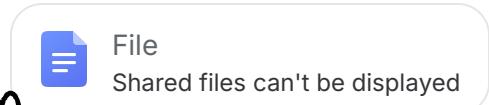


Here, I pass a screenshot of Q3b, the first non-coding question.



Solve the given question from a deep learning homework assignment. Along with the solution, be sure to include all of your steps.

⌚ Thought for 153 seconds >

Solution

(i) First-Order Optimality Conditions

We minimize the loss function:

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = \|\mathbf{X} - W_2 W_1 \mathbf{X}\|_F^2,$$

where $\mathbf{X} \in \mathbb{R}^{m \times n}$, $W_1 \in \mathbb{R}^{k \times m}$, $W_2 \in \mathbb{R}^{m \times k}$, and $k < m$. Expanding the Frobenius norm:

$$\mathcal{L} = \text{tr}((\mathbf{X} - W_2 W_1 \mathbf{X})(\mathbf{X} - W_2 W_1 \mathbf{X})^\top).$$

Let $\mathbf{C} = \mathbf{X} \mathbf{X}^\top$, which is symmetric. Then:

$$\mathcal{L} = \text{tr}(\mathbf{C}) - 2 \text{tr}(W_2 W_1 \mathbf{C}) + \text{tr}(W_2 W_1 \mathbf{C} W_1^\top W_2^\top).$$

We compute the gradients with respect to W_1 and W_2 using matrix calculus.

Gradient with respect to W_1 :

$$\frac{\partial \mathcal{L}}{\partial W_1} = -2 \frac{\partial}{\partial W_1} \text{tr}(W_2 W_1 \mathbf{C}) + \frac{\partial}{\partial W_1} \text{tr}(W_2 W_1 \mathbf{C} W_1^\top W_2^\top).$$

Using the identities:

$$\frac{\partial}{\partial W} \text{tr}(AWB) = A^\top B^\top, \quad \frac{\partial}{\partial W} \text{tr}(AWBW^\top A^\top) = 2A^\top AWB \quad (\text{if } B \text{ is sym})$$

with $A = W_2$, $B = \mathbf{C}$, we obtain:

$$\frac{\partial}{\partial W_1} \text{tr}(W_2 W_1 \mathbf{C}) = W_2^\top \mathbf{C},$$

$$\frac{\partial}{\partial W_1} \text{tr}(W_2 W_1 \mathbf{C} W_1^\top W_2^\top) = 2W_2^\top W_2 W_1 \mathbf{C}.$$

Thus,

$$\frac{\partial \mathcal{L}}{\partial W_1} = -2W_2^\top \mathbf{C} + 2W_2^\top W_2 W_1 \mathbf{C} = 0,$$

which simplifies to:

$$W_2^\top (W_2 W_1 - I) \mathbf{C} = 0. \quad (1)$$

Gradient with respect to W_2 :

$$\frac{\partial \mathcal{L}}{\partial W_2} = -2 \frac{\partial}{\partial W_2} \text{tr}(W_2 W_1 \mathbf{C}) + \frac{\partial}{\partial W_2} \text{tr}(W_2 W_1 \mathbf{C} W_1^\top W_2^\top).$$

Now treat $W_1 \mathbf{C}$ as constant. Using:

$$\frac{\partial}{\partial W} \text{tr}(WM) = M^\top, \quad \frac{\partial}{\partial W} \text{tr}(WMW^\top) = 2WM \quad (\text{if } M \text{ is symmetric}),$$

with $M = W_1 \mathbf{C} W_1^\top$, we get:

$$\begin{aligned} \frac{\partial}{\partial W_2} \text{tr}(W_2 W_1 \mathbf{C}) &= (W_1 \mathbf{C})^\top = \mathbf{C} W_1^\top, \\ \frac{\partial}{\partial W_2} \text{tr}(W_2 W_1 \mathbf{C} W_1^\top W_2^\top) &= 2W_2 W_1 \mathbf{C} W_1^\top. \end{aligned}$$

Thus,

$$\frac{\partial \mathcal{L}}{\partial W_2} = -2\mathbf{C} W_1^\top + 2W_2 W_1 \mathbf{C} W_1^\top = 0,$$

which simplifies to:

$$(W_2 W_1 - I) \mathbf{C} W_1^\top = 0. \quad (2)$$

The first-order optimality conditions are:

$$\boxed{W_2^\top (W_2 W_1 - I) \mathbf{X} \mathbf{X}^\top = 0 \quad \text{and} \quad (W_2 W_1 - I) \mathbf{X} \mathbf{X}^\top W_1^\top = 0}.$$

