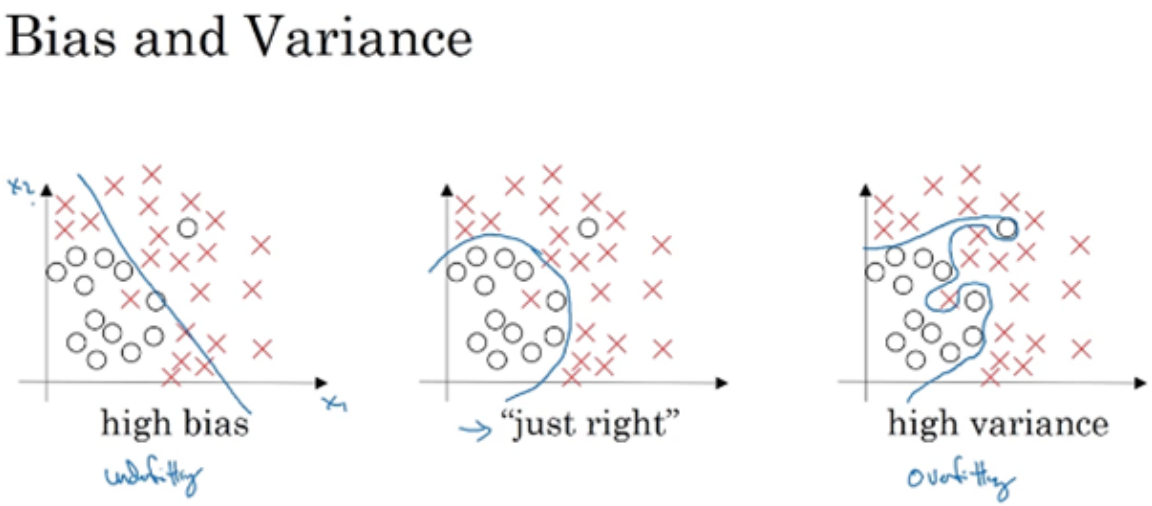

Figure: Bias and Variance

Answer:

- Bias is the error that comes from making overly simple assumptions about the data. A model with high bias does not capture important patterns and usually underfits.
- Variance is the error that comes from a model being too sensitive to training data. A model with high variance captures noise and details instead of the true patterns, leading to overfitting.

Example:

- High Bias (Underfitting): Fitting a straight line to curved data.
- High Variance (Overfitting): A deep neural network memorizing the training set but failing on new data.

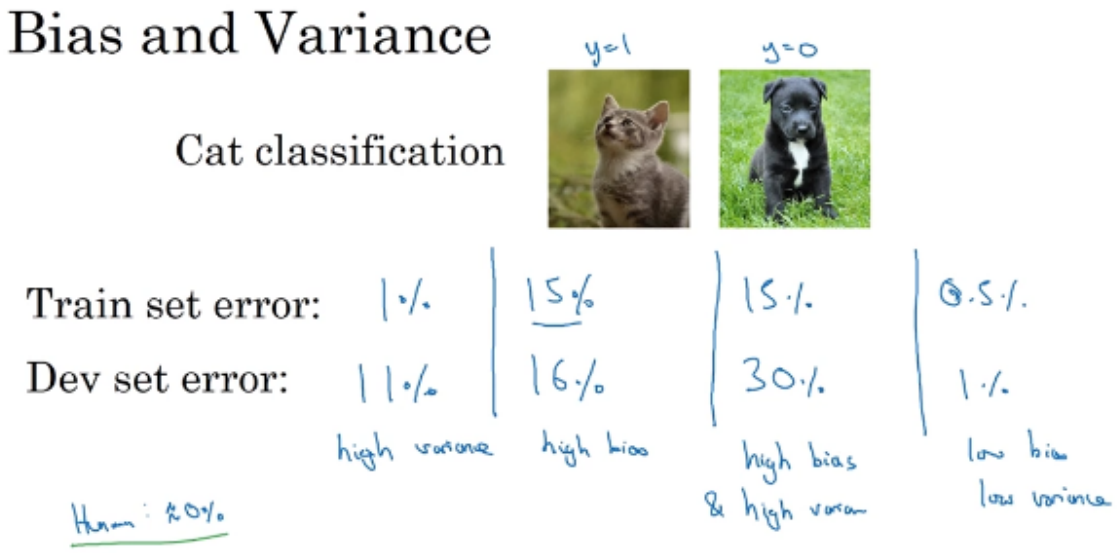

Figure: Train and Dev set error

How do bias and variance affect model performance?
Answer:

1. **High Bias (Underfitting):**
   - Model is too simple → misses important trends.
   - Performs poorly on both training and dev sets.
   - Example: Using linear regression for predicting house prices when the relationship is nonlinear.
2. **High Variance (Overfitting):**
   - Model is too complex → memorizes noise.
   - Very good on training set but poor on dev/test sets.
   - Example: A decision tree that grows too deep and perfectly fits the training set but fails on new data.
3. **Balanced Model:**
   - Finds a "sweet spot" between bias and variance.
   - Performs well on both training and dev sets.
   - Example: A well-regularized neural network that captures real trends but ignores noise.

How can we balance bias and variance in practice?
Answer (Strategies):

1. **Model Complexity:**
    - Start simple, then increase complexity if needed.
    - Linear models for simple patterns, neural networks or ensembles for complex relationships.
2. **Regularization:**
    - L2 (Ridge): Penalizes large weights → reduces overfitting.
    - L1 (Lasso): Encourages sparsity → removes unnecessary features.
3. **Cross-Validation:**
    - Use k-fold cross-validation to test model stability across different data splits.
4. **Ensemble Methods:**
    - Bagging (e.g., Random Forest): Reduces variance by averaging many models.
    - Boosting (e.g., Gradient Boosting): Reduces bias by sequentially correcting mistakes.
5. **Hyperparameter Tuning:**
    - Adjust settings like learning rate, depth of trees, or number of hidden layers to find the right balance.
6. **Feature Engineering:**
    - Remove irrelevant/noisy features to lower variance.
    - Create better, more informative features to reduce bias.
7. **Data Augmentation:**
    - Expand dataset with techniques like flipping or rotating images → improves generalization.

Why is managing bias and variance important?
Answer:
Because a model that is too simple (high bias) won't learn enough from the data, and a model that is too complex (high variance) won't generalize well. The best models balance both, achieving strong performance on unseen data.
Example:

- A balanced model might get 90% accuracy on training and 88% accuracy on dev/test.
- In contrast:

  - High Bias → 60% training / 55% test.
  - High Variance → 99% training / 65% test.

Key Takeaways:

- Bias = underfitting, Variance = overfitting.
- High bias → model too simple, misses patterns.
- High variance → model too complex, memorizes noise.
- Use regularization, ensemble methods, cross-validation, data augmentation, and feature engineering to balance the trade-off.
- A good ML model is not the simplest or the most complex but the one that generalizes well to new data.

Got it
I'll do 3 things for you:

1. Make a neat docs-style copy format of your text.
2. Summarize it into a shorter version.
3. Explain step by step with examples/figures so it's easier to understand.

Diagnosing Bias and Variance in Neural Networks
When training a neural network, you can systematically improve performance using a basic recipe:

Step 1: Check for High Bias (Underfitting)

- Look at training set error.
- If your model does not even fit the training data well → high bias.
- **Solutions:**
  - Use a bigger network (more hidden units/layers).
  - Train longer.
  - Try advanced optimizers.
  - Experiment with different architectures (sometimes works, sometimes doesn't).

Keep adjusting until your model fits the training set well.

Step 2: Check for High Variance (Overfitting)

- Look at dev/validation set error.
- If training accuracy is high but dev accuracy is low → high variance.
- **Solutions:**
  - Get more data (best solution if possible).
  - Use regularization (dropout, L2, etc.).
  - Try alternative architectures (sometimes reduces variance).

Keep adjusting until both training and dev set errors are low.

Important Notes

- The actions depend on diagnosis:
  - High bias → bigger network, longer training.
  - High variance → more data, regularization.
- **Bias-variance tradeoff:**
  - In traditional ML, reducing bias increased variance and vice versa.
  - In modern deep learning:
    - Bigger networks usually reduce bias without hurting variance (if regularized).
    - More data reduces variance without hurting bias.
- Training a bigger network rarely hurts performance (main cost = computation).

Next Step:

- Regularization is key to solving variance problems.
- It may slightly increase bias, but often not much if your network is large enough.

Summary

- Bias = error on training set (underfitting). Fix it by bigger networks, longer training, better optimizers.
- Variance = error gap between training and dev set (overfitting). Fix it by more data, regularization, or better architectures.
- Unlike older ML, deep learning lets us reduce bias and variance separately:
  - Bigger networks ↓ bias.

  - More data ↓ variance.

- Regularization is the main tool to handle variance.

Step-by-Step Explanation with Examples & Figure
Step 1: Diagnose Bias

- **Example:**
  - Training accuracy = 60%
  - Dev accuracy = 58%
  - → Model is not even fitting training data → high bias.
- **Fix:** Add layers, more neurons, train longer.
  (Figure idea: show a small network failing to capture data patterns.)

Step 2: Diagnose Variance

- **Example:**
  - Training accuracy = 95%
  - Dev accuracy = 70%
  - → Model memorized training data but fails on new data → high variance.
- **Fix:** Add more data, use dropout, L2 regularization.
  (Figure idea: show an overly complex network fitting noise in training data but failing on test data.)

Step 3: Combine Both

- **Goal = low bias + low variance.**
- **Example ideal case:**
  - Training accuracy = 95%
  - Dev accuracy = 94%
  - → Good fit, generalizes well.

  (Figure idea: show a well-regularized big network fitting data smoothly.)

In modern deep learning, we're lucky because:

- Bigger models + more data = usually better.
- The "bias-variance tradeoff" matters less than in old ML.

---

3. Regularization to Reduce Overfitting in Neural Networks

If your neural network is **overfitting** (high variance), it performs well on training data but poorly on new (dev/test) data.

Two main solutions exist:

- **Get more training data** (most effective but expensive).
- **Use regularization** (prevents overfitting by constraining parameters).

Regularization in Logistic Regression

The cost function for logistic regression is:

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

where w and b are parameters.

To regularize, add a penalty term to prevent large weights:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{j=1}^{n_x} w_j^2$$

Here:

- λ: Regularization parameter (hyperparameter)
- $\sum w_j^2$: L2 norm of the weights
- Only b is excluded from regularization (since it's a single scalar).

Types of Regularization

L2 Regularization

Uses squared weights:

$$\text{Penalty} = \frac{\lambda}{2m} \sum_{j} w_j^2$$

- Keeps all weights small (smooths the model).
- Most common; also called **Weight Decay**.

L1 Regularization

Uses absolute values of weights:

$$\text{Penalty} = \frac{\lambda}{m} \sum_j |w_j|$$

- Encourages many weights to become zero (sparse model).
- Helps compress models, but less used in deep learning.

Regularization Parameter (λ)

- Controls the strength of penalty.
- Chosen using a development set or cross-validation.
- In Python, use `lambd` instead of `lambda` (since it's reserved).

Regularization in Neural Networks

For a neural network with L layers:

$$J = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|W^{[l]}\|_F^2$$

where $\|W^{[l]}\|_F^2$ is the **Frobenius norm**:

$$\|W^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (W_{ij}^{[l]})^2$$

This prevents any weight from becoming excessively large.

Gradient Descent with Regularization

Without Regularization

$$W^{[l]} := W^{[l]} - \alpha \, dW^{[l]}$$

With L2 Regularization

$$dW^{[l]} := dW^{[l]} + \frac{\lambda}{m} W^{[l]}$$

$$W^{[l]} := W^{[l]} - \alpha \, dW^{[l]}$$

Simplified effect:

$$W^{[l]} = W^{[l]}(1 - \frac{\alpha\lambda}{m}) - \alpha(\text{gradient})$$

Each weight shrinks slightly each iteration — hence called **Weight Decay**.

Why Regularization Works

- Large weights → complex model → overfitting.
- Regularization penalizes large weights → simpler model → better generalization.

Summary Table

Concept	**Meaning / Effect**
Overfitting (High Variance)	Model memorizes training data but fails on new data
Regularization	Adds penalty to cost to keep weights small
L2 Regularization	Penalizes sum of squared weights → smoother model (used most)
L1 Regularization	Penalizes absolute weights → sparse weights (less used)
λ (lambda)	Controls regularization strength (chosen via dev set)
Frobenius Norm	Matrix version of L2 norm for neural networks
Weight Decay	Another name for L2 regularization (weights shrink slightly per update)

Step-by-Step Example

Step 1: Detect Overfitting Example:

- Training accuracy = 98%
- Dev accuracy = 78%

→ Model memorized training data → High Variance (Overfitting). Apply **regularization**.

Step 2: Add L2 Regularization (Logistic Regression Example) Original cost:

$$J = \text{average loss}$$

Regularized cost:

$$J = \text{average loss} + \frac{\lambda}{2m}\sum w_j^2$$

Example: If $w = [3, 2, 1]$ and $\lambda = 1$, then

$$\text{Penalty} = \frac{1}{2m}(3^2 + 2^2 + 1^2) = \frac{7}{2m}$$

This discourages large weights.

Step 3: Extend to Neural Networks Each layer has parameters $W^{[1]}, W^{[2]}, \ldots, W^{[L]}$. Regularization for all layers:

$$\text{Penalty} = \frac{\lambda}{2m} \sum_{l=1}^{L} \|W^{[l]}\|_F^2$$

This ensures all layers stay smooth and balanced.

Step 4: Modify Gradient Descent Before:

$$W := W - \alpha\, dW$$

After regularization:

$$W := W(1 - \frac{\alpha\lambda}{m}) - \alpha(\text{gradient})$$

This "decays" each weight slightly every iteration — L2 regularization = weight decay.

Step 5: Choose λ Wisely

- Too small → weak regularization → still overfits.
- Too large → too much regularization → underfits.

Tune λ via dev set or cross-validation.

Intuition Table

Case	Description	Result
No Regularization	Weights grow large to fit every detail	Overfits
Small λ	Slight penalty → better generalization	Balanced
Large λ	Weights forced too small → can't fit data	Underfits

Analogy

A neural network is like a **rubber sheet** fitting points:

- Without regularization → sheet bends sharply to fit every data point.
- With regularization → sheet stays smoother and fits overall shape better.

Key Takeaways

- L2 Regularization (Weight Decay) keeps weights small and prevents overfitting.
- Implemented by adding $\frac{\lambda}{2m}\|W\|_2^2$ to the cost.
- Use λ to balance underfitting and overfitting.
- Deep learning mainly uses L2 regularization, not L1.

4. Intuition Behind Why Regularization Reduces Overfitting

Why Does Regularization Help?

Regularization helps reduce **overfitting** (high variance) by penalizing large weight values in the network. The intuition is that constraining the weights makes the neural network simpler and less flexible, which prevents it from memorizing noise in the training data.

We begin with the cost function for a neural network:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Regularization adds a penalty term:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|W^{[l]}\|_F^2$$

where $\|W^{[l]}\|_F^2$ is the **Frobenius norm** (sum of squared weights).

—

Effect of Large λ

If the regularization parameter λ is large, the optimization process strongly penalizes large weights, pushing them towards zero.

- Many hidden units' weights become small, reducing their effect.
- The network effectively behaves like a smaller, simpler model.

- In extreme cases, it can behave similarly to logistic regression (linear model).

This moves the model from a **high variance (overfitting)** regime toward a **higher bias (simpler)** regime. With an appropriately chosen λ, the model achieves the "**just right**" balance.

—

Alternate Intuition with the `tanh` Activation

Consider the activation function:

$$g(z) = \tanh(z)$$

For small values of z, $\tanh(z)$ behaves almost linearly.

If λ is large, the weights W are small, and since $z = W \cdot x + b$, the values of z remain small.

Hence:

- The activations $g(z)$ stay within the linear region of tanh.
- Each layer acts approximately like a linear transformation.
- The entire deep network behaves roughly like a single linear model.

A purely linear network cannot represent complex nonlinear decision boundaries, so it becomes much less prone to overfitting.

—

Summary of the Effect

Regularization Setting	**Model Effect**
No Regularization ($\lambda = 0$)	Model highly flexible; fits noise; overfits training data
Moderate Regularization	Keeps weights small; good generalization; balanced bias/variance
Large Regularization ($\lambda \gg 0$)	Weights near zero; model becomes simpler and behaves linearly; may underfit

—

Implementation Note: Cost Function in Gradient Descent

When implementing regularization, remember that the cost function J now includes the regularization term.

If you plot the cost J during gradient descent to ensure convergence, make sure to include the **new** definition of J:

$$J = (\text{data loss}) + (\text{regularization penalty})$$

Otherwise, the plot may not decrease monotonically.

—

Key Takeaways

- Regularization penalizes large weights, making the model simpler and reducing overfitting.
- Large $\lambda \rightarrow$ smaller $W \rightarrow$ smaller $z \rightarrow$ linear activations $\rightarrow$ simpler model.
- The network transitions from complex and nonlinear to smoother and more linear.
- Always monitor the full cost J (including the penalty term) when debugging training.

—

Transition to Dropout Regularization

L2 regularization (weight decay) is one of the most common methods to control overfitting. Another powerful method used in deep learning is **Dropout Regularization**, which randomly disables neurons during training to prevent co-adaptation.

We will explore this next.

5. Dropout Regularization

Motivation

Even with L2 regularization, neural networks can still overfit complex data. **Dropout Regularization** is another powerful method that helps prevent overfitting by randomly disabling neurons during training.

The idea is simple: During training, each neuron is temporarily "dropped out" (set to zero) with a certain probability, so the network cannot rely on specific activations. This forces it to learn more robust and general features.

—

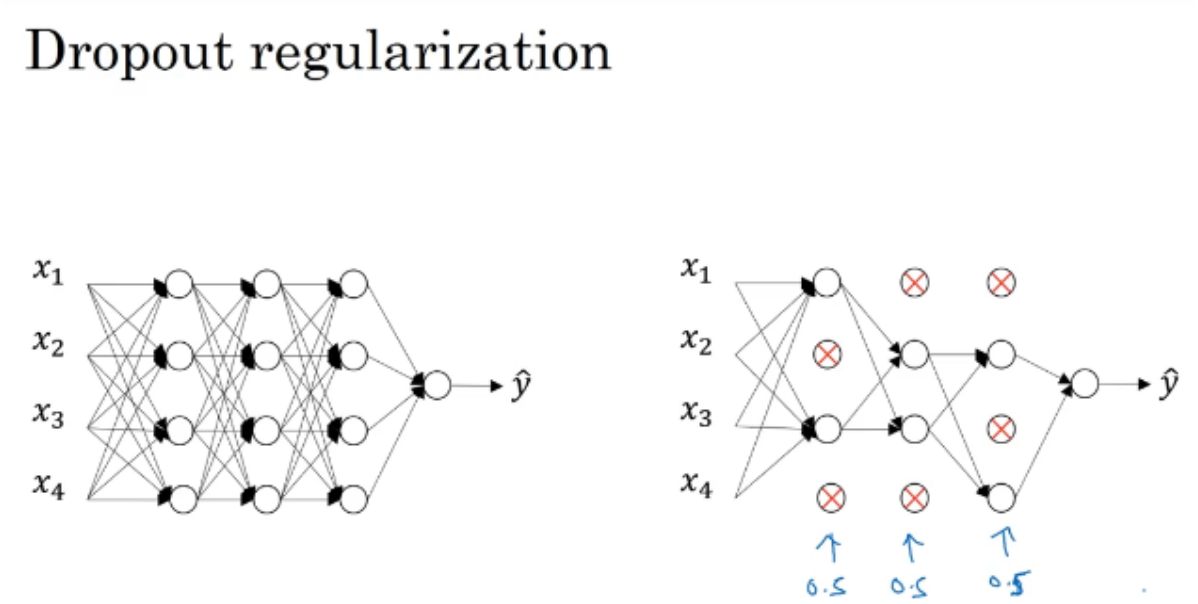

Figure: Dropout Regularization

Basic Idea

Consider a neural network that tends to overfit. Dropout works by randomly removing neurons and their connections during training.

- For each layer l, and each neuron in that layer, we "flip a coin" to decide whether to keep it active.
- Example: If the keep probability is 0.8, each neuron has an 80% chance of being kept and 20% chance of being dropped.
- The neurons that are dropped have both their activations and outgoing connections set to zero.

Each mini-batch or iteration thus trains on a slightly different, smaller network. This randomness helps prevent co-adaptation between neurons.

—

Mathematical Implementation (Inverted Dropout)

For layer $l = 3$ (as an example):

$$d^{[3]} = \text{rand}(a^{[3]}) < \text{keep\_prob}$$

where:

- $d^{[3]}$: dropout mask (same shape as $a^{[3]}$)

- `keep_prob`: probability of keeping a neuron active (e.g., 0.8)

Then:

$$a^{[3]} = a^{[3]} * d^{[3]}$$

and finally:

$$a^{[3]} = \frac{a^{[3]}}{\text{keep\_prob}}$$

This final scaling step is known as **Inverted Dropout**. It ensures that the expected value of activations remains the same during training, so that at test time, no rescaling is needed.

—

Python-Style Pseudocode

```
# Forward propagation with inverted dropout (layer l=3)
keep_prob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = a3 * d3
a3 = a3 / keep_prob
```

Explanation:

- `np.random.rand()` generates random values in [0,1].
- Each element of $d3$ is 1 (keep) with probability 0.8, 0 (drop) otherwise.
- Multiplying by $d3$ zeros out dropped neurons.
- Dividing by `keep_prob` rescales activations to maintain expected values.

