CS498 Homework 8

April 14, 2026

1 (GROUP SUBMISSION ALLOWED) Problem 1: CircuitSAT to SAT via Tseitin Transform

In Homework 7, you encoded combinatorial problems directly as SAT instances by writing clever constraints in CNF. But there is a more systematic way. Many problems are naturally expressed as Boolean circuits: networks of AND, OR, and NOT gates wired together, just like hardware design. The Circuit Satisfiability problem (CircuitSAT) asks: is there an input assignment that makes the circuit output True? CircuitSAT is NP-complete (it was actually the first problem shown NP-complete, before Cook reduced it to SAT). The beautiful insight is that we can convert any circuit to an equisatisfiable CNF formula using the **Tseitin transform**, and the blowup is only a constant factor per gate. This means building a SAT encoding becomes as easy as wiring up logic gates: you think in terms of circuits, and the Tseitin transform mechanically produces the CNF. Let's implement this.

1.1 Background

Boolean Circuits. A Boolean circuit is a directed acyclic graph (DAG) where: - **Input gates** have no incoming edges and represent input variables. - **Internal gates** compute a Boolean function of their inputs: NOT (one input), AND (two inputs), or OR (two inputs). - One gate is designated as the **output** gate.

A circuit is *satisfiable* if there exists an assignment to the input gates that makes the output gate evaluate to True.

CircuitSAT is NP-complete. Given a Boolean circuit, deciding whether it is satisfiable is NP-complete. This was actually the first problem shown to be NP-complete (the Cook-Levin theorem is often stated for SAT, but the proof goes through CircuitSAT first). Because every NP problem can be reduced to CircuitSAT, and CircuitSAT can be reduced to SAT, this gives us a powerful pipeline: **describe your problem as a circuit, then mechanically convert to CNF.** Also, a lot of functions we write in coding feel closer to CircuitSAT than SAT, so it also provides a recipe of transforming modern coding functions to SAT CNF formulas.

Why circuits? Building a circuit is similar to programming. Instead of writing constraints in CNF (which requires thinking about clauses and is error-prone), you compose AND, OR, and NOT gates to express your logic. For example, the constraint "exactly one of x_1, x_2, x_3 is true" can be built from a small circuit of gates, rather than manually enumerating all the CNF clauses. The Tseitin transform then handles the conversion to CNF automatically.

1.2 The Tseitin Transform

The Tseitin transform converts a Boolean circuit into an equisatisfiable CNF formula. The key idea has three parts:

1. **Introduce a variable for every gate.** Each gate g in the circuit gets a corresponding CNF variable v_g.
2. **Constrain each gate.** For each internal gate, add clauses that force the gate's variable to equal the correct function of its input variables. This is an *if-and-only-if* (biconditional) constraint.
3. **Assert the output.** Add a unit clause forcing the output gate's variable to be True.

The resulting CNF formula is *equisatisfiable* with the original circuit: the circuit is satisfiable if and only if the CNF formula is satisfiable. (It is not logically equivalent because the CNF has extra auxiliary variables, but any satisfying assignment to the circuit extends to a satisfying assignment of the CNF, and vice versa.)

1.2.1 Clause Templates

For each gate type, the biconditional constraint decomposes into a small set of clauses. Let variables be represented as positive integers (DIMACS convention), with negation indicated by a negative sign.

INPUT gate: No clauses needed. The variable is free.

NOT gate: $y \leftrightarrow \neg x$

Implication	Clause (DIMACS)
$y \rightarrow \neg x$	$[\text{-}y, \text{-}x]$
$\neg x \rightarrow y$ (equivalently $\neg y \rightarrow x$)	$[x, y]$

2 clauses per NOT gate. Note $[x, y]$ means a clause of x OR y.

AND gate: $y \leftrightarrow (a \wedge b)$

Implication	Clause (DIMACS)
$y \rightarrow a$	$[\text{-}y, a]$
$y \rightarrow b$	$[\text{-}y, b]$
$(a \wedge b) \rightarrow y$	$[\text{-}a, \text{-}b, y]$

3 clauses per AND gate.

OR gate: $y \leftrightarrow (a \vee b)$

Implication	Clause (DIMACS)
$y \rightarrow (a \vee b)$	$[\text{-}y, a, b]$
$a \rightarrow y$	$[\text{-}a, y]$
$b \rightarrow y$	$[\text{-}b, y]$

3 clauses per OR gate.