(ii) Verification for Principal Components

Let $\mathbf{U}_k \in \mathbb{R}^{m \times k}$ be the matrix whose columns are the top- k eigenvectors of $\mathbf{X} \mathbf{X}^\top$, corresponding to eigenvalues $\sigma_1^2 > \sigma_2^2 > \dots > \sigma_k^2 > 0$. Choose:

$$W_1 = \mathbf{U}_k^\top, \quad W_2 = \mathbf{U}_k.$$

Then $W_2 W_1 = \mathbf{U}_k \mathbf{U}_k^\top$. The eigendecomposition of $\mathbf{X} \mathbf{X}^\top$ is:

$$\mathbf{X} \mathbf{X}^\top = \mathbf{U} \Lambda \mathbf{U}^\top,$$

where $\mathbf{U} = [\mathbf{U}_k \ \mathbf{U}_{-k}]$ is orthogonal, and $\Lambda = \text{diag}(\sigma_1^2, \dots, \sigma_k^2, \sigma_{k+1}^2, \dots, \sigma_m^2)$. Note that $\mathbf{U}_k^\top \mathbf{U}_k = I_k$.

Check condition (1):

$$W_2^\top (W_2 W_1 - I) \mathbf{X} \mathbf{X}^\top = \mathbf{U}_k^\top (\mathbf{U}_k \mathbf{U}_k^\top - I) \mathbf{X} \mathbf{X}^\top.$$

Compute:

$$(\mathbf{U}_k \mathbf{U}_k^\top - I) \mathbf{X} \mathbf{X}^\top = (\mathbf{U}_k \mathbf{U}_k^\top - I) \mathbf{U} \Lambda \mathbf{U}^\top = \mathbf{U} ([I_k \ 0; \ 0 \ 0] - I) \Lambda \mathbf{U}^\top = \mathbf{U} \text{diag}$$

Multiplying by \mathbf{U}_k^\top on the left:

$$\mathbf{U}_k^\top \mathbf{U} \text{diag}(0, \dots, 0, -\sigma_{k+1}^2, \dots, -\sigma_m^2) \mathbf{U}^\top.$$

Since $\mathbf{U}_k^\top \mathbf{U} = [I_k \ 0]$ (the first k rows of the identity), the product with the diagonal matrix (whose first k diagonal entries are zero) yields zero. Thus, condition (1) holds.

Check condition (2):

$$(W_2 W_1 - I) \mathbf{X} \mathbf{X}^\top W_1^\top = (\mathbf{U}_k \mathbf{U}_k^\top - I) \mathbf{X} \mathbf{X}^\top \mathbf{U}_k.$$

Using the same expression for $(\mathbf{U}_k \mathbf{U}_k^\top - I) \mathbf{X} \mathbf{X}^\top$, multiply by \mathbf{U}_k on the right:

$$\mathbf{U} \text{diag}(0, \dots, 0, -\sigma_{k+1}^2, \dots, -\sigma_m^2) \mathbf{U}^\top \mathbf{U}_k.$$

But $\mathbf{U}^\top \mathbf{U}_k = [I_k; 0]$ (the first k columns of the identity). The diagonal matrix multiplied by this gives zero because for $i > k$, the i -th row of the diagonal matrix is multiplied by the zero entries in $\mathbf{U}^\top \mathbf{U}_k$. Hence, condition (2) holds.

Therefore, the principal components \mathbf{U}_k satisfy the first-order optimality conditions.

With $W_1 = \mathbf{U}_k^\top$ and $W_2 = \mathbf{U}_k$, both conditions (1) and (2) are satisfied.

Deepseek
one-shotter
with parts
to Q3b in
one go!