—

Why Scaling Matters

Suppose a layer has 50 neurons, and keep_prob = 0.8. On average, 20% of neurons are dropped, leaving only 40 active.

If we did not scale by 1/0.8, the activations feeding into the next layer would be 20% smaller in expectation:

$$z^{[4]} = W^{[4]}a^{[3]} + b^{[4]}$$

To maintain the same expected value for $z^{[4]}$, we divide $a^{[3]}$ by 0.8.

This scaling is what makes the implementation "inverted" dropout and simplifies inference at test time.

—

Dropout During Training vs. Test Time

During Training:

- Dropout is applied — neurons are randomly dropped each iteration.
- Different dropout masks are used for each mini-batch.
- Encourages redundancy and robustness in learned features.

During Testing (Inference):

- Dropout is **not used**.
- The full network (all neurons active) is used for prediction.
- No scaling is required, because of inverted dropout.

If dropout were used at test time, it would introduce randomness and unstable predictions. The inverted dropout technique ensures the expected outputs remain consistent.

—

Key Insights and Advantages

- Dropout prevents neurons from relying too heavily on specific other neurons. Each neuron must learn features that are independently useful.
- Each mini-batch effectively trains a different "thinned" network. The final model behaves like an average of many smaller networks.
- Dropout is simple to implement and works well with other forms of regularization.

—

Summary

Concept	**Meaning / Effect**
Dropout	Randomly disables neurons during training to reduce overfitting
Keep Probability (`keep_prob`)	Fraction of neurons kept active (e.g., 0.8 means 20% dropout)
Inverted Dropout	Divide activations by `keep_prob` to keep expected output constant
Training Phase	Apply dropout mask and scaling
Testing Phase	Use full network, no dropout applied
Effect	Trains an ensemble of smaller networks, improving generalization

—

Transition to Next Topic

Now that we understand the implementation of dropout, the next step is to explore **why** dropout works so well — the deeper intuition behind its ability to reduce overfitting and variance.

6. Dropout Intuition and Implementation Details

Dropout Intuition: Dropout is a powerful regularization technique that works by randomly "knocking out" (deactivating) certain units in the neural network during training. This may seem like a strange idea, but it has a strong regularizing effect that reduces overfitting.

- On every training iteration, a random subset of hidden units is dropped out (set to zero). This is equivalent to training a smaller, temporary sub-network.
- Because the network cannot rely on any single unit or feature (as it might disappear in the next iteration), each neuron learns to spread its weights more evenly among its inputs.
- This spreading of weights effectively **reduces the squared norm of the weights**, similar to the effect of **L2 regularization**.

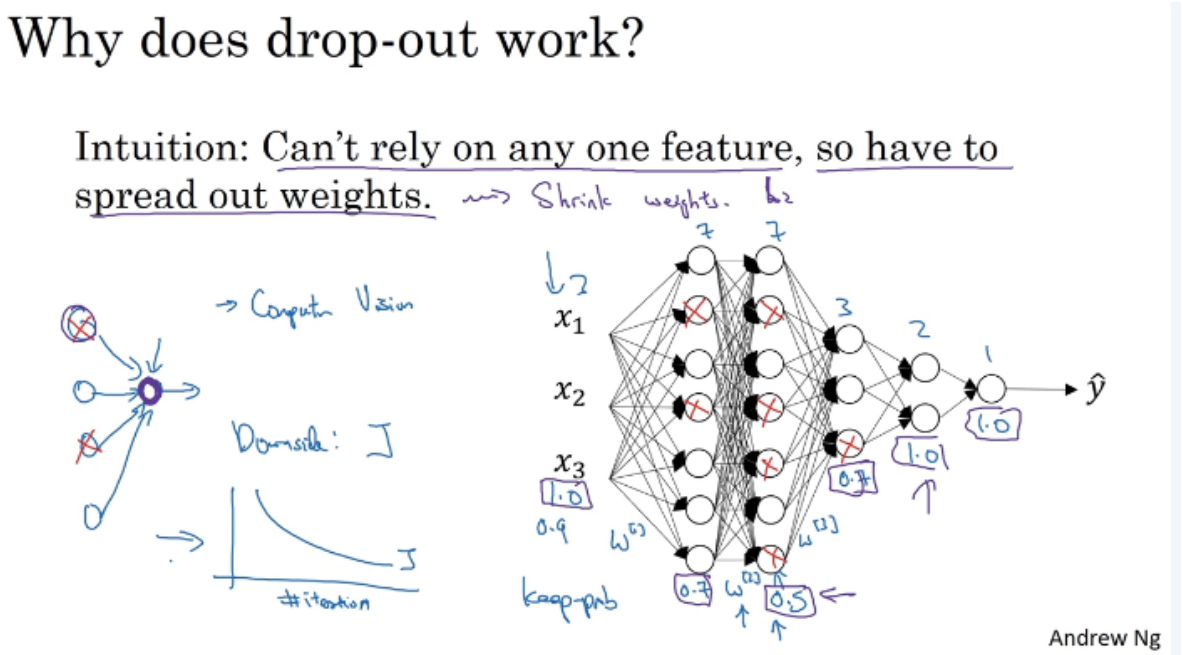

Figure: Dropout Regularization

Mathematical Effect: Dropout can be viewed as an adaptive form of L2 regularization. The L2 penalty applied to each weight depends on the magnitude of the corresponding input activation, making it layer-adaptive and data-dependent.

$$\text{Effective Regularization: } \Omega(W) \propto \sum_i \alpha_i W_i^2, \quad \text{where } \alpha_i \text{ depends on activation scale.} \tag{1}$$

Layer-wise Keep Probability: Different layers can have different dropout probabilities (*keep.prob*) depending on the risk of overfitting. Layers with more parameters often need stronger regularization (lower keep.prob).

Layer	Weight Matrix Size	Typical keep.prob	
Input Layer	$W^{[1]} \in \mathbb{R}^{7\times3}$	$0.9 - 1.0$	
Hidden Layer 1	$W^{[2]} \in \mathbb{R}^{7\times7}$	$0.5 - 0.7$	
Hidden Layer 2	$W^{[3]} \in \mathbb{R}^{3\times7}$	$0.7 - 0.9$	
Output Layer	$W^{[4]} \in \mathbb{R}^{1\times3}$	1.0 (No dropout)	

Implementation Tips:

- Dropout is most commonly used in **computer vision** because of the large number of parameters and limited data.
- In practice, apply dropout only when the model is clearly overfitting.
- **Debugging tip:** During debugging, set keep.prob = 1 (disable dropout) to check that your cost J decreases properly. Then re-enable dropout once the implementation is verified.
- The downside of dropout is that the cost function J becomes less well-defined because the model changes at every iteration. Thus, it's harder to visualize or verify gradient descent convergence directly.

Summary:

- Dropout prevents co-adaptation of neurons.
- It acts as a regularizer, similar to L2, by shrinking weights adaptively.
- Layer-specific dropout rates can control overfitting more precisely.
- Use dropout carefully — mainly when your model shows signs of overfitting.

7. Data Augmentation and Early Stopping

Overview: In addition to L2 and dropout regularization, there are other techniques for reducing overfitting in neural networks. Two commonly used methods are **data augmentation** and **early stopping**. Both aim to improve generalization, but they approach the problem differently.

Data Augmentation

Concept: When collecting additional real data is expensive or impractical, data augmentation allows you to synthetically expand your dataset by applying transformations to existing examples. This technique is widely used in computer vision tasks such as image classification.

- Augmentations can include horizontal flipping, random cropping, rotation, zooming, and translation.
- Each augmented image is treated as a new (but correlated) training example.
- Although the augmented examples do not add as much independent information as truly new data, they still improve robustness and help the model generalize better.
- For example, if an image is labeled as a "cat", then flipping or slightly rotating it should still represent a cat, teaching the model to be invariant to such transformations.

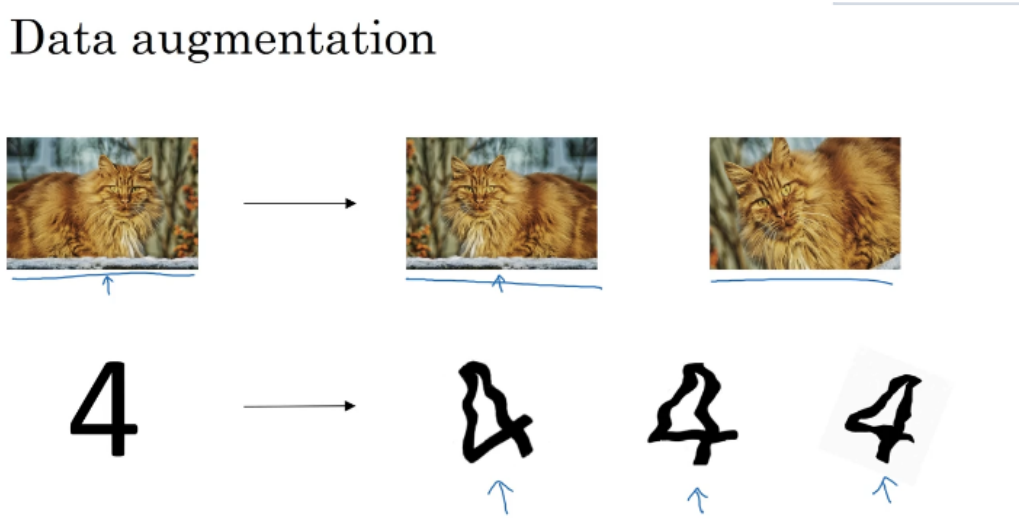

Figure 1: Example of data augmentation techniques such as flipping, cropping, and rotation applied to an image of a cat.

Applications:

- **Image classification:** Horizontal flips, crops, and small rotations.
- **OCR (Optical Character Recognition):** Random distortions, rotations, and translations applied to digits.
- **Speech recognition:** Adding background noise or changing tempo slightly.

Summary: Data augmentation is a cost-effective way to regularize neural networks by increasing the effective size of the training set without collecting new data.

Early Stopping

Concept: Early stopping is a technique where training is halted before the model overfits. During training, both training and development (validation) errors are monitored.

- The training cost J typically decreases monotonically as gradient descent progresses.
- The dev set (validation) error decreases initially but eventually begins to increase as the model starts to overfit.
- Early stopping selects the point where dev set performance is optimal — i.e., before overfitting begins.

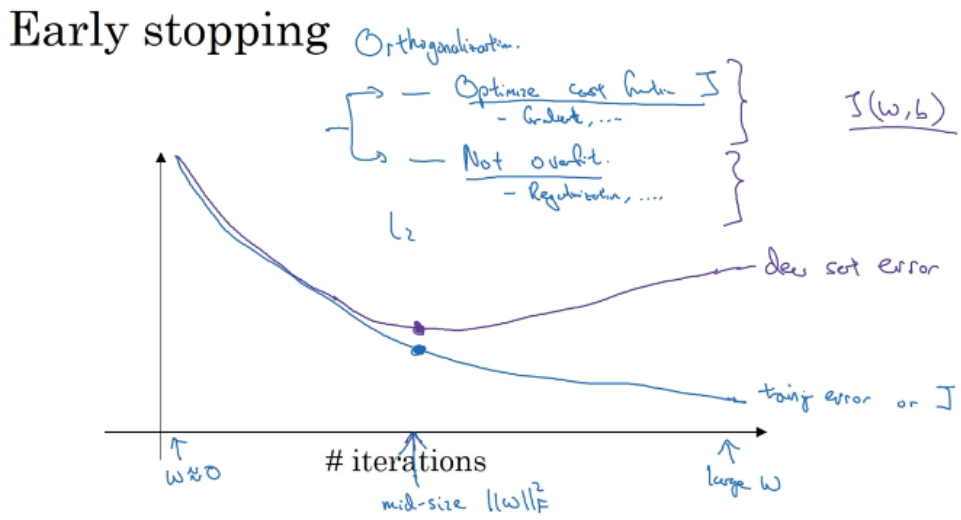

Figure 2: Illustration of early stopping: the optimal model is selected at the point where the dev set error (red) is minimal before overfitting occurs.

Why it Works: At the start of training, weights W are small (due to random initialization). As training progresses, the magnitude of weights increases. Stopping training early limits the growth of weights, leading to a model with smaller norm — similar in effect to L2 regularization.

Advantages:

- Provides a simple and efficient way to prevent overfitting without introducing an explicit regularization parameter λ.
- Reduces computational cost — one training run explores models of different complexity (small, medium, large weights).

Disadvantages:

- Couples the optimization process (minimizing J) with regularization (controlling variance), making it harder to tune them independently.

- Loses the ability to cleanly separate the tasks of optimizing cost vs. preventing overfitting — a concept known as **orthogonalization**.
- Makes debugging harder because J no longer necessarily decreases to its minimum.

Comparison with L2 Regularization: While early stopping achieves similar effects to L2 regularization, the two differ in their workflow:

- L2 regularization keeps optimization and regularization orthogonal but requires searching over many λ values.
- Early stopping simplifies computation (one training run) but mixes optimization and variance control.

Summary:

- Data augmentation and early stopping are both regularization methods that reduce variance.
- Data augmentation increases dataset diversity; early stopping limits model complexity.
- Despite coupling effects, early stopping remains a widely used and practical approach, especially when computation is limited.

8. Normalization of Training Data

When training a neural network, one of the key techniques to speed up training is to **normalize the inputs**. This ensures that each feature contributes proportionally to the learning process and avoids inefficient oscillations during optimization.

Normalize the Training Set

Let's consider a training set with two input features. The input features x are two-dimensional, represented as a scatter plot of your training data.

Normalizing your inputs involves two main steps:

1. **Zero-Center the Data:** Subtract out the mean of each feature so that the training set has zero mean:
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \quad x := x - \mu$$
This shifts the dataset so that its mean is centered around zero.

2. **Normalize the Variance:** Scale the features so that each has unit variance:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2, \quad x := \frac{x}{\sigma}$$

This ensures that features with large numerical ranges (e.g., 1–1000) do not dominate those with small ranges (e.g., 0–1).

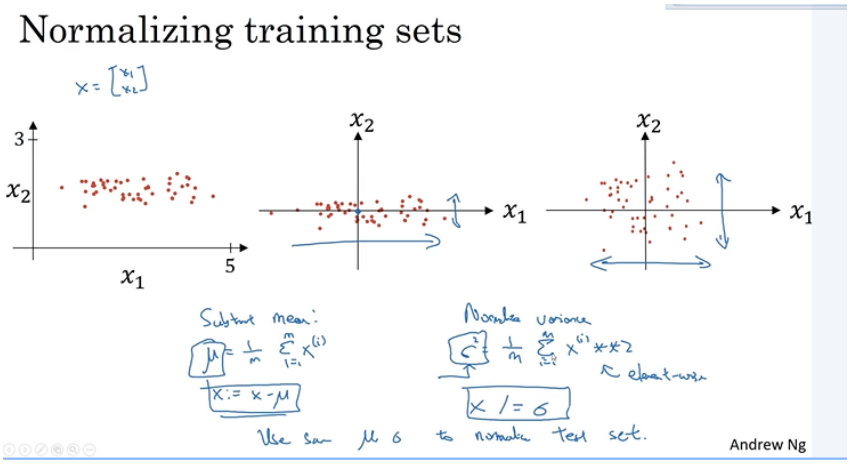

Figure 3: Effect of normalization on the training set — shifting mean to zero and scaling variance to one.

It is crucial to use the same μ and σ values computed from the training data to normalize the test data. This ensures consistent transformation and prevents data leakage.

Why Normalize the Inputs?

The motivation for normalizing features can be understood by examining the cost function landscape. Without normalization, the cost function may look like a very elongated valley, as shown conceptually below. Such imbalance arises because different features (e.g., x_1 and x_2) vary on different scales.

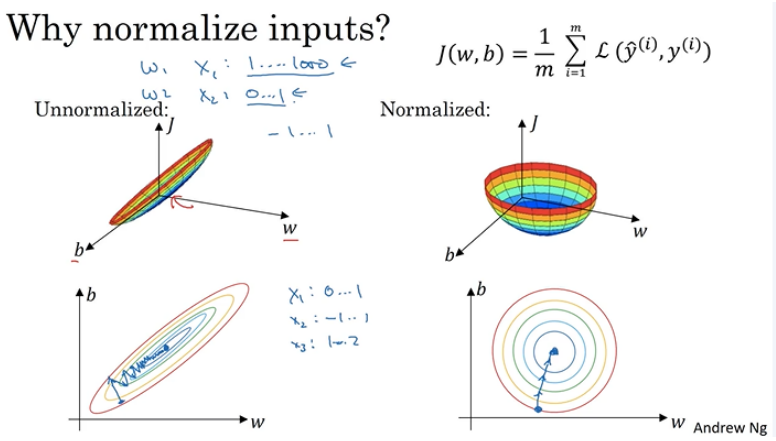

Figure 4: Impact of normalization on cost function contours — elongated vs. spherical optimization paths.

If features have dramatically different scales (for instance, x_1 ranges from 1–1000 while x_2 ranges from 0–1), the optimization landscape becomes skewed. Gradient descent will then take inefficient zig-zag steps, requiring a smaller learning rate and longer convergence time.

By contrast, when features are normalized so that all have zero mean and unit variance, the cost function becomes more **spherical and symmetric**. Gradient descent can then converge faster, taking more direct steps toward the minimum.

Key Insight: Normalization standardizes feature scales, helping gradient descent move efficiently through the cost surface. Even if your features are roughly on the same scale, applying normalization almost never hurts — it usually helps training speed and stability.

9. Vanishing and Exploding Gradients

One of the major challenges in training deep neural networks is the problem of **vanishing** and **exploding gradients**. These issues arise when derivatives (gradients) become exponentially small or large as they propagate through many layers, making training unstable or extremely slow.

Understanding the Problem

Consider a very deep neural network with parameters $W^{[1]}, W^{[2]}, \ldots, W^{[L]}$. For simplicity, assume a linear activation function $g(z) = z$ and biases $b^{[l]} = 0$.

The output of the network can then be expressed as:

$$\hat{y} = W^{[L]}W^{[L-1]} \cdots W^{[2]}W^{[1]}x$$

If each weight matrix $W^{[l]}$ is slightly larger than the identity matrix (e.g., scaled by 1.5), then we can approximate:

$$\hat{y} \approx (1.5I)^L x = 1.5^L x$$

As the number of layers L increases, the output grows exponentially, causing the **exploding gradient** problem.

Conversely, if each weight matrix is scaled by a value less than one (e.g., 0.5), then:

$$\hat{y} \approx (0.5I)^L x = 0.5^L x$$

In this case, the activations and gradients shrink exponentially, resulting in the **vanishing gradient** problem.

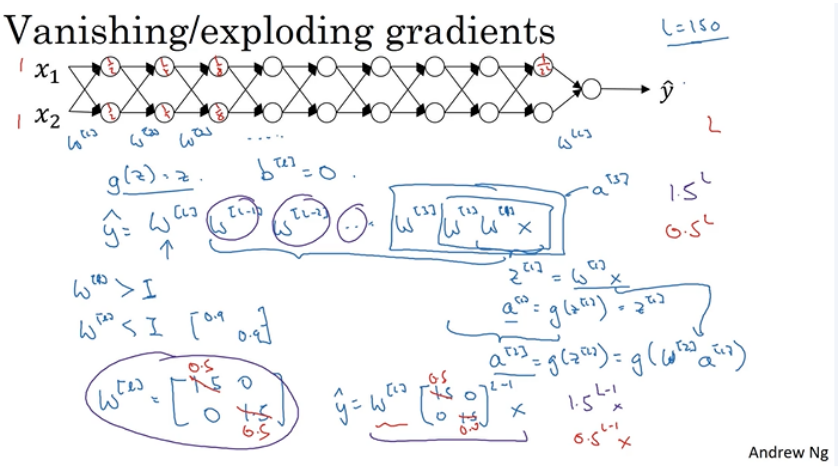

Figure 5: Illustration of vanishing and exploding gradients in deep neural networks. Gradients can exponentially decrease or increase with the number of layers.

Effect on Training

When gradients vanish, the updates to the parameters become negligibly small, causing gradient descent to take tiny steps and slow down convergence. On the other hand, exploding gradients cause updates to become excessively large, leading to instability and divergence in training.

This effect becomes especially problematic for very deep architectures, such as modern networks with over 100 layers (e.g., Microsoft's 152-layer network). In such networks, the gradients can either explode or vanish exponentially as a function of depth L.

Key Insight and Partial Solution

The takeaway is that deep networks are highly sensitive to the **initialization of weights**. If weights are slightly larger or smaller than ideal, activations and gradients can respectively explode or vanish as depth increases.

While vanishing and exploding gradients were historically a major obstacle to deep learning, a partial remedy lies in the **careful choice of weight initialization strategies**. Modern initialization techniques, such as Xavier or He initialization, help maintain stable gradient magnitudes during forward and backward propagation.

Summary:

- Deep networks can suffer from exponentially increasing or decreasing gradients.
- Exploding gradients make training unstable, while vanishing gradients slow down learning.
- Proper weight initialization significantly reduces—but does not completely eliminate—these problems.

—

10. Weight Initialization Techniques

In the previous section, we explored the problem of vanishing and exploding gradients in deep neural networks. A partial but effective solution to mitigate this issue is to use a **careful choice of random weight initialization**. Although it does not completely eliminate the problem, it significantly improves the stability of training deep models.