Output assertion: $[v_{\text{output}}]$ (a unit clause).

1.3 Worked Example

Consider the circuit for $(x_1 \wedge x_2) \vee (\neg x_3)$:

```
Gate 0: INPUT  []        -> variable 1 (x1)
Gate 1: INPUT  []        -> variable 2 (x2)
Gate 2: INPUT  []        -> variable 3 (x3)
Gate 3: AND    [0, 1]    -> variable 4 (g3 = x1 AND x2)
Gate 4: NOT    [2]       -> variable 5 (g4 = NOT x3)
Gate 5: OR     [3, 4]    -> variable 6 (g5 = g3 OR g4, output)
```

Step 1: Variable mapping. Gate i maps to CNF variable $i + 1$.

Step 2: Clauses for each gate.

Gates 0, 1, 2 are INPUTs – no clauses.

Gate 3 (AND, $y = 4$, inputs $a = 1, b = 2$): $4 \leftrightarrow (1 \wedge 2)$

#	Clause	Meaning
1	$[-4, 1]$	$y \rightarrow a$
2	$[-4, 2]$	$y \rightarrow b$
3	$[-1, -2, 4]$	$(a \wedge b) \rightarrow y$

Gate 4 (NOT, $y = 5$, input $x = 3$): $5 \leftrightarrow \neg 3$

#	Clause	Meaning
4	$[-5, -3]$	$y \rightarrow \neg x$
5	$[3, 5]$	$\neg x \rightarrow y$

Gate 5 (OR, $y = 6$, inputs $a = 4, b = 5$): $6 \leftrightarrow (4 \vee 5)$

#	Clause	Meaning
6	$[-6, 4, 5]$	$y \rightarrow (a \vee b)$
7	$[-4, 6]$	$a \rightarrow y$
8	$[-5, 6]$	$b \rightarrow y$

Step 3: Assert output gate. Gate 5 is the output, variable 6.

#	Clause	Meaning
9	[6]	Output must be True

Total: 6 variables, 9 clauses.

Satisfying assignment: Set $x_1 = \text{False}, x_2 = \text{False}, x_3 = \text{False}$. Then $g_3 = F \wedge F = F$, $g_4 = \neg F = T$, $g_5 = F \vee T = T$. The output is True. The corresponding model is: variable 1 = F, 2 = F, 3 = F, 4 = F, 5 = T, 6 = T.

1.4 Why Is the Blowup Only Constant?

Each gate in the circuit contributes at most 3 clauses (AND and OR gates) or 2 clauses (NOT gates), plus one unit clause for the output. Therefore:

$$\text{Number of clauses} \leq 3 \cdot |\text{gates}| + 1$$

$$\text{Number of variables} = |\text{gates}|$$

This is a **linear** relationship: the CNF formula is only a constant factor larger than the circuit. Compare this to a direct conversion of an arbitrary Boolean formula to CNF, which can cause an exponential blowup (e.g., distributing OR over AND). The Tseitin transform avoids this by introducing auxiliary variables – one per gate – and constraining them locally.

This is why the Tseitin transform is the standard method for converting circuits (or arbitrary Boolean formulas) to CNF in practice.

1.5 What You Must Implement

Submit a file named **`hw8_p1_circuitsat.py`** with the following two functions.

1.5.1 Circuit Representation

A circuit is a list of gate tuples:

```
# (gate_id, gate_type, input_ids)
# gate_type: 'INPUT', 'NOT', 'AND', 'OR'
# input_ids: list of gate_ids feeding into this gate
# The LAST gate in the list is the output gate.
```

1.5.2 Function 1: `tseitin_transform`

```
def tseitin_transform(circuit: list[tuple]) -> tuple[int, list[list[int]]]:
    """
    Convert a Boolean circuit to an equisatisfiable CNF formula using
    the Tseitin transformation.

    Each gate in the circuit gets a CNF variable (1-indexed).
    Gate i maps to variable i+1.

    Clause templates:
      NOT gate (y = NOT x):
        [-y, -x], [x, y]
```

```
    AND gate (y = AND(a, b)):
      [-y, a], [-y, b], [-a, -b, y]

    OR gate (y = OR(a, b)):
      [-y, a, b], [-a, y], [-b, y]

    Output: [output_var]

    Parameters
    ----------
    circuit : list of tuples (gate_id, gate_type, input_ids)
        The Boolean circuit. gate_ids are 0-indexed.
        The LAST gate is the output.

    Returns
    -------
    (num_vars, clauses) where:
        num_vars : int, total number of CNF variables
        clauses : list of list of int, each inner list is a clause
                  in DIMACS convention (positive = True, negative = False).
                  Variables are 1-indexed.
    """
```