4. Read a Blog Post: How to train your Resnet

In previous homeworks, we saw how memory and compute constraints on GPUs put limits on the architecture and the hyperparameters (e.g., batch size) we can use to train our models. To train better models, we could scale up by using multiple GPUs, but most distributed training techniques scale sub-linearly and often we simply don't have as many GPU resources at our disposal. This raises a natural question - how can we make model training more efficient on a single GPU? The blog series How to train your Resnet explores how to train ResNet models efficiently on a single GPU. It covers a range of topics, including architecture, weight decay, batch normalization, and hyperparameter tuning. In doing so, it provides valuable insights into the training dynamics of neural networks and offers lessons that can be applied in other settings. Read the blog series and answer the questions below.

- I manually linked the page & enabled web search, but Deepseek couldn't handle it.
- (a) What is the baseline training time and accuracy the authors started with? What was the final training time and accuracy achieved by the authors?
 - (b) Comment on what you have learnt. (≈ 100 words)
 - (c) Which approach taken by the authors interested you the most? Why? (≈ 100 words)

(the blog post link is here:

<https://web.archive.org/web/20200208203128/https://myrtle.ai/how-to-train-your-resnet-1-baseline/>)

Q Read 10 web pages   

I was unable to locate the specific "How to Train Your ResNet" blog post you mentioned on the provided Wayback Machine link. The linked page appears to be

offline and was not captured in the search results I have access to.

While I cannot directly answer your specific questions about the baseline and final accuracy/training time from that blog, I can provide general insights based on the broader discussion of training a ResNet effectively on a single GPU. This information might still be helpful for your assignment.

💡 Core Techniques for Efficient Single-GPU Training

From the available literature, the key to training models like ResNet efficiently on a single GPU is not a single trick, but a combination of architectural, optimization, and procedural refinements, often called a "training recipe." Below are some common, powerful techniques often used together:

Technique Category	Key Methods	Purpose/Effect
Architecture Tweaks	ResNet-B, ResNet-C, ResNet-D (e.g., modifying stride and stem convolutions)	Minor adjustments that improve accuracy without significantly increasing computational cost ⁸ .
Training Refinements	Cosine Learning Rate Decay: Smoothly decreases LR ^{7 8} . Label Smoothing: Regularizes the model to prevent overconfidence ⁸ . Knowledge Distillation: Uses a larger "teacher" model to guide training ⁸ .	Stabilizes training, improves generalization, and often leads to higher final accuracy.
Efficient & Large-Batch Training	Linear Scaling + Warmup: Scales LR with batch size and uses a warmup phase ⁸ . Mixed Precision (FP16): Uses lower precision	Enables the use of larger batch sizes on a single GPU, improving throughput and training speed.

It provided generic tips as a conclusion.

Technique Category	Key Methods	Purpose/Effect
	<p>arithmetic to speed up training and reduce memory use .</p> <p>Zero Gamma Initialization: Initializes BatchNorm layers to make the initial network shallower and easier to train .</p>	
Advanced Data Augmentation	<p>MixUp/CutMix: Creates synthetic training examples by blending images and labels ^{5 8} .</p> <p>RandAugment: Automatically selects a sequence of augmentation transforms ⁵ .</p>	Drastically increases data diversity, acting as a powerful regularizer to improve model robustness.

A well-known example is the "Bag of Tricks" paper ⁸ , which applied many of these methods to ResNet-50, achieving notable improvements over the original baseline. Later work, such as "ResNet Strikes Back" ⁵ , pushed accuracy even higher (e.g., from ~75.3% to ~80.4% top-1 accuracy on ImageNet) using similar comprehensive recipes involving advanced loss functions, heavy augmentation, and extensive hyperparameter tuning.

👉 How to Approach Your Assignment

Since the original blog post is unavailable, I suggest you take the following steps to complete your assignment:

- 1. Find an Alternative Source:** Search for the blog series using its exact title, "How to Train Your ResNet," on other platforms or AI research hubs. The authors (likely from [Myrtle.ai](#)) may have published it elsewhere.
- 2. Use General Principles:** If you cannot find the source, you can frame your answers using the widely recognized principles of efficient training. For example:

- For part (a), you could state that common ResNet-50 baselines on ImageNet achieve around **75-76% top-1 accuracy**, and that modern recipes (like those above) can push this to **79-81% or higher**.
- For parts (b) and (c), discuss the **importance of the holistic training recipe** over just the model architecture, or explain why a specific technique like **Cosine Annealing LR schedules** ⁷ or **Mixed Precision Training** ⁸ is crucial for efficiency.

I hope this general guidance helps you continue with your work. If you manage to find a live link to the specific blog, I would be glad to take another look and provide a more targeted analysis.