Single Neuron Example

To understand the intuition behind initialization, consider a single neuron with four input features x_1, x_2, x_3, x_4. The neuron computes:

$$z = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$$

and outputs $a = g(z)$. For simplicity, let $b = 0$. If the number of input features n is large, the magnitude of z will increase unless the weights w_i are small. To prevent z from becoming too large or too small, we can set the variance of w_i as:

$$Var(w_i) = \frac{1}{n}$$

This ensures that as the number of inputs grows, the contributions from each weight remain balanced and z maintains a stable scale.

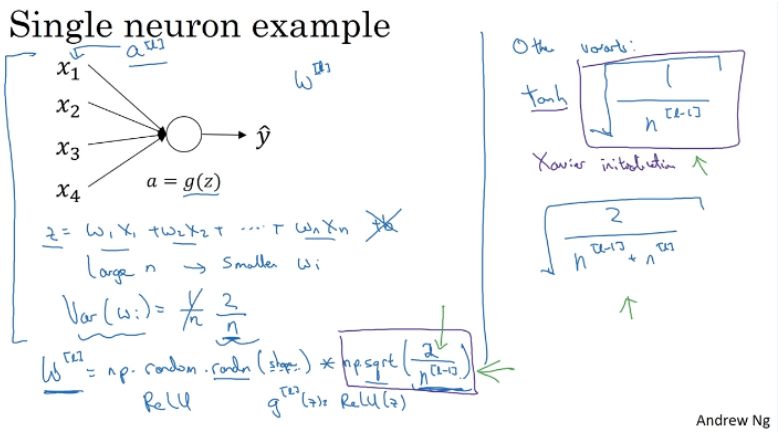

Figure 6: Example of a single neuron with multiple input features $x_1, x_2, \ldots, x_n$. Proper initialization ensures stable activations during training.

Initialization for Deep Networks

For deeper networks, each neuron in layer l receives $n^{[l-1]}$ inputs (the number of units in the previous layer). Thus, the variance of the weights for layer l is often set as:

$$Var(W^{[l]}) = \frac{1}{n^{[l-1]}}$$

This helps maintain similar magnitudes of activations across layers, reducing the risk of vanishing or exploding values.

In practice, weights are initialized from a Gaussian distribution scaled by this variance:

$$W^{[l]} \sim \mathcal{N}\left(0, \frac{1}{n^{[l-1]}}\right)$$

However, the optimal scaling factor depends on the activation function used.

He and Xavier Initialization

- **He Initialization (for ReLU and variants):**

$$Var(W^{[l]}) = \frac{2}{n^{[l-1]}}$$

  Proposed by He et al., this initialization works well with ReLU activations, ensuring that gradients maintain appropriate magnitudes during backpropagation.

- **Xavier (Glorot) Initialization (for tanh and sigmoid):**

$$Var(W^{[l]}) = \frac{1}{n^{[l-1]}}$$

  Introduced by Glorot and Bengio, this method is better suited for symmetric activation functions such as tanh or sigmoid, where preserving the variance of activations

between layers is crucial.

In both methods, weights are typically sampled from a normal distribution:

$$W^{[l]} = \text{np.random.randn(shape)} \times \sqrt{\frac{k}{n^{[l-1]}}}$$

where $k = 2$ for ReLU (He initialization) and $k = 1$ for tanh (Xavier initialization).

Effect on Training Stability

Proper initialization ensures that:

- Activations do not explode or vanish as they propagate forward.
- Gradients remain within a reasonable scale during backpropagation.
- The network converges faster and more reliably during training.

While initialization parameters can also be tuned as hyperparameters, their effect is generally moderate compared to other parameters such as learning rate or regularization strength.

Summary:

- Large W values cause exploding activations, while small W values cause vanishing activations.
- He initialization ($\frac{2}{n}$) is ideal for ReLU activations.
- Xavier initialization ($\frac{1}{n}$) works well for tanh or sigmoid activations.
- Proper weight initialization helps maintain stable activations and gradients throughout the network.

—

11. Numerical Approximation of Gradients and Gradient Checking

When implementing backpropagation, a very useful method to verify correctness is called **gradient checking**. This helps ensure that your computed gradients are accurate, as errors in backpropagation equations can be hard to detect by inspection alone.

Numerical Gradient Approximation

Consider a simple function $f(\theta) = \theta^3$. To approximate its gradient numerically, rather than shifting θ only to the right, we compute function values at both $\theta + \epsilon$ and $\theta - \epsilon$.

- Choose a small value of ϵ, e.g., $\epsilon = 0.01$.
- Compute the function values:

$$f(\theta + \epsilon), \quad f(\theta - \epsilon)$$

- The **two-sided difference approximation** of the derivative is:

$$g(\theta) \approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

This method uses both sides of the curve and results in a much more accurate approximation of the derivative than a one-sided difference.

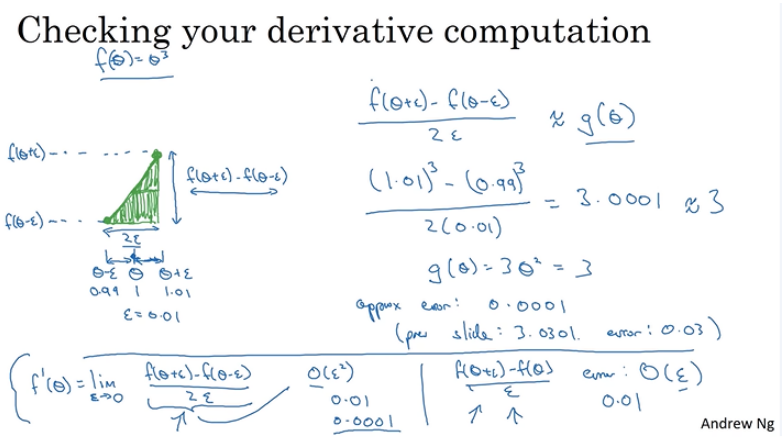

Figure 7: Two-sided difference for numerical gradient checking.

Example Calculation

For $\theta = 1$ and $\epsilon = 0.01$:

$$f(\theta + \epsilon) = (1.01)^3, \quad f(\theta - \epsilon) = (0.99)^3$$

Hence,

$$g(\theta) \approx \frac{(1.01)^3 - (0.99)^3}{2 \times 0.01} = 3.0001$$

The exact derivative is $g(\theta) = 3\theta^2 = 3$, and the numerical approximation closely matches it, with an error of only 0.0001.

Accuracy of the Approximation

The accuracy of this two-sided difference stems from its error term being on the order of $\mathcal{O}(\epsilon^2)$, while a one-sided difference has an error of $\mathcal{O}(\epsilon)$. For small ϵ, ϵ^2 is much smaller than ϵ, which makes the two-sided method significantly more accurate.

$$\text{Error (two-sided)} \propto \epsilon^2 \quad \text{vs.} \quad \text{Error (one-sided)} \propto \epsilon$$

Connection to Gradient Checking in Backpropagation

In practice, gradient checking applies this two-sided difference formula to verify the correctness of your backpropagation implementation:

$$\frac{\partial J}{\partial \theta_i} \approx \frac{J(\theta_i + \epsilon) - J(\theta_i - \epsilon)}{2\epsilon}$$

By comparing this numerical gradient to your analytical backpropagation result, you can identify bugs or inconsistencies in your implementation.

Key takeaway: The two-sided difference method provides a highly accurate numerical estimate of gradients and is essential for verifying your backpropagation algorithm.

12. Gradient Checking in Neural Networks

Gradient checking is a powerful debugging technique that helps verify the correctness of backpropagation implementations. It compares analytically computed gradients (from backpropagation) with numerically approximated gradients (using the two-sided difference method).

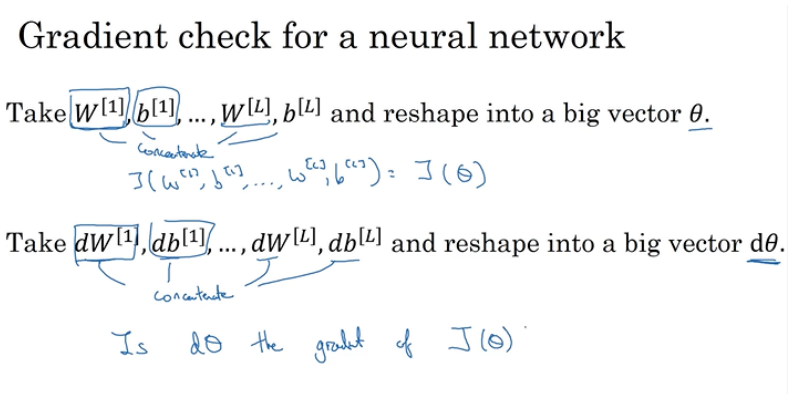

Figure 8: Neural network parameters used for gradient checking.

Reshaping Parameters into a Single Vector

For a neural network with parameters $\{W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}\}$:

- Flatten each weight matrix $W^{[l]}$ into a vector.
- Flatten each bias vector $b^{[l]}$.
- Concatenate all flattened vectors into a single long vector:

$$\theta = [W^{[1]}_{flatten}, b^{[1]}, W^{[2]}_{flatten}, b^{[2]}, \ldots, W^{[L]}_{flatten}, b^{[L]}]$$

This results in a giant parameter vector θ. Similarly, reshape and concatenate the computed derivatives $\{dW^{[l]}, db^{[l]}\}$ into another vector of the same size:

$$d\theta = [dW^{[1]}_{flatten}, db^{[1]}, \ldots, dW^{[L]}_{flatten}, db^{[L]}]$$

Computing Numerical Gradients

Treating the cost function J as a function of θ, we can compute numerical approximations for each component θ_i:

$$d\theta^{(i)}_{\text{approx}} = \frac{J(\theta_1, \ldots, \theta_i + \epsilon, \ldots) - J(\theta_1, \ldots, \theta_i - \epsilon, \ldots)}{2\epsilon}$$

where ϵ is a very small number (commonly 10^{-7}).

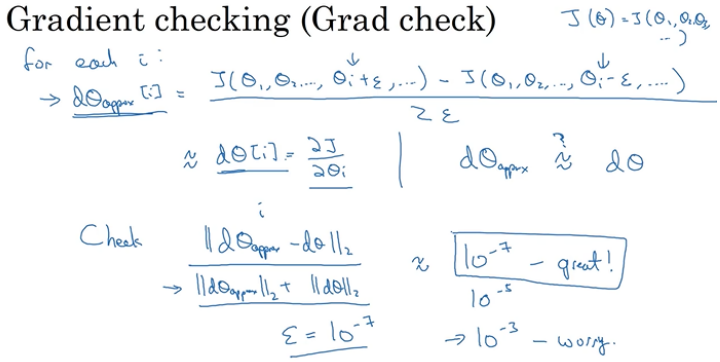

Figure 9: Two-sided numerical gradient checking for neural network parameters.

Comparing Analytical and Numerical Gradients

After computing both $d\theta$ (from backpropagation) and $d\theta_{\text{approx}}$ (from numerical estimation), compare them using the following ratio:

$$\text{difference} = \frac{||d\theta_{\text{approx}} - d\theta||_2}{||d\theta_{\text{approx}}||_2 + ||d\theta||_2}$$

Interpretation of results:

- If difference $< 10^{-7}$ — Excellent, implementation is likely correct.
- If difference $\approx 10^{-5}$ — Possibly fine, but inspect components carefully.

- If difference $> 10^{-3}$ — Likely a bug in the backpropagation implementation.

Debugging and Best Practices

If the difference is large:

- Inspect individual components of $d\theta_{\text{approx}}$ and $d\theta$ to find where discrepancies occur.
- Verify layer-by-layer gradient computations.
- Check dimensions and reshaping steps for accuracy.

Once debugging leads to a small numerical difference, you can be confident that your backpropagation is correctly implemented.

Key takeaway: Gradient checking provides a reliable way to detect subtle bugs in neural network training code by comparing analytical and numerical gradients before deploying large-scale models.

13. Practical Tips for Implementing Gradient Checking

In the previous section, we discussed the concept and mathematics behind **gradient checking**. This section provides practical guidelines on how to correctly implement it and avoid common pitfalls when debugging neural network training.

1. Use Gradient Checking Only for Debugging

Gradient checking is computationally expensive since it requires evaluating the cost function $J(\theta)$ multiple times for each parameter θ_i. Therefore, it should be used only for debugging purposes—not during actual training.

- During training, compute gradients using backpropagation only.
- When debugging, compare backpropagation gradients $d\theta$ with numerically estimated gradients $d\theta_{\text{approx}}$.
- Once verified, disable gradient checking to improve runtime efficiency.

2. Investigate Failing Gradient Checks

If the gradient check fails (i.e., the difference between $d\theta_{\text{approx}}$ and $d\theta$ is large), inspect individual components to identify potential sources of error.

- Examine which indices i produce the largest discrepancies.
- If discrepancies correspond mostly to bias parameters $db^{[l]}$, focus debugging efforts there.
- If differences are concentrated in weight parameters $dW^{[l]}$, recheck their derivative computations for the affected layer(s).

This process helps narrow down the bug to specific layers or parameters within the network.

3. Include Regularization Terms in the Cost Function

When using regularization, ensure that it is included consistently in both the cost function and its gradient:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||W^{[l]}||_F^2$$

Note: The analytical gradient $d\theta$ must also include the derivative of the regularization term. Forgetting this is a common source of gradient check failures.

4. Gradient Checking and Dropout

Gradient checking does not work properly when **dropout** is active, since dropout introduces randomness by dropping different sets of neurons during each iteration.

- Temporarily disable dropout during gradient checking by setting `keep_prob = 1.0`.
- Verify gradient correctness without dropout.
- After validation, re-enable dropout for training.

While technically possible to fix the random dropout mask and perform gradient checking, this is rarely done in practice.

5. Run Gradient Check at Multiple Stages

Although rare, it is possible that your backpropagation implementation behaves correctly near random initialization (when W and b are small), but becomes inaccurate after training for several iterations.

- Run gradient check at initialization (small weights).
- Train the network for a few epochs to let parameters deviate from initial values.
- Run gradient check again to ensure gradient correctness at larger parameter magnitudes.

6. Summary of Key Recommendations

- Use gradient checking only for debugging, not during training.
- Compare $d\theta_{\text{approx}}$ and $d\theta$ carefully.
- Always include regularization in both cost and gradient.
- Disable dropout during gradient checking.
- Optionally re-check after partial training.

Conclusion: Gradient checking is a valuable debugging tool for neural network implementations. When used correctly, it ensures the correctness of your backpropagation algorithm and prevents subtle computational errors before deploying or scaling your models.

14. Optimization Algorithms

Overview

In this section, you learn about **optimization algorithms** that enable faster training of neural networks. Deep learning is highly empirical and iterative, often requiring training multiple models to find one that performs best. Efficient optimization algorithms greatly enhance productivity, especially when working with large datasets.

Mini-Batch Gradient Descent

Traditional **batch gradient descent** processes the entire training set before taking a single update step. This can be slow when m (the number of training examples) is very large. To address this, the dataset is divided into smaller subsets called **mini-batches**.

- Each mini-batch (denoted as $X^{\{t\}}$, $Y^{\{t\}}$) contains a subset of training examples.
- For example, if the full dataset has 5 million examples and each mini-batch contains 1,000 examples, there will be 5,000 mini-batches total.
- Notation summary:
  - $X^{(i)}$ — the i^{th} training example
  - $X^{[l]}$ — the activations or parameters at layer l
  - $X^{\{t\}}$ — the t^{th} mini-batch
- Dimensions:

$$X^{\{t\}} \in \mathbb{R}^{n_x \times 1000}, \quad Y^{\{t\}} \in \mathbb{R}^{1 \times 1000}$$

Illustrations

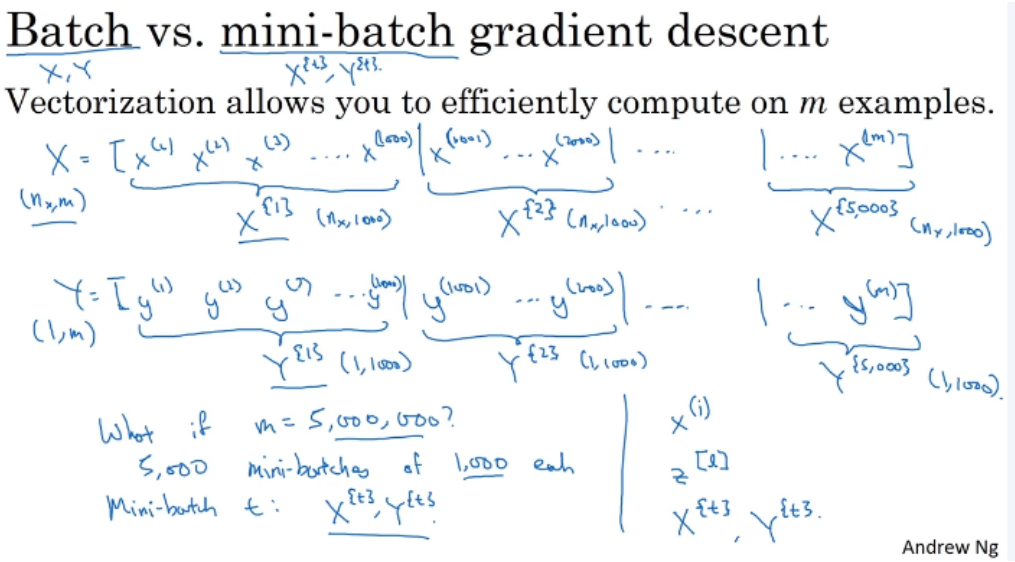

Figure 10: Comparison between Batch and Mini-Batch Gradient Descent.

Mini-Batch Gradient Descent Process

For each mini-batch:

1. Perform **forward propagation** on $X^{\{t\}}$ to compute activations:

$$Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}, \quad A^{[1]} = g^{[1]}(Z^{[1]}), \ldots, A^{[L]} = g^{[L]}(Z^{[L]})$$

2. Compute the cost function for the mini-batch:

$$J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{1000}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l}\|W^{[l]}\|_F^2$$

3. Perform **backpropagation** to compute gradients $\frac{\partial J^{\{t\}}}{\partial W^{[l]}}$, $\frac{\partial J^{\{t\}}}{\partial b^{[l]}}$.

4. Update parameters:

$$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

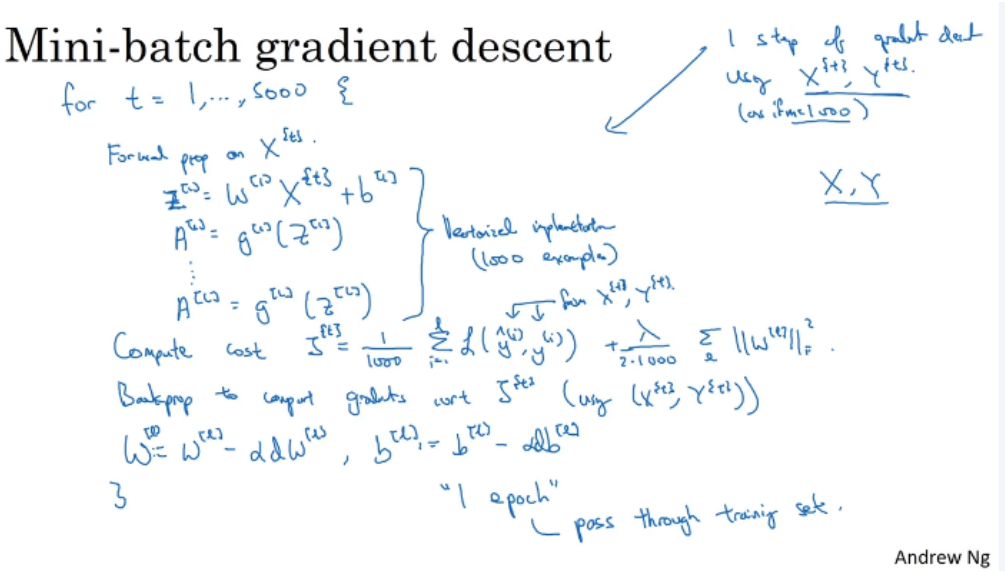

Figure 11: Working process of Mini-Batch Gradient Descent.

Epoch and Convergence

- A single pass through the entire training set (all mini-batches) is called an **epoch**.
- In batch gradient descent, one epoch equals one gradient descent step.
- In mini-batch gradient descent, one epoch equals multiple gradient descent steps (e.g., 5,000 steps if there are 5,000 mini-batches).
- Multiple epochs are run until convergence or near-convergence.

Advantages of Mini-Batch Gradient Descent

- Faster convergence compared to full-batch gradient descent.
- More efficient use of vectorization and parallel computation.
- Provides a balance between **stochastic gradient descent (SGD)** and full-batch methods.

Key Takeaways

- Mini-batch gradient descent splits large datasets into manageable subsets to speed up training.
- Each mini-batch allows an update step, making convergence faster.
- The choice of mini-batch size affects the efficiency and stability of learning.
- Widely used in deep learning frameworks for large-scale training.

15. Understanding Mini-Batch Size and Training Behavior

Overview

In the previous section, you learned how mini-batch gradient descent can help you start training even before processing your entire dataset. In this section, we explore how the cost function behaves during training and how to choose an appropriate mini-batch size.

Behavior of the Cost Function

- In **batch gradient descent**, the cost function J decreases smoothly on every iteration since it is computed over the entire dataset.
- In **mini-batch gradient descent**, the cost function $J^{\{t\}}$ is computed using only a subset of the data $(X^{\{t\}}, Y^{\{t\}})$.
- As a result, the cost may fluctuate (or oscillate) slightly across iterations due to variability between mini-batches. However, it should generally trend downward.
- Small oscillations are normal; large or increasing oscillations may indicate a learning rate that is too high.