1.5.3 Function 2: `solve_circuit`

```
def solve_circuit(circuit: list[tuple]) -> dict:
    """
    Solve a Boolean circuit using Tseitin transform + a SAT solver.

    Uses tseitin_transform to get a CNF, then solves with pysat.

    Returns
    -------
    dict with keys:
        'satisfiable' : bool
        'assignment'  : dict mapping input gate_ids to bool values,
                        or None if unsatisfiable
        'num_vars'    : int, number of CNF variables
        'num_clauses' : int, number of CNF clauses
    """
```

1.6 Hints

- **Variable mapping is simple.** Gate i maps to CNF variable $i+1$ (since DIMACS variables are 1-indexed). So `var = gate_id + 1`.
- **Use `python-sat` (pysat). Download using `pip install python-sat`.** The `pysat` package provides efficient SAT solvers with a simple interface:

```
from pysat.solvers import Glucose3

solver = Glucose3()
for clause in clauses:
    solver.add_clause(clause)
if solver.solve():
    model = solver.get_model()  # list of signed ints, e.g. [1, -2, 3]
```

A positive integer in the model means that variable is True; negative means False.

- **Do not forget the unit clause.** The output gate's variable must be asserted True. This is the single most common mistake.
- **Extracting the input assignment.** After solving, iterate over the circuit to find INPUT gates, look up their variable in the model, and build the assignment dictionary.
- **General AND/OR gates.** The problem specifies 2-input AND and OR gates, which is sufficient for all Boolean functions (any multi-input gate can be decomposed into a tree of 2-input gates).

1.7 Sanity Checks

```
from hw8_p1_circuitsat import tseitin_transform, solve_circuit

# Test 1: Simple NOT gate
circuit = [
    (0, 'INPUT', []),
    (1, 'NOT', [0]),
]
num_vars, clauses = tseitin_transform(circuit)
print(f"NOT gate: {num_vars} vars, {len(clauses)} clauses")
assert num_vars == 2
assert len(clauses) == 3  # 2 for NOT + 1 for output

# Test 2: (x1 AND x2) OR (NOT x3) -- should be satisfiable
circuit = [
    (0, 'INPUT', []),
    (1, 'INPUT', []),
    (2, 'INPUT', []),
    (3, 'AND', [0, 1]),
    (4, 'NOT', [2]),
    (5, 'OR', [3, 4]),
]
result = solve_circuit(circuit)
print(f"Circuit SAT: {result}")
assert result['satisfiable'] == True
assert result['num_vars'] == 6
# Check: plug assignment back in
a = result['assignment']
g3 = a[0] and a[1]
```

```
g4 = not a[2]
g5 = g3 or g4
assert g5 == True, "Assignment should satisfy the circuit"

# Test 3: x AND (NOT x) -- unsatisfiable
circuit = [
    (0, 'INPUT', []),
    (1, 'NOT', [0]),
    (2, 'AND', [0, 1]),
]
result = solve_circuit(circuit)
print(f"UNSAT circuit: {result}")
assert result['satisfiable'] == False
assert result['assignment'] is None

print("All sanity checks passed!")
```

2 (INDIVIDUAL SUBMISSION ONLY) Problem 2: Implementing ResNet from Scratch for CIFAR-100

This problem must be completed individually. You will log in to the GPU cluster and implement and train, from scratch, a neural network based on the ResNet paper.

In 2015, Kaiming He and his colleagues at Microsoft Research posed a disturbing question: why do deeper neural networks perform worse than shallower ones? Adding layers to a plain convolutional network should never hurt, because the extra layers could always learn the identity function and replicate the shallower network's behavior. But in practice, deeper plain networks had higher training error, not just higher test error. This was not overfitting. It was a fundamental optimization problem: deep networks are hard to train because gradient-based methods struggle to learn identity mappings through many nonlinear layers (recall what we discussed on vanishing gradients). Their solution was breathtakingly simple: add shortcut connections that skip one or more layers, so the network learns residual functions F(x) = H(x) - x rather than the full mapping H(x). If the optimal transformation is close to identity, learning F(x) close to zero is far easier than learning H(x) close to x. This idea, called the Residual Network (ResNet), enabled training of networks with over 100 layers and won first place in the 2015 ImageNet competition. In this problem, you will implement ResNet from scratch and train it on CIFAR-100.