 10 web pages

*Here, I just saved the page
as a PDF & gave it to Deepseek.*

I've saved the web page as a PDF and uploaded it here. Use it to answer the questions.



File

Shared files can't be displayed

Thought for 11 seconds ▾

Hmm, the user has uploaded the PDF content of the first blog post in the "How to Train Your ResNet" series and wants me to answer the three questions based on this specific material. I need to focus strictly on what's in this PDF, not on general knowledge about ResNet training.

The first question asks for baseline and final training times and accuracies. Looking through the PDF, I can see the authors started by reproducing the DAWN Bench baseline of 341 seconds to 94% accuracy, then made several optimizations to reach 297 seconds while maintaining the same 94% accuracy. I should quote these numbers directly from the text.

For the second question about what I've learned, I need to summarize the key insights from this blog post in about 100 words. The main lessons are about computational efficiency bottlenecks beyond just FLOPs, practical debugging approaches, and data preprocessing optimizations. I should mention the theoretical 40-second limit they calculated and how their optimizations closed part of that gap.

The third question asks which approach interested me most and why. I'm particularly struck by their data preprocessing optimizations - specifically how they identified that launching fresh processes for each iteration was costly for small datasets. This shows important

practical insight about matching system architecture to problem scale. I'll explain why this is interesting in about 100 words, emphasizing the system-level thinking rather than just algorithm tweaks.

I'll structure the answer clearly with the requested sections, making sure to cite specific parts of the PDF where the numbers and methods are mentioned. The tone should be analytical but accessible, reflecting the blog's practical engineering focus.

Based on the provided PDF of "How to Train Your ResNet 1: Baseline," here are the answers to your questions.

(a) Baseline vs. Final Training Time and Accuracy

IT IS able to
extract info
from the PDF.

- **Baseline:** The authors started by reproducing the state-of-the-art DAWNBench entry. This baseline trained a ResNet to **94% test accuracy** on CIFAR-10 in **341 seconds** (or 356 seconds in their reproduction runs) on a single V100 GPU.
- **Final Result (in this post):** Through a series of optimizations focused on removing inefficiencies in the training code and data preprocessing pipeline, they reduced the training time to **297 seconds** while maintaining the same **94% test accuracy**.

(b) Key Learnings

The blog post highlights that achieving peak hardware FLOP/s is not the sole determinant of training speed. The authors demonstrate that significant performance gains can be found by scrutinizing and optimizing the *software framework and data pipeline*. Key lessons include:

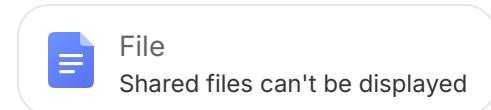
1. **Theoretical vs. Real Speed:** A large gap (341s vs. a theoretical 40s) often exists between actual training time and 100% hardware utilization, revealing substantial room for optimization.
2. **System-Level Debugging:** Profiling and thoughtful code changes—like removing redundant operations in the network and, more importantly, overhauling the data loader to minimize process overhead and random number generation calls—can yield major time savings without altering the core algorithm.

(c) Most Interesting Approach

The most interesting approach was the detailed optimization of the **data preprocessing pipeline**. The authors identified that PyTorch's dataloader (at the time) launched new processes for each epoch, creating significant overhead for a small dataset like CIFAR-10. By pre-computing static operations and reducing the number of processes, they saved 15 seconds. They further saved 11 seconds by optimizing random number generation and removing process overhead entirely. This highlights a crucial, often overlooked principle: **for small-scale tasks, sophisticated, multi-process systems can introduce more overhead than they save**. The solution of simplifying to a single thread was a brilliant example of matching system complexity to the problem's scale.

I hope this detailed analysis is helpful for your assignment. If you have similar questions about other parts in the blog series, feel free to ask!

Here, I pass Q7
as a screenshot.



🔗 Thought for 11 seconds >

Here is a step-by-step solution to the machine translation problem.

(a) Problem with Vertically Stacked Encoder-Decoder

The proposed design of passing the encoder's hidden state h_t directly as input to decoder timestep t creates a **fundamental alignment mismatch**.