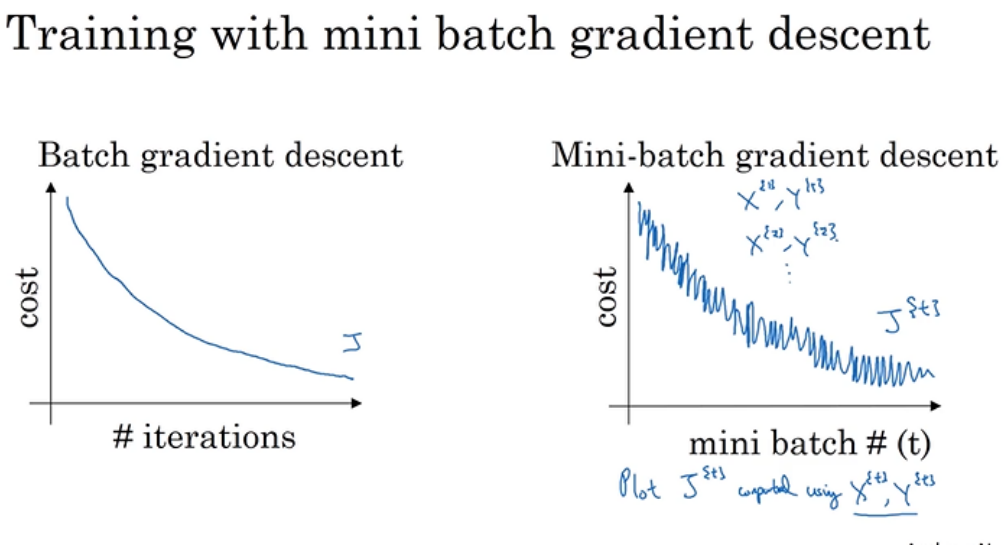

Figure 12: Training progress comparison between Batch and Mini-Batch Gradient Descent. Notice the smoother cost curve in batch gradient descent and the noisier, but faster-progressing curve in mini-batch gradient descent.

Extremes of Mini-Batch Size

- Let m be the total number of training examples.

- If **mini-batch size** = m, you get **batch gradient descent**:

  One iteration = one full pass over all training examples.

- If **mini-batch size** = 1, you get **stochastic gradient descent (SGD)**:

  One iteration = one training example per update step.

- In practice, using a size between 1 and m achieves the best trade-off between speed and stability.

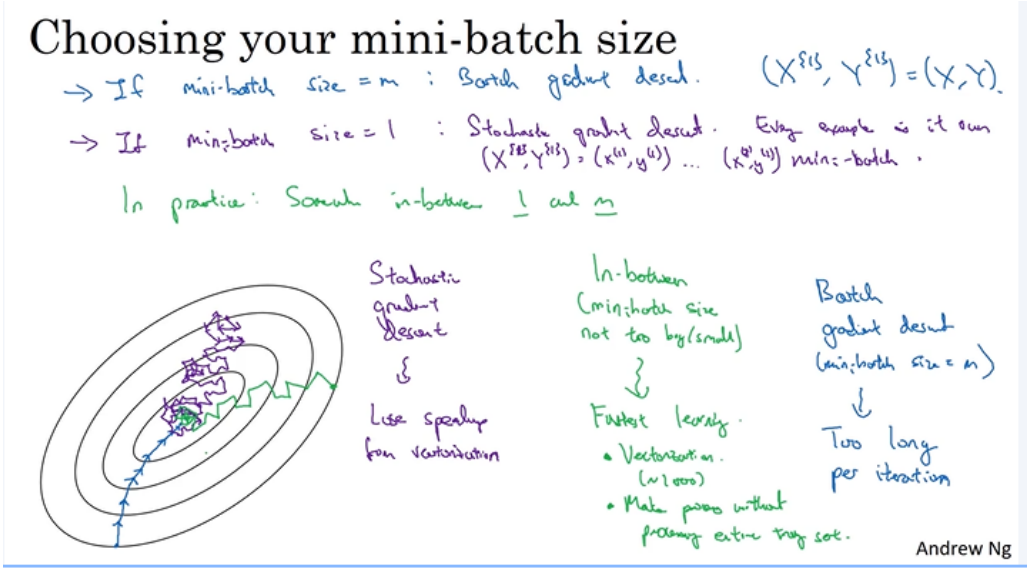

Figure 13: Visualization of Batch, Stochastic, and Mini-Batch Gradient Descent paths on a cost function contour. Batch descent follows a smooth path; stochastic descent oscillates widely; mini-batch strikes a balance.

Trade-Off Analysis

- **Batch Gradient Descent:**
  - Pros: Stable and smooth convergence.
  - Cons: Very slow for large datasets; requires computing gradients over all data.
- **Stochastic Gradient Descent:**
  - Pros: Updates occur frequently; fast initial progress.
  - Cons: Highly noisy; cannot fully leverage vectorization; rarely converges precisely.
- **Mini-Batch Gradient Descent:**

- Pros: Combines the advantages of both; supports parallelization; offers smoother convergence.
- Cons: Slightly noisier than batch descent, but significantly faster overall.

Choosing Mini-Batch Size

- Choose a mini-batch size that fits efficiently in your CPU/GPU memory.
- Common choices: 32, 64, 128, 256, or 512 examples per batch.
- Too large → slower updates, less stochasticity.
  Too small → unstable updates, poor vectorization.
- Empirical testing often required to find the optimal size.

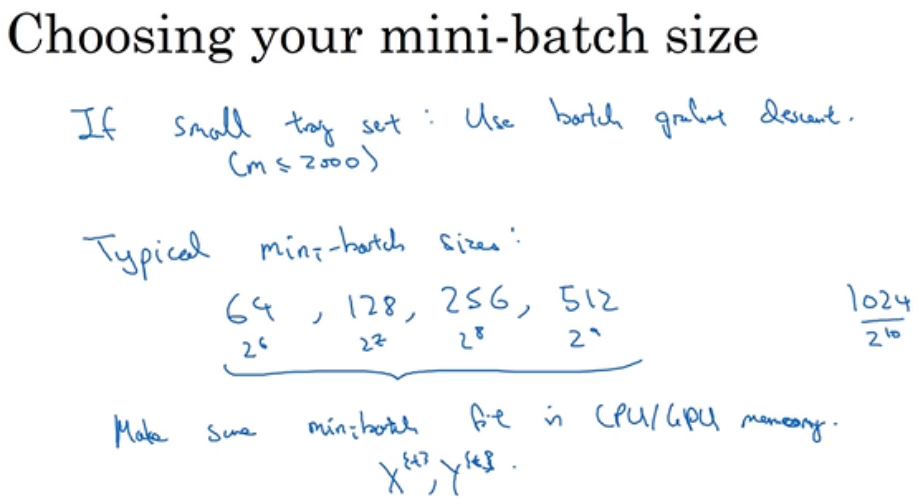

Figure 14: Guidelines for choosing mini-batch size — balancing computational efficiency, convergence stability, and memory constraints.

Key Takeaways

- Mini-batch gradient descent introduces controlled noise, improving generalization and training speed.
- Batch size directly affects computational efficiency and convergence stability.
- Optimal batch size depends on dataset size, hardware, and learning rate.

16. Exponential Weighted Averages

In this section, we introduce the concept of **Exponentially Weighted Averages (EWA)**, also known as **Exponentially Weighted Moving Averages (EWMA)** in statistics. This concept serves as the foundation for faster optimization algorithms such as Momentum and Adam.

Understanding the Concept

Consider daily temperature data from London over one year. The raw data appears noisy, but we can compute a smooth trend using an exponentially weighted average.

The idea is to maintain a moving average where recent data points have higher weight, while older ones exponentially decay in influence. The formula is:

$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t$$

where:

- v_t: exponentially weighted average up to day t
- θ_t: observed temperature on day t
- β: smoothing parameter $(0 < \beta < 1)$

Interpretation:

- Larger $\beta \rightarrow$ smoother curve, but slower to respond to changes.
- Smaller $\beta \rightarrow$ more responsive curve, but noisier.

Example Visualization

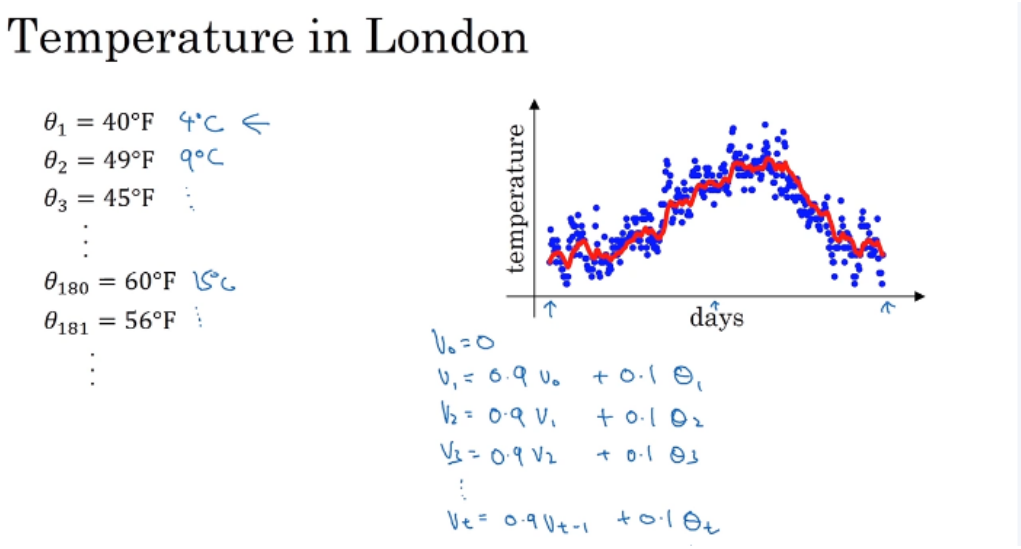

Figure 15: Raw daily temperature (blue) and exponentially weighted average (red).

If $\beta = 0.9$, the average roughly covers the last $\frac{1}{1-\beta} = 10$ days. Setting $\beta = 0.98$ averages over about 50 days, producing a much smoother curve but introducing **latency**—the average lags behind actual changes.

Effect of Different Beta Values

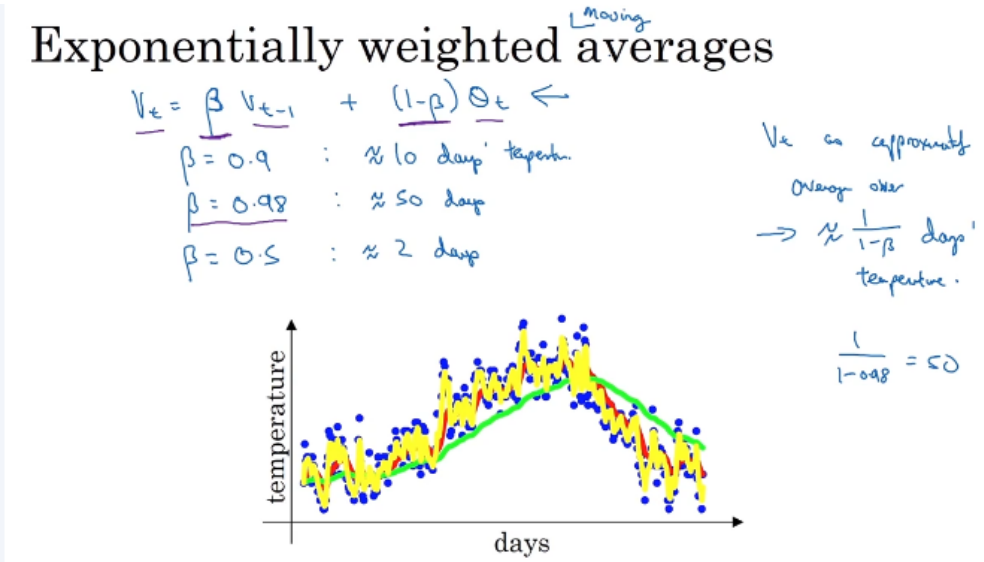

Figure 16: Effect of different β values: smaller β reacts faster but is noisier, while larger β is smoother but lags more.

Observation:

- $\beta = 0.5$: very responsive but highly noisy (averages over 2 days)
- $\beta = 0.9$: moderately smooth and balanced (averages over 10 days)
- $\beta = 0.98$: very smooth but with noticeable delay (averages over 50 days)

Summary

Exponentially weighted averages provide a balance between responsiveness and smoothness in noisy data. By adjusting β, one can control the trade-off:

$$\text{Higher } \beta \Rightarrow \text{Smoother but slower adaptation.}$$

This principle will later be used in optimization algorithms to smooth gradient updates and accelerate convergence.

17. Exponential Weighted Averages (Detailed Intuition and Implementation)

Mathematical Intuition

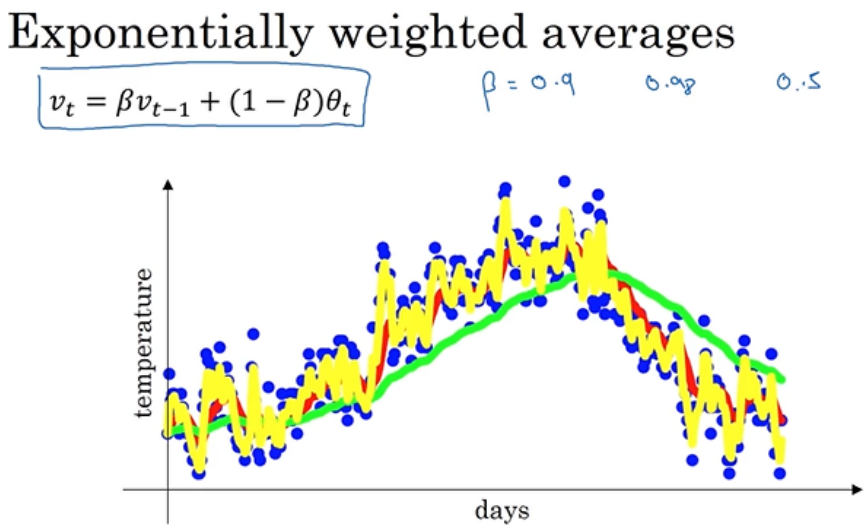

Figure 17: Mathematical derivation and intuition of exponentially weighted averages.

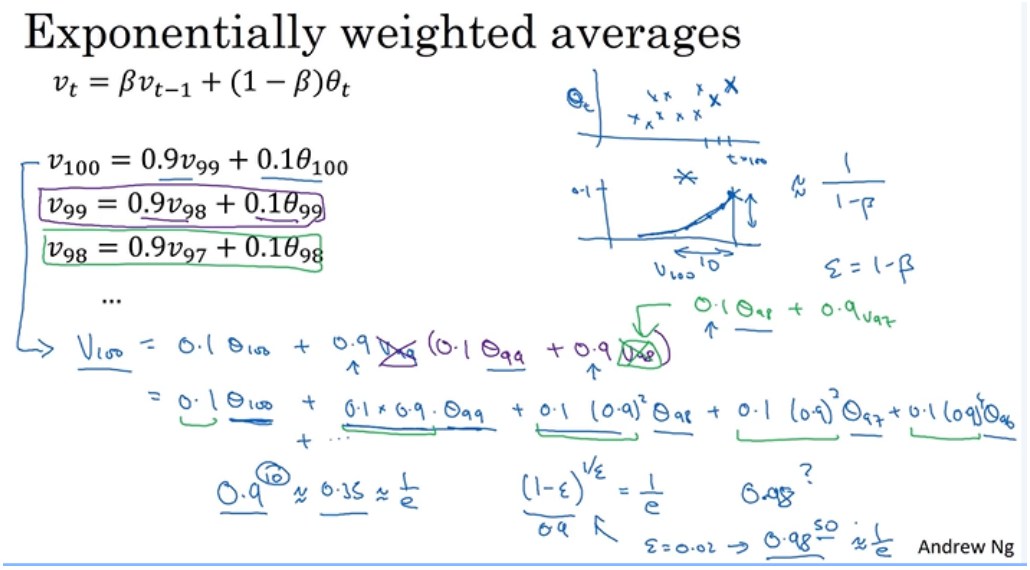

Figure 18: Effect of different β values on exponentially weighted averages.

Key Notes:

- The exponentially weighted average is a core concept used in optimization algorithms such as Momentum, RMSprop, and Adam.
- The governing formula:

  $$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

  where β controls how much past observations are remembered.
- Expanding the formula recursively:

  $$v_{100} = 0.1\theta_{100} + 0.1(0.9)\theta_{99} + 0.1(0.9)^2\theta_{98} + \dots$$

  shows that recent observations have exponentially higher weights.
- Effective averaging window $\approx \frac{1}{1-\beta}$:
  - $\beta = 0.9 \Rightarrow$ roughly 10 days.
  - $\beta = 0.98 \Rightarrow$ roughly 50 days.
- Approximation relationship:

  $$(1-\epsilon)^{1/\epsilon} \approx \frac{1}{e}, \quad \text{where } \epsilon = 1-\beta$$

- Larger β values smooth the averages but introduce lag; smaller β values react faster but add noise.

Implementing Exponentially Weighted Averages

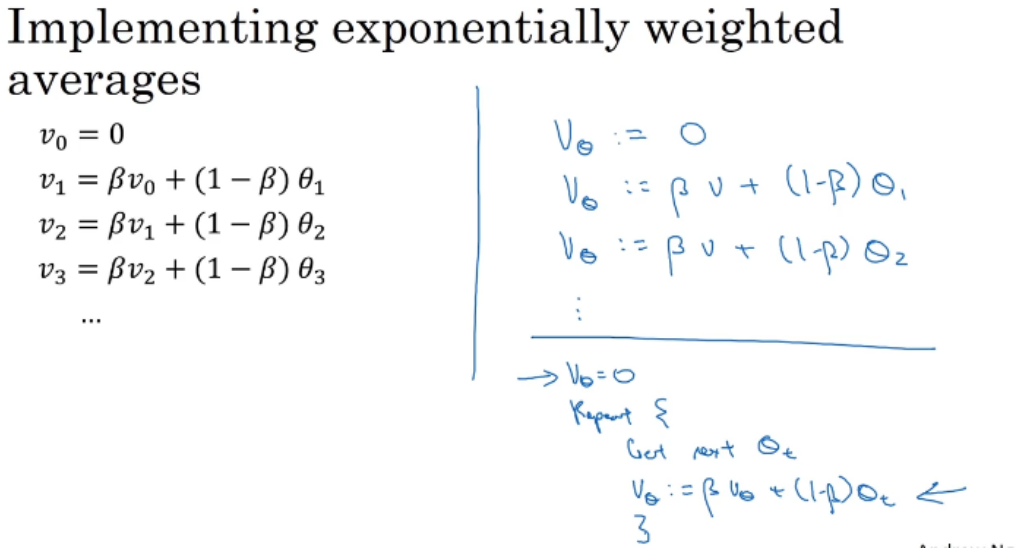

Figure 19: Implementation logic of exponentially weighted averages.

Implementation Steps:

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= \beta v_0 + (1-\beta)\theta_1 \\
v_2 &= \beta v_1 + (1-\beta)\theta_2 \\
v_3 &= \beta v_2 + (1-\beta)\theta_3 \\
&\vdots
\end{aligned}
$$

In code-like pseudocode:

$$v_\theta := 0$$

Repeat for each time step t:

$$v_\theta := \beta v_\theta + (1-\beta)\theta_t$$

Explanation:

- You start with $v_0 = 0$, then iteratively update v using the formula.
- This approach stores only a single value of v in memory at any time — no need to store historical data.
- It is therefore extremely memory-efficient and computationally lightweight.

- Though not the most accurate way to compute a true average (like a fixed-size moving window), it performs well in large-scale learning settings where efficiency matters.
- For instance, storing the last 10 or 50 days' data to compute a traditional average is more accurate but computationally and memory expensive.
- This single-line iterative update makes the exponentially weighted average ideal for machine learning optimization algorithms.

Summary: Exponentially weighted averages provide a simple, efficient, and memory-friendly way to smooth data over time. Their implementation requires just one line of update code and a single stored variable, making them highly suitable for large-scale systems. The next step introduces the concept of *bias correction*, which adjusts for initialization effects when $v_0 = 0$.

18. Bias Correction in Exponentially Weighted Averages

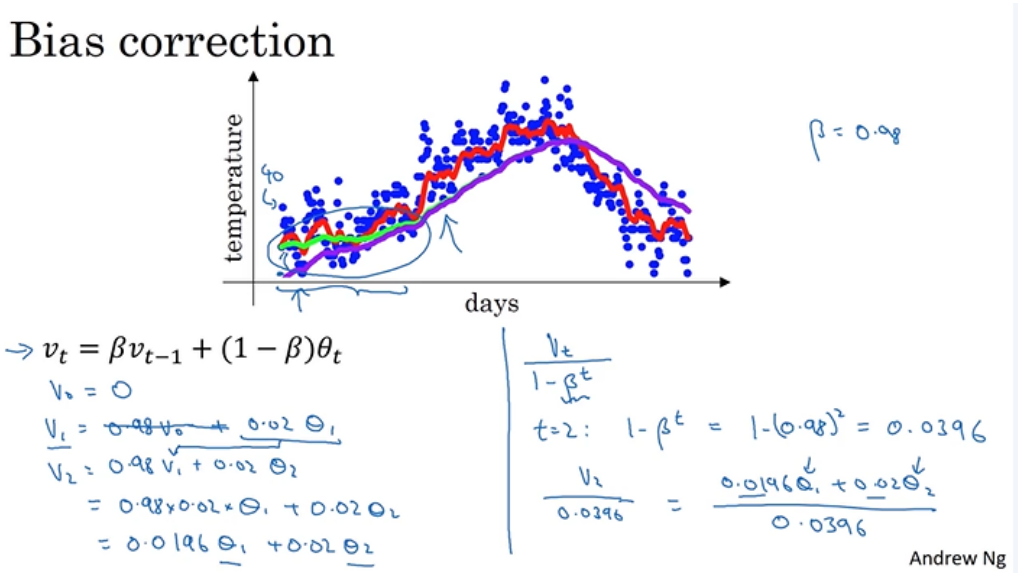

Figure 20: Effect of bias correction on exponentially weighted averages.