2.1 Background: CIFAR-100 and the Depth Problem

The dataset. CIFAR-100 is a standard benchmark in computer vision. It contains 60,000 color images of size 32×32 pixels, split into 50,000 training images and 10,000 test images. Each image belongs to one of 100 fine classes (e.g., apple, aquarium fish, baby, bear, beaver, …). With 100 classes and only 500 training images per class, CIFAR-100 is substantially harder than CIFAR-10.

Why deeper networks? Intuitively, deeper networks should be more powerful. A network with more layers can represent more complex functions. Early successes in deep learning (AlexNet with 8 layers, VGGNet with 19 layers) seemed to confirm this: more depth leads to better performance.

The degradation problem. But experiments told a different story. When researchers trained plain networks (stacks of convolutional layers without skip connections) with 20 versus 56 layers on CIFAR-10, the 56-layer network had *higher training error* than the 20-layer network. This was not overfitting (which would show as higher test error but lower training error). The deeper network was simply harder to optimize. If the 56-layer network could at least learn to copy what the 20-layer network does (with the extra layers learning identity mappings), it should perform no worse. But plain networks struggle to learn identity mappings through chains of nonlinear layers because of the vanishing gradient issue we talked about in class.

The key insight. He et al. proposed a solution: instead of asking each block of layers to learn the desired mapping $H(\mathbf{x})$ directly, ask it to learn the *residual* $F(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x}$. The output of each block is then $F(\mathbf{x}) + \mathbf{x}$. This is implemented by adding a shortcut connection (also called a skip connection) that bypasses the block's layers and adds the input directly to the output. If the optimal transformation is close to identity, the network only needs to push $F(\mathbf{x})$ toward zero, which is much easier than pushing $H(\mathbf{x})$ toward $\mathbf{x}$.

```
+------ [skip connection] ------+
```

```
          |                                 |
          |                                 v
x ----> [conv -> BN -> ReLU -> conv -> BN] -->(+)--> ReLU --> output
          |           F(x)                  |
          +---------------------------------+

Where BN is batch norm (layer normalization), and conv is a convolutional layer.
Output = ReLU( F(x) + x ) where F(x) = BN(conv(Relu(BN(conv(x)))))

This is refered to as a residual block. The skip connection carries x unchanged past
the convolutional layers. The layers only need to learn the
residual F(x) = H(x) - x.
```

Reference. He, K., Zhang, X., Ren, S., & Sun, J. (2016). "Deep Residual Learning for Image Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778. arXiv: https://arxiv.org/abs/1512.03385

2.2 The Residual Learning Framework

The core idea of ResNet is remarkably simple. Consider a stack of layers that is supposed to compute some mapping $H(\mathbf{x})$. Instead of learning $H(\mathbf{x})$ directly, we restructure the layers to learn $F(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x}$, and then compute the output as:

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$$

This is called **residual learning**. The function $F(\mathbf{x})$ is the residual: the difference between what we want and what we already have.

Why does this help? Consider two scenarios:

1. **The optimal mapping is close to identity** (i.e., the layer should mostly pass information through). With residual learning, the network only needs to learn $F(\mathbf{x}) \approx \mathbf{0}$. This is easy: just push all the weights toward zero. Without residual learning, the network must learn $H(\mathbf{x}) \approx \mathbf{x}$ through a chain of nonlinear layers, which is much harder.
2. **The optimal mapping is not identity** (i.e., the layer should transform the input). Residual learning does not hurt: the network can still represent any function $H(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}$ by learning the appropriate F.

In practice, the residual functions have small responses. The learned weights in residual blocks tend to be small, confirming the hypothesis that the identity shortcut provides a good default and the layers only need to make small adjustments.

The skip connection is the mechanism that implements this. It is simply an identity mapping: the input $\mathbf{x}$ is added to the output of the convolutional layers. This requires no additional parameters and adds negligible computation.