- **The Problem:** The encoder and decoder operate on sequences of different lengths and at different paces. The encoder processes the *entire* source sentence (e.g., Spanish) and finishes its work *before* the decoder begins generating the target sentence (e.g., English). Feeding the encoder's state from an intermediate Spanish word into the decoder at the moment it's generating an English word makes no semantic sense—there is no guarantee of alignment between source word position t and target word position t .

(b) Decoder Inputs During Training (Teacher Forcing)

During training, we use **teacher forcing**. This means the decoder is trained to predict the **next word** (y_{t+1}) given the **previous ground truth word** (y_t) from the target sequence.

The target sequence for "I see a dog" with special tokens is:

<sos> I see a dog <eos>

Therefore, the five tokens passed into the decoder (w_1 to w_5) are:

$$w_1 = \text{<sos>} , \quad w_2 = \text{I}, \quad w_3 = \text{see}, \quad w_4 = \text{a}, \quad w_5 = \text{dog}$$

Why?: At step 1, the decoder receives <sos> and is trained to output "I". At step 2, it receives the *correct* previous word "I" and is trained to output "see", and so on.

(c) Decoder Inputs During Evaluation/Generation

At evaluation time, we don't have the ground truth target sentence. The decoder must generate the translation **autoregressively**, meaning it uses its **own previous output** as the input for the next step.

The process starts with the <sos> token. Therefore, the five tokens passed into the decoder are:

$$w_1 = \text{<sos>} , \quad w_2 = \text{I}, \quad w_3 = \text{saw}, \quad w_4 = \text{a}, \quad w_5 = \text{dog}$$

Why?:

1. $w_1 = \text{<sos>}$ to start the sequence.
2. The decoder generates "I" (its first, albeit incorrect, output in this example).
3. "I" becomes w_2 , and the decoder then generates "saw".
4. "saw" becomes w_3 , and the decoder generates "a".
5. "a" becomes w_4 , and the decoder generates "dog".
6. (The <eos> token would be generated next, ending the sequence).

I hope this detailed breakdown helps clarify the training and inference mechanics of a basic encoder-decoder model. If you have further questions, feel free to ask.

One - shotted all 3 parts in one response!



File

Shared files can't be displayed

⌚ Thought for 255 seconds >

Here, I
passed a
screenshot of
Q8.

(a) Implementing the Autoencoder in PyTorch

(i) How many linear layers do you need?

2

The model consists of two distinct linear transformations: the encoder w_1 (from dimension m to k) and the decoder w_2 (from k back to m). In PyTorch, these are represented as two separate `nn.Linear` layers.

(ii) What is the loss function that you will be using?

`nn.MSELoss`

The first term in objective (6) is the mean squared error (MSE) between the original data \mathbf{X} and its reconstruction $W_2 W_1 \mathbf{X}$. PyTorch's `nn.MSELoss` computes exactly this.

(iii) Which of the following would you need to optimize (6) exactly as it is written?

Weight Decay, SGD optimizer

- **Weight Decay:** The terms $\lambda \|W_1\|_F^2 + \lambda \|W_2\|_F^2$ are equivalent to L2 regularization (weight decay) on both weight matrices. In an SGD optimizer, this is implemented via the `weight_decay` parameter.
- **SGD optimizer:** The problem specifies "SGD-style training," making the SGD optimizer a necessary component.
- **Dropout, Layer Norm, Batch Norm:** These are not part of the objective in equation (6) and are not needed.

(b) Inductive Bias from Weight Decay

Yes, using a small nonzero λ introduces an inductive bias towards finding a W_2 matrix with approximately orthonormal columns.

Reasoning:

Consider the singular value decompositions (SVDs) of the optimal weight matrices.

For the product $M = W_2 W_1$ to minimize the reconstruction error, its singular values

s_i should be close to 1 (approaching the top- k projection). The regularization terms $\lambda(\|W_1\|_F^2 + \|W_2\|_F^2)$, for a fixed M , are minimized when the singular values of W_1 and W_2 are equal and balanced, i.e., $a_i = b_i = \sqrt{s_i}$ (this follows from minimizing $a_i^2 + b_i^2$ subject to $a_i b_i = s_i$).

For small λ , the optimal $s_i \approx 1 - \lambda/\lambda_i$ (where λ_i are the data eigenvalues), which is close to 1. Consequently, $a_i \approx b_i \approx \sqrt{s_i} \approx 1$. A matrix with all singular values equal