Understanding the Problem

When implementing exponentially weighted averages, we initialize the estimate with:

$$v_0 = 0$$

Then, for each time step:

$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t$$

However, because the initial value $v_0 = 0$, the early estimates are significantly biased toward zero. This bias occurs because in the early iterations, not enough past data has accumulated to reflect the true average. The effect is especially noticeable when β is large (e.g., 0.98 or 0.99), where the algorithm relies heavily on past data.

Illustration of the Bias Effect

Consider the first few updates:

$$\begin{aligned} v_1 &= (1 - \beta)\theta_1, \\ v_2 &= \beta v_1 + (1 - \beta)\theta_2 = \beta(1 - \beta)\theta_1 + (1 - \beta)\theta_2. \end{aligned}$$

If $\beta = 0.98$, then:

$$v_1 = 0.02\theta_1 \quad \text{and} \quad v_2 = 0.0196\theta_1 + 0.02\theta_2$$

This shows that v_1 and v_2 are much smaller than either θ_1 or θ_2, producing underestimated values (purple curve in the figure).

The Bias Correction Solution

To address this, we introduce a corrected estimate:

$$\hat{v}_t = \frac{v_t}{1 - \beta^t}$$

Here: - v_t is the raw exponentially weighted average. - $\hat{v}_t$ is the bias-corrected version. - The denominator $1 - \beta^t$ adjusts for the initialization bias.

Example: When $t = 2$ and $\beta = 0.98$:

$$1 - \beta^t = 1 - 0.98^2 = 0.0396$$

Then:

$$\hat{v}_2 = \frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

This scaling corrects for the initialization lag and gives a much better estimate of the true moving average early in the process.

Interpretation and Impact

- As t increases, $\beta^t \to 0$, so $1 - \beta^t \approx 1$. Therefore, the correction becomes negligible after a few iterations.

- In early iterations, bias correction helps the moving average quickly approach the true value — the green curve in the figure.
- Without correction, the initial phase underestimates (purple curve).

Why Bias Correction Matters in Machine Learning

- In optimization algorithms like **Adam** and **RMSProp**, bias correction ensures early estimates of moving averages (for gradients or squared gradients) are not underestimated.
- It provides a more stable and faster convergence, especially at the beginning of training.
- However, in simple applications such as smoothing time-series data, some practitioners skip bias correction, accepting minor initial bias.

Summary: Bias correction compensates for the initialization effect caused by $v_0 = 0$. By dividing v_t by $1 - \beta^t$, we normalize early estimates, obtaining more accurate averages during the "warm-up" phase. While the effect diminishes as t grows, it can significantly improve the stability of early iterations in optimization algorithms. This correction is an essential component of advanced optimizers such as **Adam**, which builds upon these exponentially weighted averages for gradient estimation.

19. Gradient Descent with Momentum

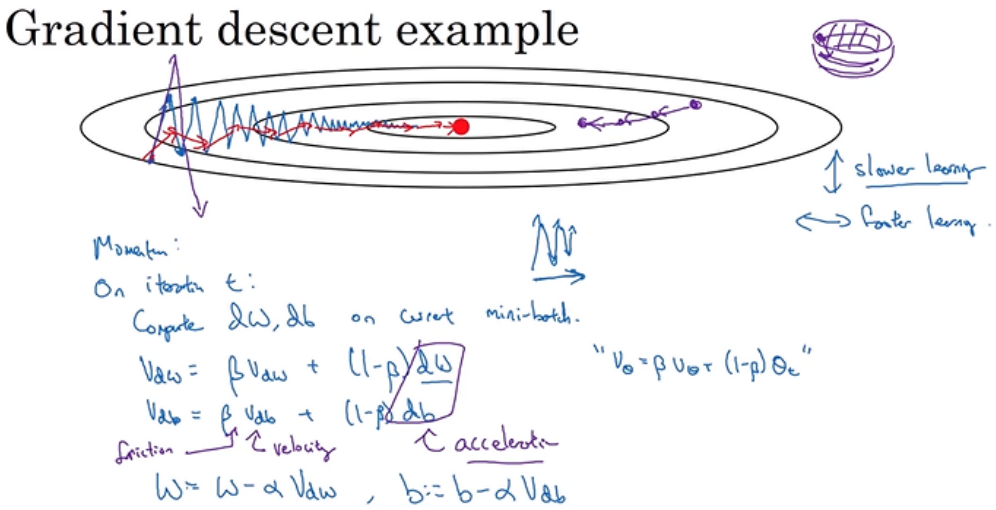

Figure 21: Gradient descent with momentum: momentum smooths oscillations and accelerates progress toward the minimum.

High-level intuition — what momentum buys you

Standard gradient descent updates parameters using only the current gradient. In ill-conditioned problems (narrow valleys / elongated contours) this causes the optimizer to oscillate in the high-curvature direction and take small progress in the low-curvature direction. Momentum fixes this by accumulating an exponentially weighted average of past gradients (a "velocity") and using that velocity to update parameters.

Two complementary intuitions help:

- **Signal averaging:** In directions where gradients consistently point the same way (the shallow direction toward the minimum), the running average reinforces those updates and the algorithm moves faster. In directions where gradients alternate sign (the steep direction), positive and negative values cancel out in the average, reducing oscillation.
- **Physics analogy:** Think of parameters as a ball rolling down the loss surface. Gradients act like forces (acceleration). The velocity (momentum) accumulates these accelerations and carries the ball through flat regions while friction (the momentum decay) prevents runaway speed.

The result: fewer zig-zag steps, larger effective step sizes along consistent directions, and faster convergence in practice — especially on deep networks and mini-batch training.

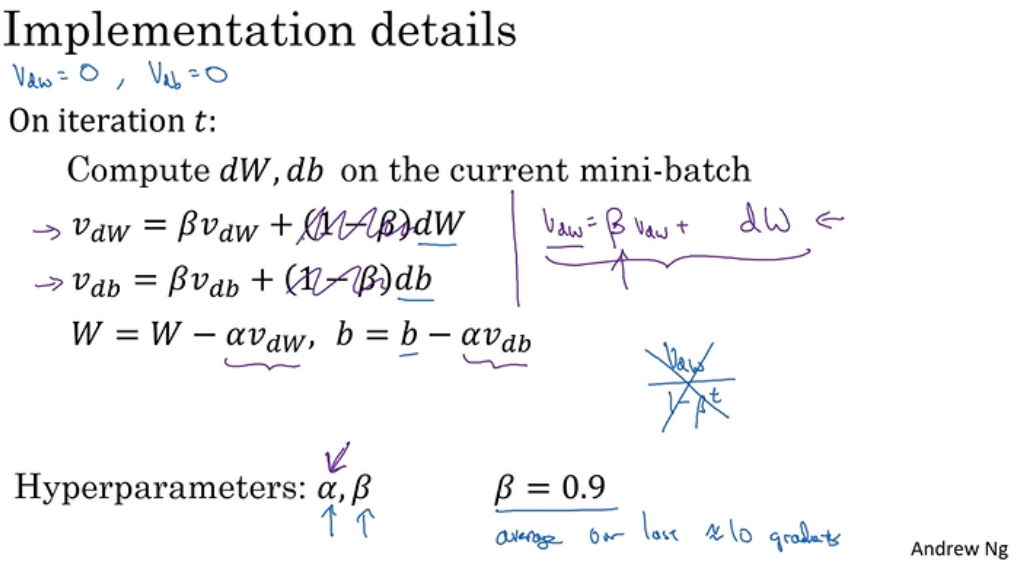

Figure 22: Implementation schematic: computing moving averages of gradients and using them to update parameters.

Algorithmic implementation and equations

Below is the standard implementation used in practice (per-parameter/matrix form). Initialize velocity terms to zero with the same shapes as weights and biases:

$$v_{dW} \leftarrow 0, \qquad v_{db} \leftarrow 0$$

On each iteration t (e.g., per mini-batch) compute the gradients dW, db using back-prop on that mini-batch, then update the velocity and parameters as:

$$\begin{aligned} v_{dW} &:= \beta\, v_{dW} \;+\; (1-\beta)\, dW, \\ v_{db} &:= \beta\, v_{db} \;+\; (1-\beta)\, db, \\ W &:= W - \alpha\, v_{dW}, \\ b &:= b - \alpha\, v_{db}. \end{aligned}$$

Notes on the above form:

- α is the learning rate, β the momentum coefficient (typical default $\beta = 0.9$).
- The $(1-\beta)$ multiplier makes v the *discounted average* of recent gradients (so magnitudes are comparable to raw gradients). Some variants omit $(1-\beta)$ and use $v \leftarrow \beta v + dW$; that works too but changes the scale of v and therefore requires retuning α.
- Initialization $v = 0$ introduces a small early bias in v. In momentum it is usually negligible after a few iterations, so bias correction is rarely used. (Contrast with Adam, where bias correction is applied.)

Practical tips and hyperparameter guidance

- **Default hyperparameters:** $\beta = 0.9$ works well in most cases. Start with the learning rate you used for plain SGD and try slightly larger values — momentum often allows larger effective step sizes.
- **Where it helps most:** Problems with elongated loss surfaces (strongly different curvature along axes), deep nets, and mini-batch training.
- **Monitoring:** Expect training loss vs iterations to be smoother than raw SGD, and training to reach lower loss in fewer epochs. If you see divergence, reduce α first, then consider lowering β.
- **Implementation detail:** Keep v_{dW} and v_{db} the same shape as the corresponding parameters (vectorize across layers). Use the same update repeatedly for each mini-batch.

- **Alternative variant:** Some libraries implement Nesterov momentum (look-ahead gradient) which often gives slightly better performance; conceptually similar but uses a gradient evaluated at a look-ahead position.

Summary

Momentum replaces noisy per-step updates with a smoothed velocity that accelerates in consistent directions and damps oscillations in inconsistent ones. It is simple to implement (one line to update v, one line to update parameters) and yields faster, more stable convergence in nearly all practical deep learning settings.

20. RMSprop (Root Mean Square Propagation)

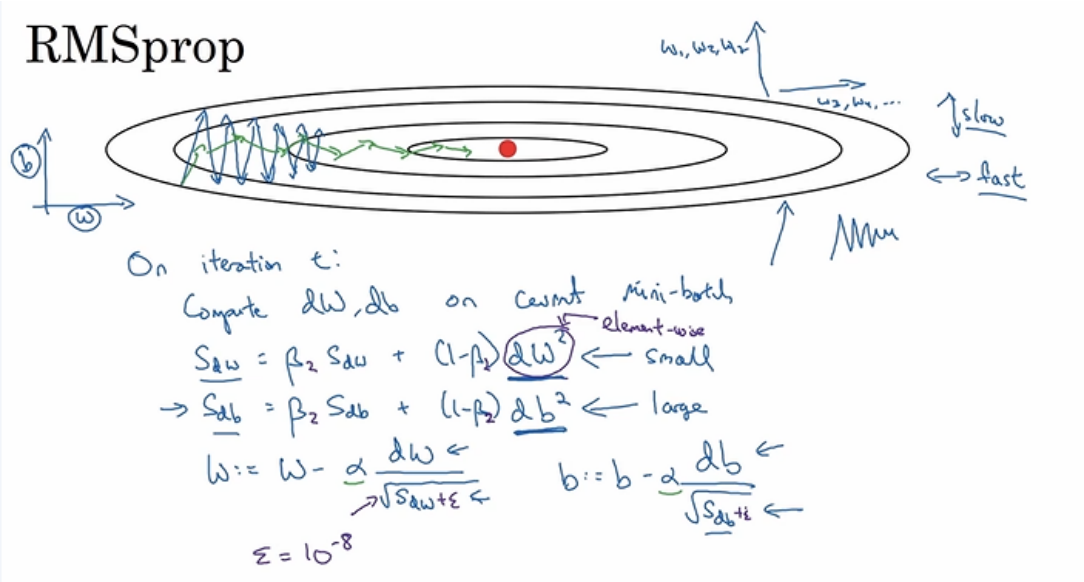

Figure 23: Illustration of RMSprop: damping oscillations in the vertical direction while accelerating horizontal convergence.

RMSprop (**Root Mean Square Propagation**) is an optimization algorithm designed to adapt the learning rate for each parameter individually. It helps speed up convergence and reduce oscillations, especially when the gradients differ in scale across dimensions.

Concept and Intuition

Traditional gradient descent may suffer from large oscillations in directions with steep curvature (e.g., the vertical axis b) while progressing slowly in flatter directions (e.g., the horizontal axis w). RMSprop addresses this by maintaining an **exponentially weighted average of squared gradients**. This effectively scales the learning rate differently across each parameter dimension:

- Parameters with large gradients (steep slopes) get smaller updates.
- Parameters with small gradients (flat regions) get larger updates.

Thus, RMSprop automatically dampens oscillations in steep directions and accelerates movement along flat regions.

Algorithm

On each iteration t:

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2$$
$$S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$$

where:

- S_{dW}, S_{db} are exponentially weighted averages of the squared gradients,
- β_2 is a decay rate (commonly $\beta_2 = 0.9$),
- Squaring operations (dW^2, db^2) are performed **element-wise**.

The parameter updates are computed as:

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}} + \varepsilon}$$
$$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

where ε is a small constant (e.g., 10^{-8}) added for numerical stability to prevent division by zero.

Intuitive Explanation

- When the vertical direction has large gradients (db large), S_{db} becomes large, leading to smaller update steps — reducing oscillations.
- When the horizontal direction has smaller gradients (dW small), S_{dW} remains small, allowing larger updates and faster convergence.

This dynamic adjustment allows RMSprop to:

- Dampen oscillations in directions of high curvature,
- Maintain faster learning in smoother directions,
- Enable higher learning rates (α) without divergence.

Remarks

- RMSprop effectively normalizes gradient magnitudes by their root mean square.
- It was introduced by Geoffrey Hinton in a Coursera lecture (not a paper), and later became a standard adaptive optimizer.
- RMSprop is often combined with momentum (forming the Adam optimizer).

21. Adam Optimization Algorithm (Adaptive Moment Estimation)

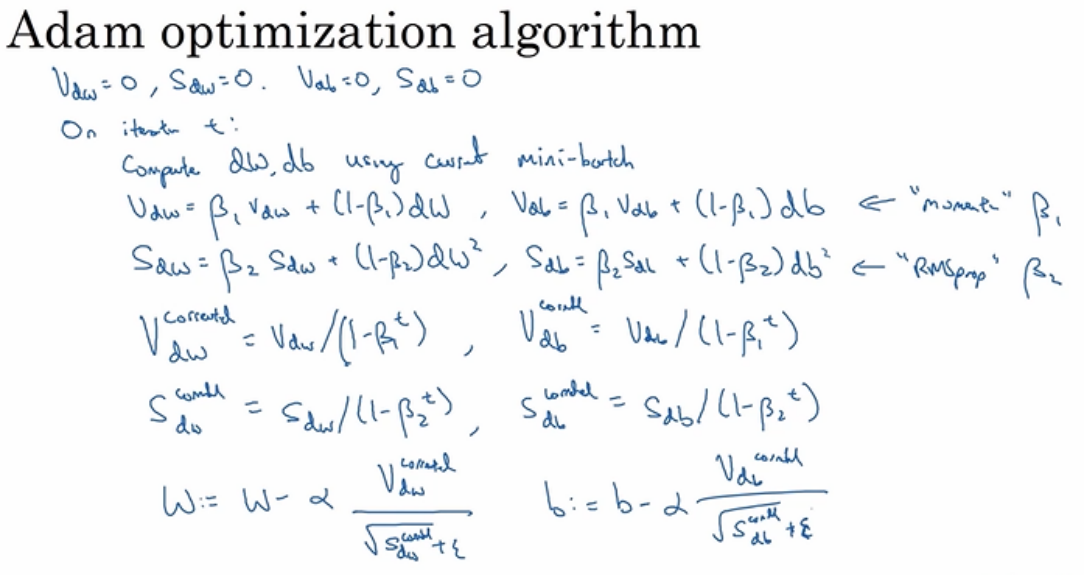

Figure 24: Adam Optimization combines the benefits of Momentum and RMSprop for faster, more stable convergence.

Adam (**Adaptive Moment Estimation**) is one of the most widely used optimization algorithms in deep learning. It integrates the ideas of both **Momentum** and **RMSprop** to provide efficient, adaptive learning rates for each parameter. The algorithm has demonstrated robust performance across a wide range of neural network architectures, making it a go-to choice for practitioners.

Concept and Motivation

Earlier, Momentum helped accelerate learning in consistent gradient directions, while RMSprop adjusted learning rates based on the magnitude of recent gradients. Adam combines both:

- The **Momentum component** (first moment) smooths out the direction of gradients using an exponentially weighted moving average.
- The **RMSprop component** (second moment) scales learning by the inverse root of an exponentially weighted average of squared gradients.

This combination enables Adam to converge faster, reduce oscillations, and adapt effectively to complex loss landscapes.

Algorithm Steps

Initialize:

$$V_{dW} = 0, \quad S_{dW} = 0, \quad V_{db} = 0, \quad S_{db} = 0$$

For each iteration t:

1. **Compute gradients:**

$$dW,\ db \quad \text{on the current mini-batch.}$$

2. **Momentum (first moment estimate):**

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$$

$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$$

3. **RMSprop-like update (second moment estimate):**

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2$$

$$S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$$

4. **Bias Correction:**

$$V_{dW}^{\text{corr}} = \frac{V_{dW}}{1 - \beta_1^t}, \quad S_{dW}^{\text{corr}} = \frac{S_{dW}}{1 - \beta_2^t}$$

$$V_{db}^{\text{corr}} = \frac{V_{db}}{1 - \beta_1^t}, \quad S_{db}^{\text{corr}} = \frac{S_{db}}{1 - \beta_2^t}$$

5. **Parameter Update:**

$$W := W - \alpha \frac{V_{dW}^{\text{corr}}}{\sqrt{S_{dW}^{\text{corr}}} + \varepsilon}$$

$$b := b - \alpha \frac{V_{db}^{\text{corr}}}{\sqrt{S_{db}^{\text{corr}}} + \varepsilon}$$

Intuitive Understanding

Adam's adaptive mechanism can be interpreted as follows:

- The **momentum term** accumulates velocity in directions with consistent gradients, accelerating movement toward minima.
- The **RMSprop term** normalizes gradient magnitudes, preventing large, unstable steps.
- **Bias correction** ensures that early iterations (when moving averages are biased toward zero) are adjusted properly.

This allows Adam to maintain stability while adapting the learning rate dynamically for each parameter dimension.

Hyperparameter Selection

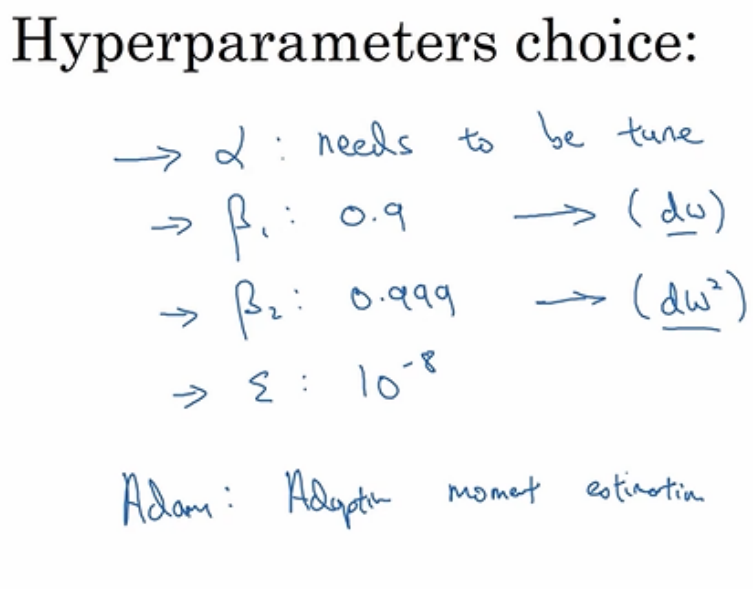

Figure 25: Common default hyperparameters for Adam and their effects.

The Adam algorithm introduces the following hyperparameters:

- Learning rate (α): typically tuned manually; common defaults are $\alpha = 10^{-3}$.
- $\beta_1 = 0.9$: controls the exponential decay rate for the first moment (momentum term).
- $\beta_2 = 0.999$: controls the exponential decay rate for the second moment (RMSprop term).
- $\varepsilon = 10^{-8}$: ensures numerical stability; rarely tuned in practice.

In most implementations, β_1, β_2, and ε are left at default values, while α is adjusted experimentally for the best results.

Remarks

- **Adam = Momentum + RMSprop:** combines acceleration and adaptive scaling.
- Performs well across CNNs, RNNs, Transformers, and general-purpose deep networks.
- The name *Adam* stands for **Adaptive Moment Estimation**:
  - β_1: first moment (mean of gradients),
  - β_2: second moment (mean of squared gradients).
- Proven to be one of the most robust and general-purpose optimizers in deep learning.