2.3 The BasicBlock

The **BasicBlock** is the building block of ResNet-18, ResNet-34, and the CIFAR variants of ResNet. It contains two 3×3 convolutional layers, each followed by batch normalization. ReLU activation

is applied after the first batch norm and after the addition of the skip connection.

Structure:

```
Input x
  |
  +--> Conv2d(3x3, stride) -> BatchNorm2d -> ReLU
  |            |
  |            v
  |    Conv2d(3x3, stride=1) -> BatchNorm2d
  |            |
  |            v
  +-- [skip] --+--> Addition
                       |
                       v
                     ReLU
                       |
                       v
                    Output
```

In detail:

1. **First convolution:** `Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)` followed by `BatchNorm2d(out_channels)` and `ReLU`.
2. **Second convolution:** `Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)` followed by `BatchNorm2d(out_channels)`. Note: no ReLU here yet.
3. **Skip connection:** Add the (possibly transformed) input to the output of step 2.
4. **Final ReLU:** Apply ReLU after the addition.

Why `bias=False`? When a convolutional layer is immediately followed by batch normalization, the bias is redundant. Batch normalization subtracts the mean, which cancels any bias term. Omitting the bias saves parameters.

Handling dimension mismatches. The skip connection adds $\mathbf{x}$ to $F(\mathbf{x})$. For this addition to work, both tensors must have the same shape. When the spatial dimensions change (stride > 1) or the number of channels changes, we cannot use a plain identity skip. Instead, the skip connection applies a 1×1 convolution with the appropriate stride to match the dimensions:

```
self.shortcut = nn.Sequential(
    nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
    nn.BatchNorm2d(out_channels)
)
out = self.F(x) + self.shortcut(x) # Skip layer
```

When `in_channels == out_channels` and `stride == 1`, the skip connection is simply the identity (no transformation needed) so,

```
self.shortcut = nn.Sequential()
out = self.F(x) + self.shortcut(x) # Skip layer
```

2.4 ResNet Architecture for CIFAR

The original paper describes two ResNet variants: one for ImageNet (224×224 images) and one for CIFAR (32×32 images). Since CIFAR images are small, the CIFAR variant uses a simpler initial layer and fewer downsampling steps.

CIFAR ResNet architecture:

Component	Details
Initial conv	`Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)` + `BatchNorm2d(16)` + `ReLU`
Stage 1	n BasicBlocks, 16 channels, 32×32 spatial resolution
Stage 2	n BasicBlocks, 32 channels, 16×16 spatial (first block: stride 2)
Stage 3	n BasicBlocks, 64 channels, 8×8 spatial (first block: stride 2)
Pooling	Global average pooling ($8 \times 8 \to 1 \times 1$)
Classifier	`Linear(64, 100)`

Total layer count. Each BasicBlock has 2 convolutional layers. With n blocks per stage and 3 stages, the total number of convolutional layers is $6n$. Adding the initial conv and the final FC layer gives $6n + 2$ total layers. This defines the standard CIFAR ResNet family:

Model	n (blocks/stage)	Conv layers	Total layers ($6n + 2$)	Approx. params (CIFAR-100)
ResNet-20	3	18	20	~280K
ResNet-32	5	30	32	~470K
ResNet-44	7	42	44	~660K
ResNet-56	9	54	56	~860K
ResNet-110	18	108	110	~1.7M

For this assignment, you will implement and train **ResNet-56** ($n = 9$).

2.5 What You Must Implement

Complete the skeleton file `hw8_p2_resnet.py`. You must implement:

1. **`BasicBlock.__init__` and `BasicBlock.forward`**: The residual block with two 3×3 convolutions and a skip connection. Handle the case where spatial dimensions or channels change.
2. **`ResNet.__init__`, `ResNet._make_stage`, and `ResNet.forward`**: The full ResNet architecture for CIFAR. The `_make_stage` method should create a sequence of n BasicBlocks, where the first block may have stride 2 to downsample.

3. **`train_one_epoch` and `evaluate`**: Training and evaluation functions using cross-entropy loss.
4. **Training loop**: Set up the optimizer, learning rate scheduler, and training loop. Log metrics each epoch and save the model.

Do not modify the model architecture. The autograder loads your saved weights into the same `BasicBlock` and `ResNet` classes with $n = 9$ and `num_classes=100`. Any changes to layer names, sizes, or structure will cause loading to fail.