22. Learning Rate Decay

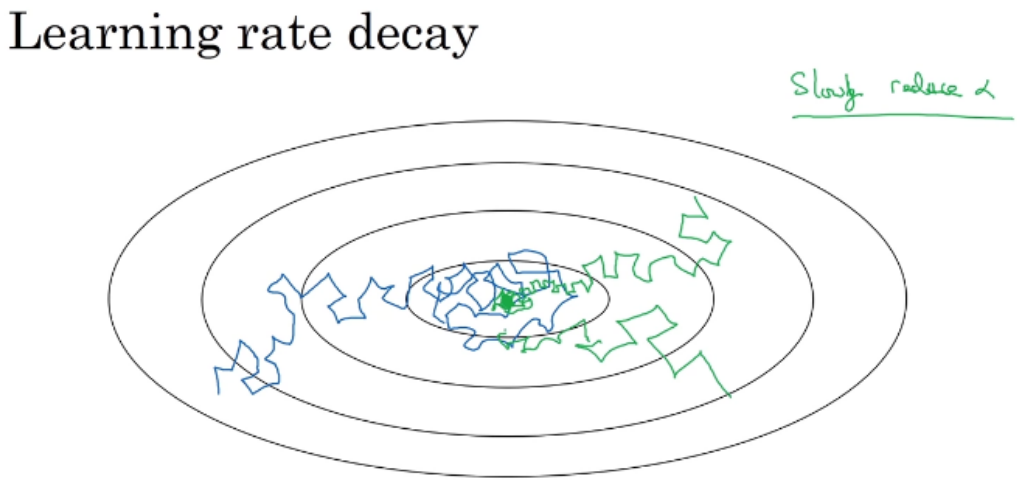

Figure 26: Some final step Learning Rate Decay to avoid divergence

During neural network training, one effective strategy to improve convergence is to gradually reduce the learning rate α over time, a technique known as **Learning Rate Decay**. The motivation behind this is straightforward:

- Early in training, large learning rates allow the model to learn quickly and make significant progress toward the minima.
- Later, as the model approaches convergence, smaller learning rates enable fine-tuning without oscillations or divergence.

Without learning rate decay, mini-batch gradient descent might keep oscillating around the minimum due to noise in mini-batches, preventing convergence.

Mathematical Formulation

A commonly used decay rule is the **time-based decay**, expressed as:

$$\alpha = \frac{\alpha_0}{1 + \text{decay\_rate} \times \text{epoch\_num}}$$

where:

- α_0 = initial learning rate,
- decay_rate = hyperparameter controlling the rate of decay,
- epoch_num = number of epochs completed.

Example: If $\alpha_0 = 0.2$ and decay_rate $= 1$:

Epoch 1: $\alpha = 0.1$
Epoch 2: $\alpha = 0.067$
Epoch 3: $\alpha = 0.05$
Epoch 4: $\alpha = 0.04$

Thus, as training continues, the learning rate decreases gradually, leading to smaller and more stable updates.

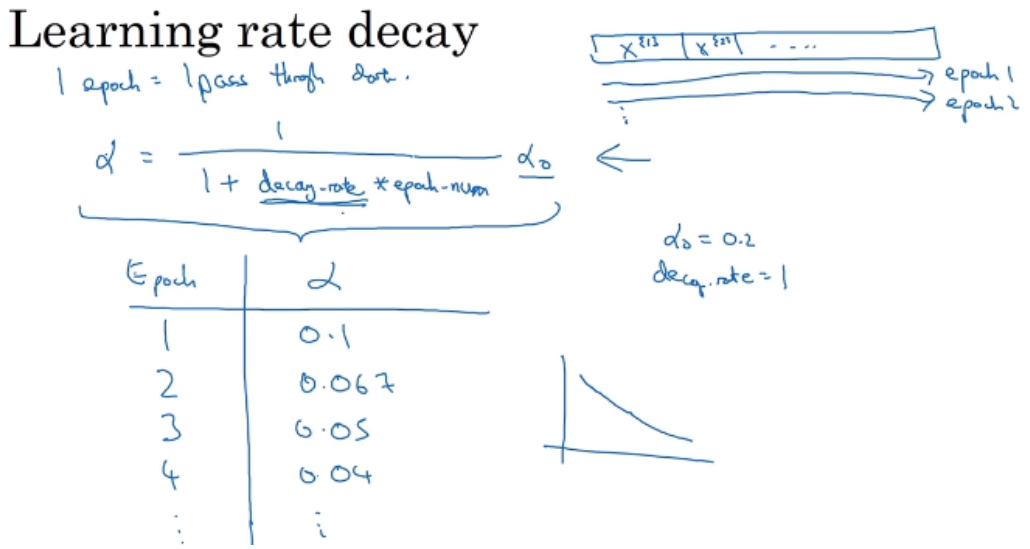

Figure 27: Illustration of Learning Rate Decay

Alternative Decay Methods

1. Exponential Decay:

$$\alpha = \alpha_0 \times k^{\text{epoch\_num}}, \quad k < 1$$

Typically $k = 0.95$. This decays the learning rate exponentially fast.

2. Square Root Decay:

$$\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \times \alpha_0$$

This slower decay is often used for large datasets.

3. Step (Staircase) Decay:

$$\alpha = \alpha_0 \times 0.5^{\lfloor \frac{\text{epoch\_num}}{k} \rfloor}$$

The learning rate drops by half every k epochs, producing discrete decreases.

4. Manual Decay: In some cases, practitioners manually reduce α when progress plateaus. This is useful for long training runs, but not scalable to large experiments.

Practical Notes

- The initial learning rate α_0 is the most important hyperparameter.
- The decay rate can be tuned, but default small values (e.g., 0.001–0.01) often work well.
- Learning rate decay is helpful but typically lower priority than getting a good fixed α.
- Often combined with optimizers such as Adam, RMSprop, or SGD with Momentum.

Summary of Common Decay Methods

Decay Type	**Formula**	**Typical Parameter**	**Behavior**
Time-based	$\frac{\alpha_0}{1+d\times t}$	$d = 0.01$	Smooth linear decay
Exponential	$\alpha_0 \times k^t$	$k = 0.95$	Rapid early decay
Square Root	$\frac{k}{\sqrt{t}} \times \alpha_0$	$k = 1$	Slow, stable decay
Step	$\alpha_0 \times 0.5^{\lfloor t/k \rfloor}$	$k = 5$	Sudden drops
Manual	—	—	Human-controlled

Table 1: Summary of Learning Rate Decay Methods

Learning rate decay is a powerful technique to refine convergence by balancing fast early learning with precise late-stage optimization.

23. Local Optima and the Problem of Plateaus

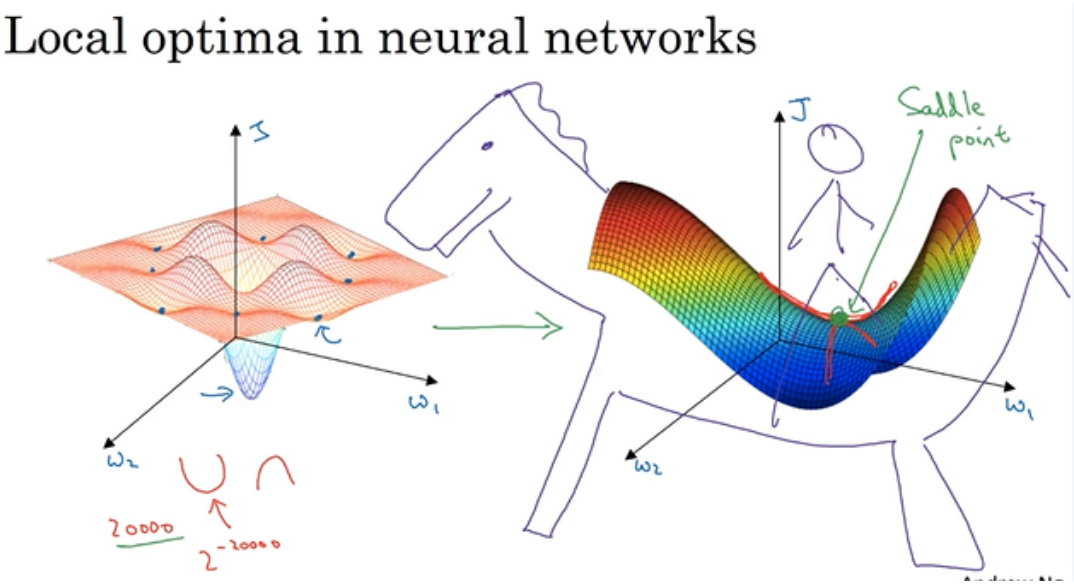

Figure 28: Visualization of Local Optima in Neural Network Cost Surface

In the early stages of deep learning, researchers were concerned that optimization algorithms might get stuck in bad **local optima**. However, as the field has matured, our understanding of the cost surfaces of neural networks has evolved significantly. It is now understood that, in high-dimensional parameter spaces, true local minima are relatively rare, and most points of zero gradient correspond instead to **saddle points**.

Local Optima vs. Saddle Points

Consider a cost function $J(W_1, W_2)$ plotted in two dimensions. It is easy to visualize multiple local minima and maxima, leading to the earlier belief that gradient descent could frequently get trapped in poor local minima. However, neural networks operate in very high-dimensional spaces — often with tens or hundreds of thousands of parameters.

In a high-dimensional space:

- For a point to be a **local minimum**, the curvature must be positive (convex-like) along *all* directions.
- For a point to be a **saddle point**, the curvature is positive in some directions and negative (concave-like) in others.

Thus, the probability of encountering a true local minimum in a 20,000-dimensional parameter space is extremely small (roughly $2^{-20,000}$), whereas encountering saddle points is much more common.

Saddle Point Intuition

A saddle point can be visualized as a region that curves upward in one direction and downward in another, similar to a horse saddle. At such a point, the gradient is zero, but it is not a true minimum. Optimization algorithms may slow down near these regions, since the gradient provides little direction for progress.

$$\nabla J(W) = 0 \quad \text{but} \quad \exists\, v_1, v_2 \text{ such that } \frac{\partial^2 J}{\partial v_1^2} > 0,\ \frac{\partial^2 J}{\partial v_2^2} < 0$$

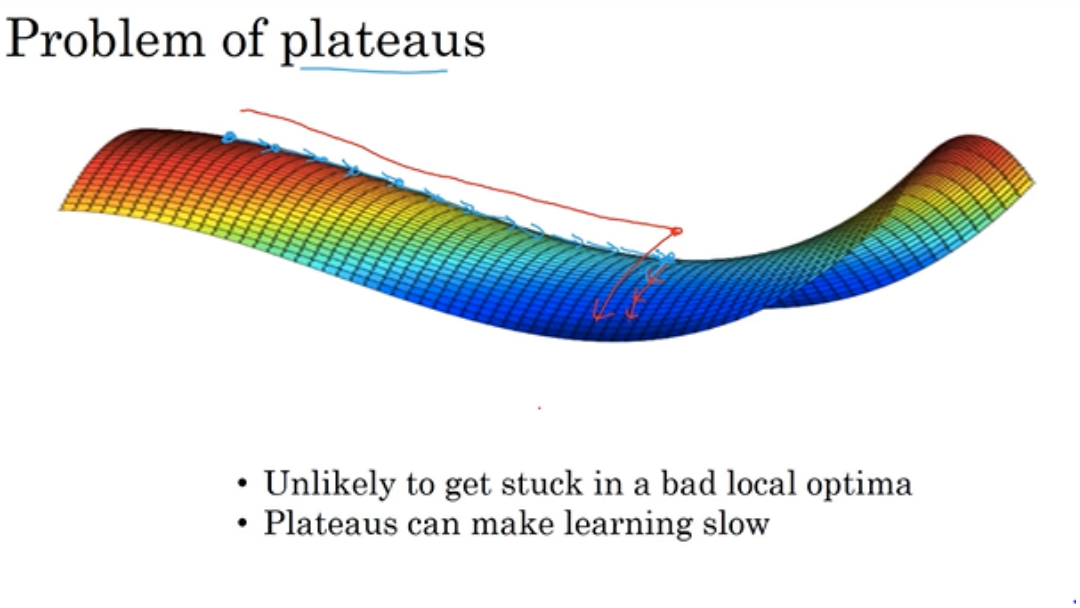

Figure 29: The Problem of Plateaus in High-Dimensional Optimization

The Problem of Plateaus

While local optima are rarely a concern, **plateaus** can significantly slow down learning. A plateau is a region where the gradient is very small or nearly zero for an extended area, making it difficult for gradient descent to progress.

- On a plateau, updates become tiny because $\nabla J(W) \approx 0$.
- As a result, the optimizer moves slowly, "wandering" across the flat region before escaping.
- Random perturbations or noise from mini-batch updates may eventually push the optimizer off the plateau.

Role of Advanced Optimizers

Algorithms such as **Momentum**, **RMSProp**, and **Adam** help overcome plateaus by:

- Maintaining a moving average of gradients (momentum) to keep consistent direction.
- Adapting learning rates across dimensions (RMSProp, Adam), allowing faster movement in flat regions.

These techniques make it possible to traverse plateaus faster and escape saddle regions efficiently.

Key Takeaways

- In high-dimensional neural networks, true local minima are rare — most zero-gradient points are saddle points.
- Plateaus, not local minima, are the main cause of slow convergence.
- Optimization algorithms like Adam effectively mitigate plateau effects and accelerate convergence.
- Our intuition from low-dimensional plots does not accurately represent the complexity of high-dimensional cost surfaces.

In summary, modern optimization challenges in deep learning are less about getting trapped in poor local optima, and more about efficiently navigating through flat regions (plateaus) and saddle points within high-dimensional loss landscapes.

24. Hyperparameter Tuning

Training deep neural networks involves setting numerous **hyperparameters** that significantly influence model performance. In this section, we will discuss how to systematically organize your hyperparameter tuning process to make it more efficient and effective.

Hyperparameter Tuning

Hyperparameter tuning (also called *hyperparameter optimization*) is the process of finding the best set of hyperparameters for a machine learning model — those that make the model perform best on unseen data.

24.1.1 What are Hyperparameters?

Hyperparameters are the settings you choose before training a model — they are not learned automatically from the data.
Examples:

Model Type	Common Hyperparameters
Linear/Logistic Regression	Regularization strength (λ)
Neural Networks	Learning rate, batch size, number of layers, number of neurons
Decision Trees	Maximum depth, minimum samples per leaf
Random Forests	Number of trees, max features
Gradient Boosting	Learning rate, number of estimators
SVM	Kernel type, C, γ

24.1.2 Summary

Term	Description
Hyperparameters	Settings that control the learning process
Tuning	Searching for the best hyperparameters
Goal	Maximize model performance on unseen data

Common Hyperparameters in Deep Learning

When training neural networks, you may encounter the following key hyperparameters:

- Learning rate (α)
- Momentum term (β) or Adam parameters (β_1, β_2, ϵ)
- Number of layers and hidden units
- Learning rate decay parameters
- Mini-batch size

Among these, the **learning rate** α is generally the most critical hyperparameter to tune. After α, the following parameters are often adjusted next:

- The momentum term β (default value $\beta = 0.9$)
- The mini-batch size
- The number of hidden units

These are followed by:

- The number of layers
- Learning rate decay parameters

When using the Adam optimization algorithm, it is common to keep the default settings:

$$\beta_1 = 0.9, \quad \beta_2 = 0.999, \quad \epsilon = 10^{-8}$$

as they generally perform well without further tuning.

Which Hyperparameters Matter Most?

1. **Highest priority:** Learning rate (α)
2. **Second priority:** Momentum (β), mini-batch size, and number of hidden units
3. **Third priority:** Number of layers and learning rate decay

Although these priorities are widely accepted, different practitioners may have slightly different intuitions depending on their specific architectures and datasets.

Grid Search vs. Random Search

In early machine learning, hyperparameter search often used a **grid search** approach:

- Choose a fixed number of discrete values for each hyperparameter.
- Evaluate all combinations (e.g., a 5×5 grid tests 25 models).

While grid search is simple, it becomes inefficient in deep learning where some hyperparameters (like α) are far more important than others.

Example: If hyperparameter 1 is the learning rate α, and hyperparameter 2 is the Adam parameter ϵ, then:

- The choice of α significantly affects performance.
- The choice of ϵ often has negligible impact.

A grid search may waste most trials varying ϵ unnecessarily.

Random Search

Instead of grid search, **random search** samples hyperparameter combinations randomly. This approach allows more diverse exploration of values for the most influential hyperparameters.

- Random search covers the hyperparameter space more efficiently.
- It increases the likelihood of discovering effective settings for critical parameters like α.
- For N trials, you can test N distinct values of α rather than repeating similar values.

Random search becomes even more powerful when tuning many hyperparameters (3 or more), as it samples more effectively within the high-dimensional search space.

Coarse-to-Fine Search Strategy

Another effective practice is to adopt a **coarse-to-fine** tuning process:

1. Begin with a wide (coarse) random search over the entire hyperparameter space.
2. Identify the region or cluster of hyperparameters that perform best.
3. Conduct a finer-grained search around that promising region.

This hierarchical approach ensures efficient use of computational resources by focusing search efforts where the most promising results were found.

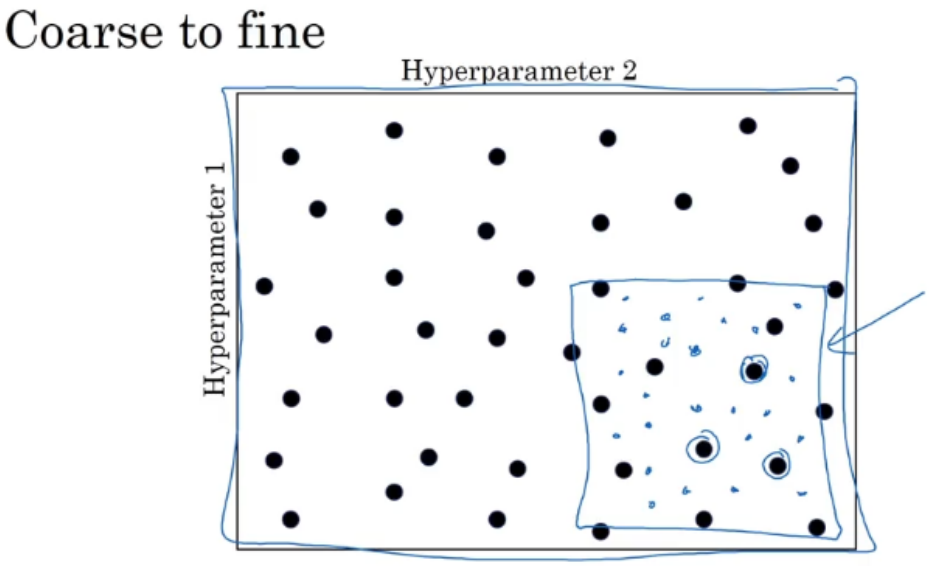

Figure 30: Illustration of the Coarse-to-Fine Hyperparameter Search Strategy

Evaluating Hyperparameter Performance

Once multiple configurations have been tested, select the combination that achieves the best result on the:

- **Development (validation) set**, or
- **Training objective**, depending on the experiment design.

This ensures that tuning decisions are based on generalization performance rather than overfitting the training data.

Key Takeaways

- The learning rate α is the most important hyperparameter to tune.
- Use **random search** instead of grid search for better coverage and efficiency.

- Employ a **coarse-to-fine strategy** to focus search efforts on promising regions.
- Default Adam parameters $(\beta_1, \beta_2, \epsilon)$ typically work well without tuning.
- Hyperparameter tuning is inherently empirical — iterative experimentation is essential.

By applying these principles, you can organize a more systematic and efficient hyperparameter tuning process that significantly improves model performance.

25. Sampling Hyperparameters on the Right Scale

In the previous section, we saw that random sampling allows for more efficient exploration of the hyperparameter space compared to grid search. However, random sampling does not necessarily mean sampling uniformly across the range of possible values. It is crucial to select the **appropriate scale** for sampling each hyperparameter.

Linear vs. Logarithmic Sampling

For certain hyperparameters, such as the number of hidden units $n^{[l]}$ or the number of layers L, it is reasonable to sample uniformly on a **linear scale**. For example:

- Number of hidden units: $n^{[l]} \in [50, 100]$
- Number of layers: $L \in \{2, 3, 4\}$

In such cases, sampling uniformly (or using grid search) is appropriate because the scale is linear and differences between values are meaningful.

However, for other hyperparameters such as the learning rate α, uniform sampling on a linear scale is inefficient. Suppose α ranges between 0.0001 and 1. Sampling uniformly across this interval would allocate 90% of samples between 0.1 and 1, wasting most resources in a narrow region.

Instead, α should be sampled on a **logarithmic scale**, as follows:

$$r = a + (b - a) \times \text{rand}()$$

$$\alpha = 10^r$$

where $a = \log_{10}(0.0001) = -4$ and $b = \log_{10}(1) = 0$. This ensures that α is sampled uniformly between 10^{-4} and 10^0 on a logarithmic scale, allowing more balanced exploration across orders of magnitude.

Example in Python

```
r = -4 * np.random.rand()
alpha = 10 ** r
```

Here, α is sampled between 10^{-4} and 1, effectively exploring values on the log scale.

Sampling for β (Exponential Moving Average)

When tuning β, such as in exponentially weighted averages, the range is typically close to 1 (e.g., $0.9 \leq \beta \leq 0.999$). In this case, linear sampling again performs poorly because small changes in β near 1 cause large effects due to the term $\frac{1}{1-\beta}$.

To handle this correctly, we instead sample over the range of $(1 - \beta)$ on a log scale:

$$r \sim U(-3, -1)$$

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

This approach samples more densely near $\beta = 1$, where the algorithm is most sensitive to variations.

Sensitivity of β

When β is close to 1, even minor changes significantly alter the effective averaging window:

$$\text{Averaging window} \approx \frac{1}{1 - \beta}$$

- $\beta = 0.9 \Rightarrow$ averaging over ~ 10 values
- $\beta = 0.999 \Rightarrow$ averaging over ~ 1000 values

Thus, sampling densely near $\beta = 1$ ensures more precise control over this sensitive range.

Summary

- Use linear scale for discrete or small-range hyperparameters (e.g., number of layers).
- Use log scale for continuous hyperparameters that vary over orders of magnitude (e.g., learning rate α, momentum β).
- Sampling α or $(1 - \beta)$ on a logarithmic scale ensures efficient exploration.
- Coarse-to-fine search can still refine the results even if the initial scale choice is imperfect.

By choosing the right scale for each hyperparameter, you can make your search process more efficient and avoid wasting computational resources on less relevant regions.

26. Organizing the Hyperparameter Search Process

Having explored various methods for hyperparameter tuning, it is now important to understand how to **organize and manage** the overall hyperparameter search process. This section discusses practical strategies for maintaining good hyperparameter performance over time and introduces two main approaches for conducting experiments based on available computational resources.

1. Cross-Fertilization of Ideas Across Domains

Deep learning applications span diverse domains such as computer vision, speech recognition, and natural language processing (NLP). Over time, there has been significant **cross-fertilization** of techniques:

- Architectures like Convolutional Neural Networks (CNNs) and Residual Networks (ResNets) from vision have inspired methods in speech and NLP.
- Conversely, concepts developed in speech recognition have found applications in text understanding.

Because of these interactions, insights from one domain may guide your intuition in another. However, it is crucial to remember that **hyperparameter intuitions can become stale**. Changes in:

- Data distribution,
- Hardware (e.g., upgraded GPUs/servers), or
- Algorithmic refinements

can all shift the optimal hyperparameter configuration. **Recommendation:** Periodically re-evaluate your hyperparameter settings (e.g., every few months) to ensure continued performance.

2. Two Strategies for Hyperparameter Search

There are two main strategies for managing hyperparameter exploration, depending on available computational resources.

(a) The Panda Approach — Babysitting One Model

This strategy is used when computational resources are limited, and you can only train one (or a few) models at a time.

- Start with initial random hyperparameters and begin training.

- Monitor key metrics (cost function J, training error, or validation error) daily.
- Adjust parameters gradually based on performance trends.
- For instance:
  - Day 1: Start with base learning rate.
  - Day 2: Increase learning rate slightly if convergence is slow.
  - Day 3: Adjust momentum or decay if learning oscillates.
- If a configuration performs worse, revert to the previous day's settings.

This "babysitting" process continues iteratively, refining one model over several days or weeks. It is careful but slow, resembling how a panda raises one cub at a time—investing deep attention into a single model.

(b) The Caviar Approach — Training Many Models in Parallel

When sufficient computational capacity (e.g., multiple GPUs or clusters) is available, the preferred approach is to train **many models in parallel**, each with a different set of hyperparameters.

- Launch multiple training runs simultaneously, each exploring different combinations of α, β, batch size, or architecture.
- Track their performance curves over time.
- Compare outcomes and select the best-performing configuration.

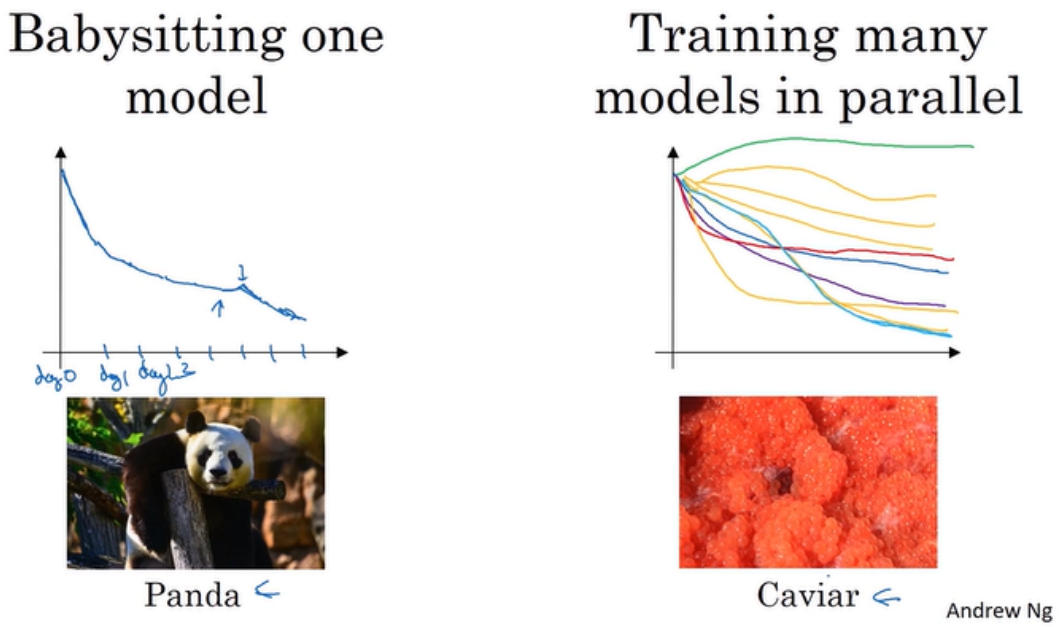

Figure 31: Comparison between the Panda and Caviar approaches to hyperparameter search.

In this setting:

- Each orange curve represents a model's learning trajectory (cost or validation error).
- Some models diverge, while others converge faster or reach better minima.
- The best-performing model is selected for further refinement.

This "caviar" strategy mirrors how fish lay many eggs at once, paying little attention to each individually—yet increasing the overall chance of finding an optimal outcome.

3. Choosing Between the Two Approaches

- **Panda Approach:** Use when training is expensive or datasets are extremely large (e.g., in computer vision or online advertising).
- **Caviar Approach:** Use when parallel computation is feasible, enabling rapid exploration of the hyperparameter space.

Even when using the panda approach, you can still train a few models sequentially (e.g., weekly) to emulate multiple trials over time.

4. Summary

- Hyperparameter intuitions can degrade over time—retest periodically.
- Choose your search strategy based on computational resources.
- **Panda:** Refine one model iteratively with careful monitoring.
- **Caviar:** Train many models in parallel and pick the best.

By understanding and balancing these two approaches, you can make your hyperparameter tuning process more systematic, adaptive, and efficient.

27. Batch Normalization

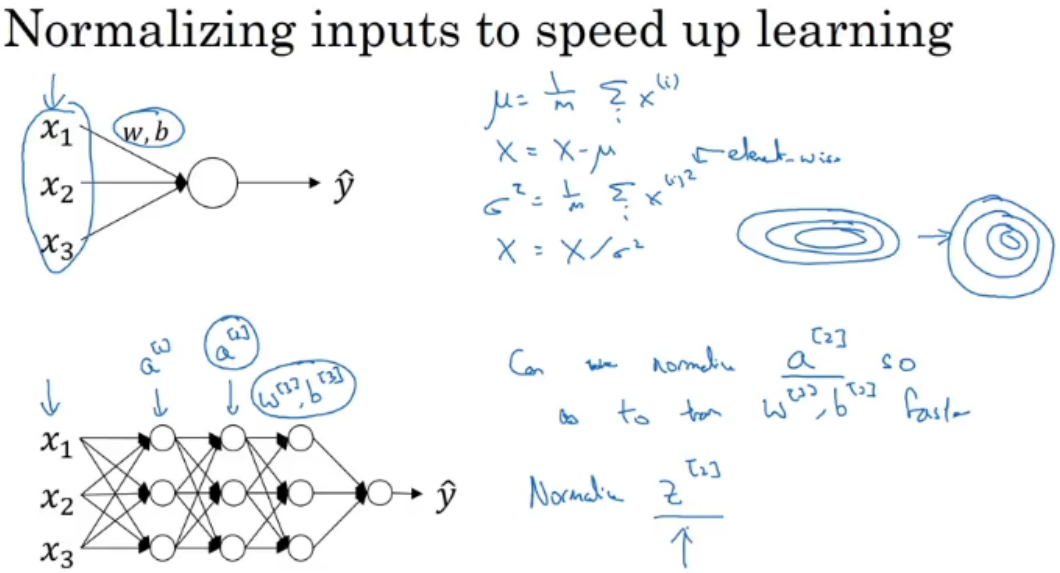

Figure 32: Batch Normalization Process

In the rise of deep learning, one of the most important ideas has been an algorithm called **Batch Normalization (BatchNorm)**, introduced by **Sergey Ioffe** and **Christian Szegedy**. Batch normalization simplifies the hyperparameter tuning process, makes neural networks more robust, and enables much easier training of deep models. It expands the range of hyperparameters that work well and stabilizes the optimization process.

Motivation

When training models such as logistic regression, you might recall that **normalizing input features** (computing means, subtracting them, and dividing by the variances) can significantly speed up learning. It transforms the cost contours from elongated to more circular, which helps gradient descent converge faster.

In deep neural networks, however, we have multiple layers — with activations $a^{[1]}, a^{[2]}, a^{[3]}, \ldots$. Suppose we are training parameters $(W^{[3]}, b^{[3]})$; their input is $a^{[2]}$. So, if we normalize $a^{[2]}$, the training of $(W^{[3]}, b^{[3]})$ becomes more efficient. This is the key idea of **Batch Normalization**: normalize the internal activations during training, not just the inputs.

Although some researchers debate whether to normalize before or after activation, it is more common to normalize **pre-activation values** $z^{[l]}$ rather than $a^{[l]}$. Therefore, the following explanation focuses on normalizing $z^{[l]}$.

Computation Steps

Given a layer with activations (or pre-activations) $z_1, z_2, \ldots, z_m$:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} z_i$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(z_i - \mu)^2$$

$$z_i^{norm} = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Here, ϵ is a small constant added for numerical stability.

After normalization, we introduce two **learnable parameters**, γ and β, to allow the model to rescale and shift the normalized values:

$$\tilde{z}_i = \gamma z_i^{norm} + \beta$$

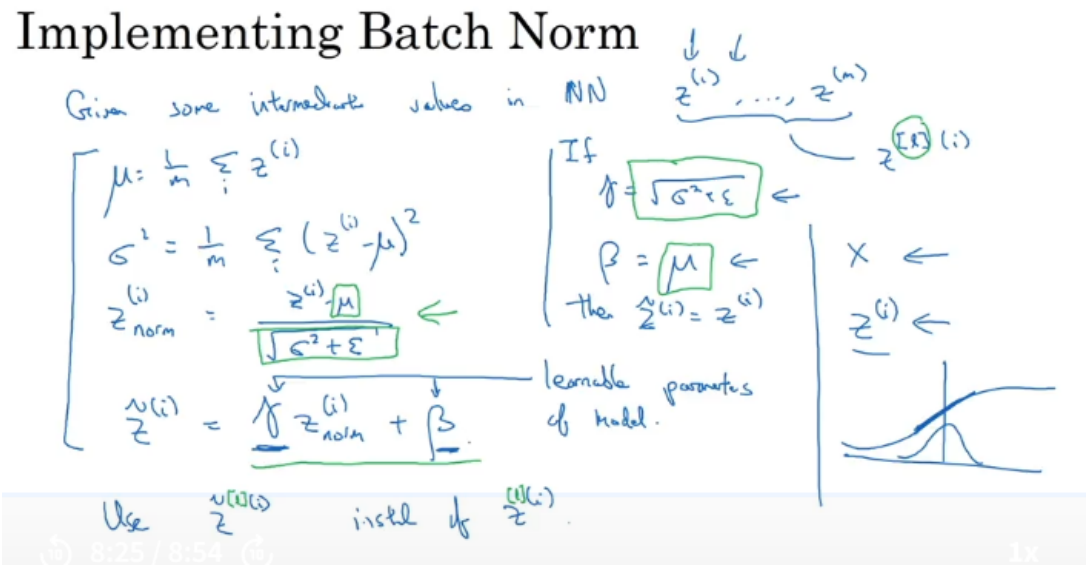

Figure 33: Batch Normalization Process

Explanation

The parameters γ and β enable the network to flexibly control the mean and variance of hidden activations. For instance:

- If $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$, then $\tilde{z}_i = z_i$, meaning normalization can be completely "undone".
- For other values, γ and β allow the model to adjust the scale and mean of activations.

Thus, Batch Normalization does not restrict activations to have mean 0 and variance 1—it gives the network the ability to learn an appropriate distribution.

Integration in Neural Networks

After computing $\tilde{z}_i$, these normalized values are passed to subsequent layers instead of the original z_i values. This normalization is applied to each layer independently.

Key Insights

- Batch Normalization extends feature normalization to internal hidden layers.
- It stabilizes training and allows much higher learning rates.
- It reduces sensitivity to initialization.
- It provides a smoother optimization landscape.

With the parameters γ and β, the model can flexibly maintain a desirable activation distribution. This technique has become a foundational element in nearly all modern deep learning architectures.

28. Batch Normalization in Deep Networks

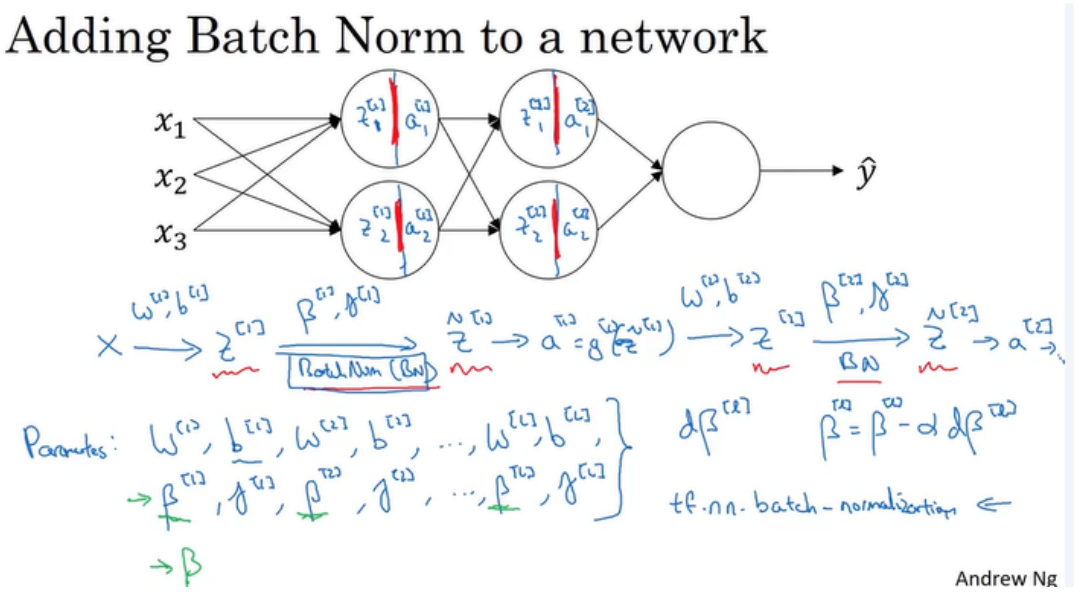

Figure 34: Batch Normalization Applied Between $Z^{[l]}$ and $A^{[l]}$ in a Deep Neural Network

After understanding the basic equations of Batch Normalization for a single hidden layer, let us now see how it integrates into the training of a **deep neural network**.

Each neuron performs two main computations:

1. Compute $Z = WA^{[l-1]} + b$
2. Apply the activation function $A = g(Z)$

Without Batch Normalization, this process continues layer by layer:

$$X \to Z^{[1]} \to A^{[1]} \to Z^{[2]} \to A^{[2]} \to \dots$$

With Batch Normalization, we insert a normalization step between $Z^{[l]}$ and $A^{[l]}$. For each layer l:

$$Z^{[l]} \xrightarrow{\text{BatchNorm}} \tilde{Z}^{[l]} \xrightarrow{g^{[l]}} A^{[l]}$$

The Batch Norm step is governed by two additional parameters for each layer:

$$\beta^{[l]},\ \gamma^{[l]}$$

so that

$$\tilde{Z}^{[l]} = \gamma^{[l]} Z^{[l]}_{\text{norm}} + \beta^{[l]}$$

These β and γ parameters are learnable and updated during training via gradient descent or other optimization algorithms.

Mini-Batch Normalization

In practice, Batch Normalization is not computed across the entire dataset, but rather across each **mini-batch**. For a given mini-batch, we compute:

$$\begin{aligned}
\mu^{[l]} &= \frac{1}{m}\sum_{i=1}^{m} Z_i^{[l]} \\
\sigma^{2[l]} &= \frac{1}{m}\sum_{i=1}^{m} (Z_i^{[l]} - \mu^{[l]})^2 \\
Z^{[l]}_{\text{norm},i} &= \frac{Z_i^{[l]} - \mu^{[l]}}{\sqrt{\sigma^{2[l]} + \epsilon}} \\
\tilde{Z}_i^{[l]} &= \gamma^{[l]} Z^{[l]}_{\text{norm},i} + \beta^{[l]}
\end{aligned}$$

Here, m is the number of examples in the mini-batch, and ϵ ensures numerical stability.

This process repeats for each mini-batch during training:

$$X^{(1)}, X^{(2)}, \dots, X^{(T)} \quad \text{(each mini-batch processed independently)}$$

Parameterization Simplification

Recall that:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

However, during Batch Normalization, the mean $\mu^{[l]}$ of $Z^{[l]}$ is subtracted out. Thus, adding a bias term $b^{[l]}$ becomes redundant because it will be canceled out by the normalization step. Therefore, we can eliminate the bias term:

$$Z^{[l]} = W^{[l]} A^{[l-1]}$$

and replace its functionality with $\beta^{[l]}$, which effectively controls the shift of the normalized output.

Hence, the parameters per layer become:

$$\{W^{[l]}, \gamma^{[l]}, \beta^{[l]}\}$$

and the bias term $b^{[l]}$ is removed.

The dimensions of these parameters are:

$$W^{[l]} : (n^{[l]}, n^{[l-1]}) \quad \gamma^{[l]} : (n^{[l]}, 1) \quad \beta^{[l]} : (n^{[l]}, 1)$$

where $n^{[l]}$ is the number of hidden units in layer l.

Training Process with Batch Norm

During training with mini-batch gradient descent:

1. For each mini-batch $t = 1, \ldots, T$:
    - Perform forward propagation, replacing $Z^{[l]}$ with $\tilde{Z}^{[l]}$ in each layer.
    - Compute the loss function J.
    - Perform backpropagation to compute gradients for $W^{[l]}, \gamma^{[l]}, \beta^{[l]}$.
    - Update parameters:

$$\begin{aligned} W^{[l]} &\leftarrow W^{[l]} - \alpha\, dW^{[l]} \\ \beta^{[l]} &\leftarrow \beta^{[l]} - \alpha\, d\beta^{[l]} \\ \gamma^{[l]} &\leftarrow \gamma^{[l]} - \alpha\, d\gamma^{[l]} \end{aligned}$$

      where α is the learning rate.

Integration with Other Optimizers

Batch Normalization parameters β and γ can be updated using not just gradient descent, but also advanced optimization algorithms such as:

- Gradient Descent with Momentum

- RMSProp
- Adam

Summary

- Batch Normalization normalizes activations within each mini-batch, stabilizing training.
- The bias term $b^{[l]}$ is unnecessary and replaced by $\beta^{[l]}$.
- Each layer learns its own normalization parameters $\gamma^{[l]}$ and $\beta^{[l]}$.
- The process can be easily implemented in most deep learning frameworks with a single line of code.

Batch Normalization, when integrated across all layers, allows deep networks to train faster and more reliably, even with higher learning rates or imperfect hyperparameter tuning.

29. Why Does Batch Normalization Work?

Batch Normalization (Batch Norm) is a key technique that stabilizes and accelerates the training of deep neural networks. One intuition is that it normalizes the intermediate activations of the network so that each layer receives inputs with a consistent distribution. This process is similar to normalizing input features (X) to have zero mean and unit variance before training, which helps models learn faster.

Covariate Shift: The Fundamental Idea

Before understanding how Batch Norm helps, we first need to recall the concept of **covariate shift**. Covariate shift occurs when the distribution of input features X changes, even though the underlying mapping from $X \to Y$ remains the same. When this happens, a model trained on one input distribution might perform poorly on another, because it has not seen data drawn from the new distribution.

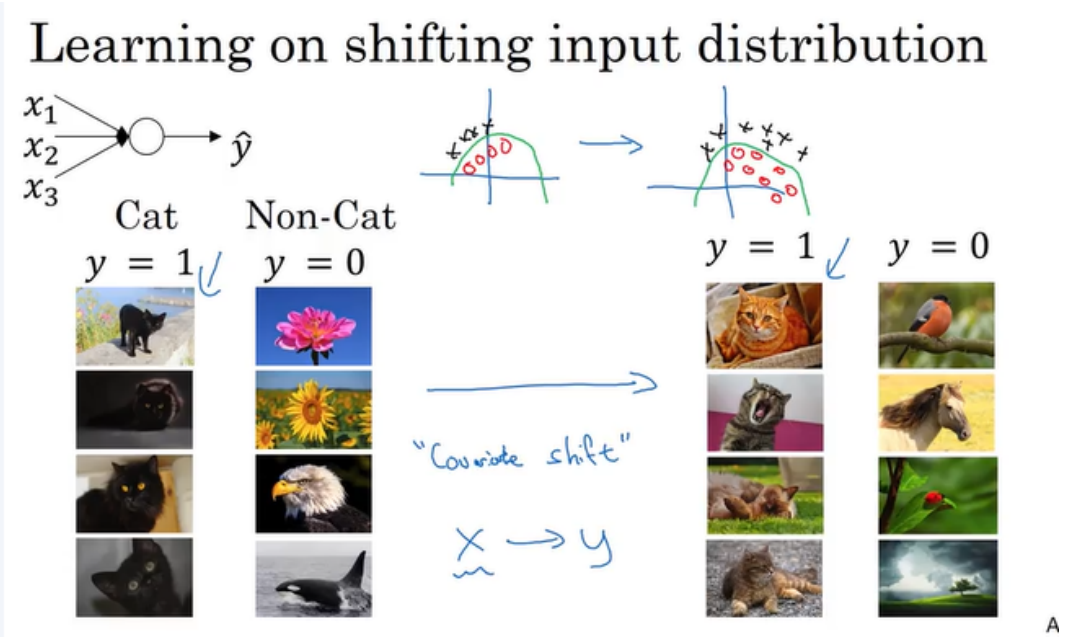

Figure 35: Covariate shift example in a cat vs. non-cat classification problem. Training data contains black cats, but test data includes colored cats—so the input distribution changes while the classification task remains the same.

As shown above, suppose a model is trained to classify *black cats vs. non-cats*. If it is later tested on *colored cats*, the input distribution has shifted, even though the correct labeling rule (cat or not) is unchanged. The model may fail because it has not learned to generalize across the new input distribution.

Internal Covariate Shift in Deep Networks

The same phenomenon occurs inside deep neural networks and is called **internal covariate shift**. Each hidden layer receives as input the activations of the previous layer. During training, as earlier layers continuously update their parameters $(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots)$, the distribution of these activations keeps changing. As a result, the later layers must constantly adapt to these new distributions, which slows down the overall training process and makes optimization harder.

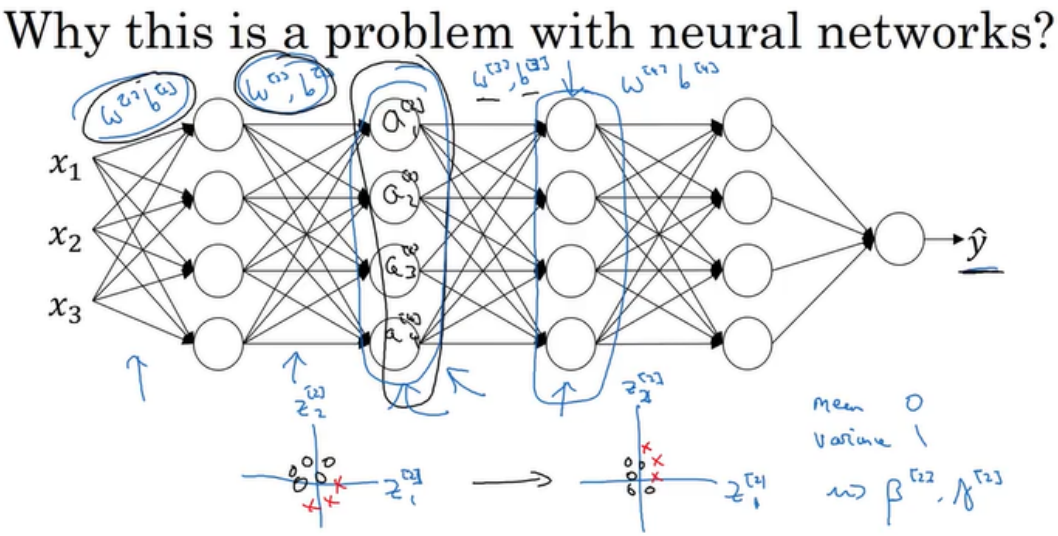

Figure 36: Internal covariate shift: distributions of activations change across training iterations, forcing deeper layers to continually readjust.

Batch Normalization addresses this issue by normalizing the activations (or pre-activations) $Z^{[l]}$ in each layer. For a given mini-batch, it computes:

$$\mu^{[l]} = \frac{1}{m}\sum_{i=1}^{m} Z^{[l](i)}, \qquad \sigma^{2[l]} = \frac{1}{m}\sum_{i=1}^{m}(Z^{[l](i)} - \mu^{[l]})^2$$

Then, each activation is normalized and re-scaled as follows:

$$\hat{Z}^{[l](i)} = \frac{Z^{[l](i)} - \mu^{[l]}}{\sqrt{\sigma^{2[l]} + \epsilon}}, \qquad \tilde{Z}^{[l](i)} = \gamma^{[l]}\hat{Z}^{[l](i)} + \beta^{[l]}$$

This ensures that the inputs to each layer maintain a stable mean and variance during training, mitigating internal covariate shift and allowing each layer to learn more independently.

Regularization and Mini-Batch Effects

Because Batch Norm computes statistics from each mini-batch rather than the entire dataset, it introduces small fluctuations in the mean and variance estimates. This introduces a mild **regularization effect**, somewhat similar to dropout—though weaker and more controlled. The smaller the batch size, the stronger this noise effect; conversely, larger batches make the normalization more stable.

Summary

- **Covariate shift** refers to changes in the input distribution while the true function remains unchanged.

- **Internal covariate shift** occurs inside neural networks when hidden layer activations change their distributions during training.
- Batch Norm reduces internal covariate shift by normalizing layer inputs to maintain consistent distributions.
- It speeds up convergence, stabilizes optimization, and adds a minor regularization effect.

30. Batch Normalization at Test Time

Batch Normalization processes data one mini-batch at a time during training, but at test time, we often need to process examples individually. Therefore, we must adapt how Batch Norm computes the required mean (μ) and variance (σ^2) so that the model can make predictions one example at a time.

Batch Norm During Training

During training, for each layer L and a given mini-batch of m examples, Batch Norm computes:

$$\mu^{[L]} = \frac{1}{m}\sum_{i=1}^{m} Z^{[L](i)}, \qquad \sigma^{2[L]} = \frac{1}{m}\sum_{i=1}^{m}\left(Z^{[L](i)} - \mu^{[L]}\right)^2$$

Then it normalizes and scales each activation as follows:

$$\hat{Z}^{[L](i)} = \frac{Z^{[L](i)} - \mu^{[L]}}{\sqrt{\sigma^{2[L]} + \epsilon}}, \qquad \tilde{Z}^{[L](i)} = \gamma^{[L]}\hat{Z}^{[L](i)} + \beta^{[L]}$$

Here, ϵ is a small constant added for numerical stability, and $\gamma^{[L]}, \beta^{[L]}$ are learned parameters that allow the network to scale and shift the normalized values.

Challenge at Test Time

At test (or inference) time, the model may receive a single example at a time. Computing $\mu^{[L]}$ and $\sigma^{2[L]}$ for a single example is meaningless, since there is no batch from which to calculate meaningful statistics. Thus, we need a method to estimate $\mu^{[L]}$ and $\sigma^{2[L]}$ that reflects their expected values during training.

Using Running (Exponential) Averages

During training, Batch Norm keeps track of the running averages of $\mu^{[L]}$ and $\sigma^{2[L]}$ across mini-batches using **exponentially weighted averages**. For example:

$$\text{RunningMean}^{[L]} \leftarrow \beta_{\text{EMA}} \cdot \text{RunningMean}^{[L]} + (1 - \beta_{\text{EMA}})\mu_{\text{current}}^{[L]}$$

$$\text{RunningVar}^{[L]} \leftarrow \beta_{\text{EMA}} \cdot \text{RunningVar}^{[L]} + (1 - \beta_{\text{EMA}})\sigma_{\text{current}}^{2[L]}$$

Here, β_{EMA} (commonly around 0.9 or 0.99) controls how much weight is given to recent batches. These running averages act as long-term estimates of the true mean and variance of activations for each layer.

Batch Norm During Inference

At test time, instead of recalculating statistics from the input batch, we reuse the stored running averages. The normalization step becomes:

$$\hat{Z}^{[L]} = \frac{Z^{[L]} - \text{RunningMean}^{[L]}}{\sqrt{\text{RunningVar}^{[L]} + \epsilon}}$$

$$\tilde{Z}^{[L]} = \gamma^{[L]}\hat{Z}^{[L]} + \beta^{[L]}$$

This ensures that each layer continues to receive normalized inputs consistent with what it saw during training, even when processing one example at a time.

Key Takeaways

- During training, $\mu^{[L]}$ and $\sigma^{2[L]}$ are computed on each mini-batch.
- During inference, Batch Norm uses running (exponentially weighted) averages of $\mu^{[L]}$ and $\sigma^{2[L]}$ computed during training.
- This approach ensures consistent normalization without requiring a batch of data at test time.
- The exact estimation method for μ and σ^2 is not critical—deep learning frameworks handle this automatically.

Summary

Batch Norm helps maintain stable activation distributions not only during training but also at inference by using the moving averages of mean and variance. This allows neural networks to make consistent predictions on single inputs, enabling smooth transition from training to deployment.

31. Softmax Regression and Multi-Class Classification

So far, the classification problems we have discussed involve binary outputs — for example, predicting whether an image is a cat ($y = 1$) or not ($y = 0$). However, many real-world tasks require identifying one among multiple possible classes. **Softmax Regression** is a generalization of Logistic Regression that handles such multi-class classification problems.

Motivation

Suppose you want to classify an image as either a *cat*, *dog*, *baby chick*, or *none of the above*. We can label these as:

Class 0: None, Class 1: Cat, Class 2: Dog, Class 3: Baby Chick

Let the total number of classes be denoted by C. In this case, $C = 4$, and the classes are indexed from 0 to $C - 1$.

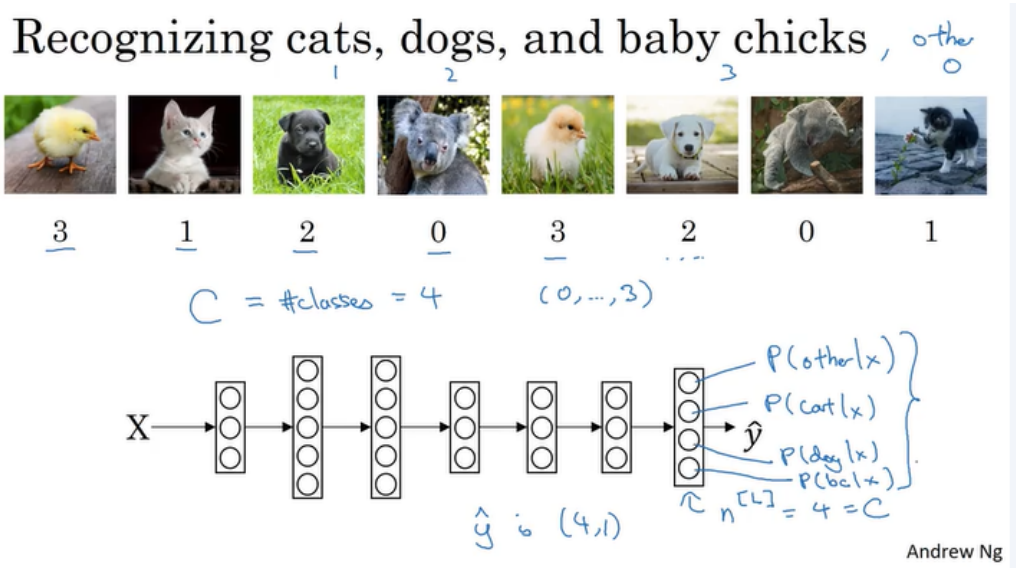

Figure 37: Example of multi-class classification setup for Softmax Regression.

Network Structure

In Softmax regression, the output layer of the neural network contains C output units — one per class. Thus, for the final layer L, the output vector $\hat{y}$ (also denoted $a^{[L]}$) has shape $(C, 1)$ and represents the predicted probabilities for each class:

$$\hat{y} = \begin{bmatrix} P(y=0|x) \\ P(y=1|x) \\ P(y=2|x) \\ \vdots \\ P(y=C-1|x) \end{bmatrix}, \quad \text{with} \quad \sum_{i=0}^{C-1} P(y=i|x) = 1$$

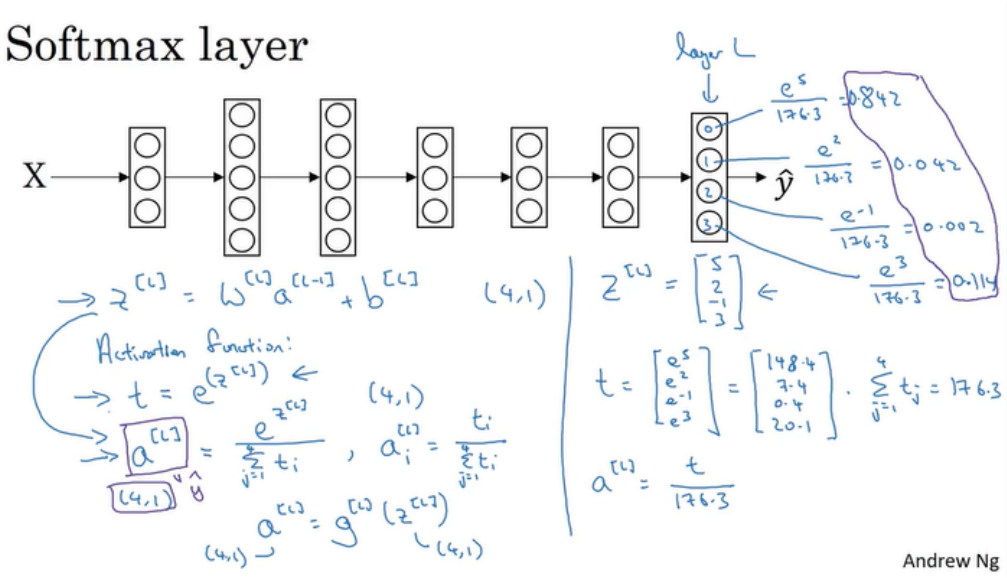

Figure 38: Neural network with Softmax output layer producing class probabilities.

Softmax Activation Function

The final layer first computes a linear transformation:

$$Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$$

where $Z^{[L]}$ is a $(C, 1)$ vector representing the raw, unnormalized scores for each class.

Then, the **Softmax activation function** converts these raw scores into probabilities:

$$A^{[L]} = \text{Softmax}(Z^{[L]})$$

Mathematically, for each element i in the vector:

$$A_i^{[L]} = \frac{e^{Z_i^{[L]}}}{\sum_{j=1}^{C} e^{Z_j^{[L]}}}$$

This ensures that:

$$0 \leq A_i^{[L]} \leq 1, \quad \text{and} \quad \sum_{i=1}^{C} A_i^{[L]} = 1$$

Example Calculation

Consider a case where the final layer outputs the raw vector:

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

Compute exponentials:

$$e^{Z^{[L]}} = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

Sum of all exponentials:

$$\text{Sum} = 148.4 + 7.4 + 0.4 + 20.1 = 176.3$$

Normalize to obtain the probabilities:

$$A^{[L]} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

Thus, the model predicts:

- Class 0 (None): 84.2%
- Class 1 (Cat): 4.2%
- Class 2 (Dog): 0.2%
- Class 3 (Baby Chick): 11.4%

Intuition

The Softmax activation converts the network's unscaled outputs (Z values) into a probability distribution over classes. It emphasizes the largest Z_i values and suppresses smaller ones, enabling clear class selection while maintaining differentiability for training.

Softmax Without Hidden Layers

A simple Softmax model with no hidden layers performs:

$$Z = WX + b, \quad \hat{Y} = \text{Softmax}(Z)$$

This is equivalent to a multi-class version of Logistic Regression, capable of forming linear decision boundaries that separate multiple classes.

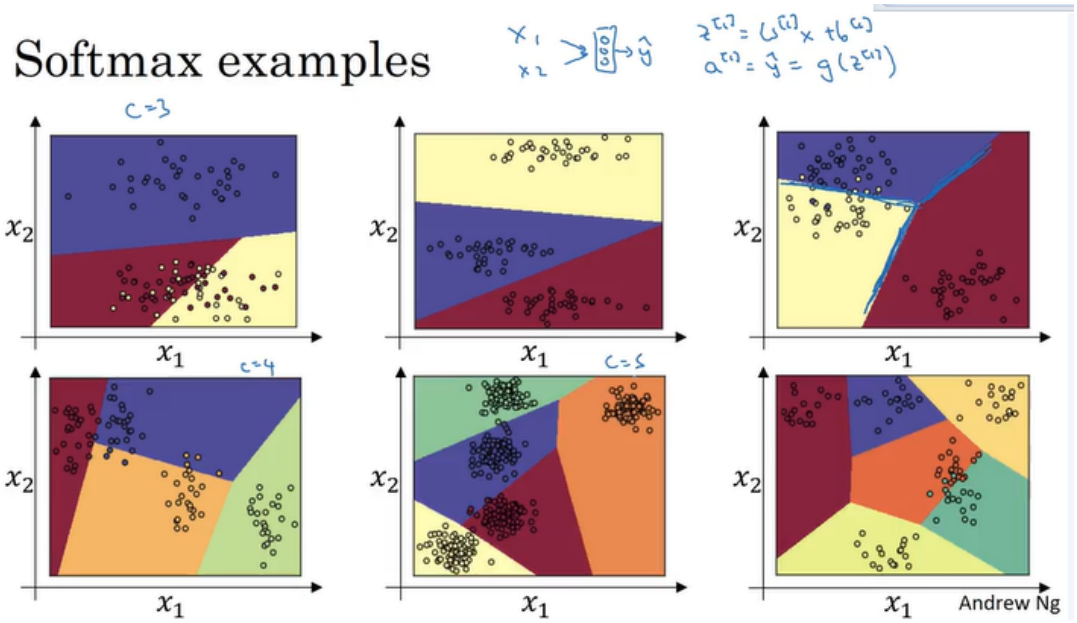

Figure 39: Decision boundaries formed by a Softmax layer for multiple classes.

Key Takeaways

- Softmax Regression generalizes Logistic Regression to multi-class problems.
- The Softmax layer converts raw scores into a probability distribution.
- Each output neuron corresponds to a class, and all probabilities sum to 1.
- A Softmax classifier with no hidden layers produces linear decision boundaries.
- Deeper networks with Softmax output layers can form complex non-linear class boundaries.

Summary

Softmax Regression is a powerful extension of binary Logistic Regression for multi-class classification tasks. By exponentiating and normalizing the output scores, it ensures probabilistic and interpretable outputs, enabling neural networks to predict across multiple classes with a single forward pass.

32. Softmax Classification and Training

In the previous section, we discussed the **Softmax activation function**. Here, we deepen our understanding of **Softmax classification** and learn how to train a model using a Softmax output layer.

From Hardmax to Softmax

Recall that the output layer computes:

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

where for $C = 4$ classes, $z^{[L]} \in \mathbb{R}^{4\times 1}$.

We define a temporary variable $t = e^{z^{[L]}}$, applied element-wise. The activation function $g^{[L]}$ is the **Softmax function**, given by:

$$a_i^{[L]} = \frac{e^{z_i^{[L]}}}{\sum_{j=1}^{C} e^{z_j^{[L]}}}$$

This normalizes the exponentiated values to produce probabilities that sum to one.

The term **"Softmax"** contrasts with **"Hardmax"**, which assigns a value of 1 to the largest element and 0 to all others. Softmax instead provides a smooth distribution over all classes, offering probabilistic confidence rather than a single-class decision.

Relation to Logistic Regression

Softmax regression generalizes logistic regression from binary classification ($C = 2$) to multi-class classification ($C > 2$). When $C = 2$, the Softmax function simplifies to:

$$a^{[L]} = \begin{bmatrix} \sigma(z) \\ 1 - \sigma(z) \end{bmatrix}$$

which is equivalent to logistic regression.

Loss Function for Softmax Classification

For a single training example, let the true label be represented by a one-hot vector:

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{and the predicted output } \hat{y} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix}$$

The loss function is defined as:

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

In this example, since only $y_2 = 1$, the loss simplifies to:

$$\mathcal{L} = -\log(\hat{y}_2)$$

This encourages the model to maximize the probability $\hat{y}_2$ of the correct class.

Cost Function and Vectorization

Over the entire training set, the cost function is:

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

For vectorized implementation:

$$Y = \begin{bmatrix} | & | & & | \\ y^{(1)} & y^{(2)} & \cdots & y^{(m)} \\ | & | & & | \end{bmatrix}, \quad \hat{Y} = \begin{bmatrix} | & | & & | \\ \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \\ | & | & & | \end{bmatrix}$$

Both Y and $\hat{Y}$ are matrices of size $C \times m$.

Gradient Computation and Backpropagation

For Softmax output, the key derivative for initializing backpropagation is:

$$dZ^{[L]} = \hat{Y} - Y$$

where $dZ^{[L]}$ has dimensions (C, m). This expression directly links the prediction error to the output logits, simplifying the gradient computation in the final layer.

Implementation Notes

During training:

- Use gradient descent (or variants like Adam) to minimize $J(W, b)$.
- Modern deep learning frameworks automatically handle backpropagation when you define the forward pass.

- Softmax regression can classify inputs into one of C classes with a probabilistic interpretation of predictions.

Conclusion: Softmax regression extends logistic regression to multiple classes and forms the foundation of multi-class classification in neural networks. With proper implementation, it enables the model to output probabilities across all categories and be trained efficiently via cross-entropy loss.