2.6 Training Setup

Use the following hyperparameters:

Hyperparameter	Value	Reason
Optimizer	SGD with momentum (or AdamW)	SGD with momentum 0.9 is the original paper's choice; AdamW also works well
Learning rate	0.1 (SGD) or 1e-3 (AdamW)	Standard starting LR for each optimizer
Weight decay	1e-4 (SGD) or 0.01 (AdamW)	L2 regularization
Momentum	0.9 (SGD only)	Standard momentum value
LR schedule	Cosine annealing	Smoothly decays LR to near zero
Epochs	200	Enough for convergence
Batch size	128	Standard for CIFAR
Loss function	Cross-entropy	Standard for multi-class classification

Option A: SGD (from the paper).

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1,
                            momentum=0.9, weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
```

Option B: AdamW.

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
```

Either optimizer will reach the accuracy thresholds if you train for 200 epochs with data augmentation.

You are given `get_data_loaders` to get both the training and testing data loaders that loads the data and normalizes it for you.

Expected accuracy trajectory (SGD, ResNet-56): - After 10 epochs: ~45-50% test accuracy - After 50 epochs: ~58-62% test accuracy - After 65 epochs: ~60-64% test accuracy (may appear

to plateau here; this is normal) - After 100 epochs: ~63-68% test accuracy - After 200 epochs: ~70-72% test accuracy

If your test accuracy is below 20% after 10 epochs, check your learning rate, and that skip connections are implemented correctly. If you train on the GPU and your implementation is correct, each epoch should roughly take 10 seconds.

2.7 Running on NCSA Delta

You will train this model on the NCSA Delta supercomputer, which has NVIDIA A100 GPUs. ResNet-56 on CIFAR-100 trains in roughly 30 minutes to 1 hour on an A100.

Step 1: Transfer your script from your computer to Delta:

```
scp hw8_p2_resnet.py <your-ncsa-id>@login.delta.ncsa.illinois.edu:/projects/bgvu/<your-ncsa-id
```

Step 2: SSH into Delta.

```
ssh <your-netid>@login.delta.ncsa.illinois.edu
```

Step 2b: Go into your course directory.

```
cd /projects/bgvu/<your-ncsa-id>
```

Step 2c: Download CIFAR-100 on the login node. Compute nodes may not have internet access, so download the dataset before submitting your job:

```
module load pytorch-conda/2.8
python -c "from torchvision import datasets; datasets.CIFAR100('./data', download=True)"
```

This creates a `./data` directory with the dataset (~178 MB). Your training script uses `download=False` and reads from this directory.

Step 3: Create a SLURM batch script. Create a file in this directory called `train_resnet.sh`:

```
#!/bin/bash
#SBATCH --job-name=resnet56
#SBATCH --account=bgvu-delta-gpu
#SBATCH --partition=gpuA100x4
#SBATCH --gpus-per-node=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
#SBATCH --mem=64G
#SBATCH --time=00:50:00
#SBATCH --output=resnet_%j.out
#SBATCH --error=resnet_%j.err

# Load PyTorch (do NOT module purge, the default modules include CUDA)
module load pytorch-conda/2.8

# Run training
python hw8_p2_resnet.py
```

Important: Do **not** use `module purge` before loading `pytorch-conda`. Make sure `hw8_p2_resnet.py` that you scp'd from your computer is in the current directory.

Step 4: Submit the job. Run:

```
sbatch train_resnet.sh
```

This will tell you `Submitted job <jobid>`. Keep track of this jobid.

Step 5: Monitor the job.

```
squeue -u $USER              # Check job status, this will tell you if the job started or not.
scontrol show job <jobid>    # If your job hasn't started, this will have "StartTime=2026-04-1
                             # which is the estimated time that the scheduler thinks your job w
cat resnet_<jobid>.out       # View output (while running or after)
```

Step 6: Copy results back to your local machine. After your job terminates, you will have both `resnet_model.pt` and `trainin_log.json` in your directory. You want to download them back to your (local) computer. To do this, from your own computer terminal (NOT from ncsa machine), run:

```
scp <your-ncsa-id>@login.delta.ncsa.illinois.edu:/projects/bgvu/<your-ncsa-id>/resnet_model.pt
scp <your-ncsa-id>@login.delta.ncsa.illinois.edu:/projects/bgvu/<your-ncsa-id>/training_log.js
```

Troubleshooting: - Start by requesting less time (say 3-4 minutes `#SBATCH --time=00:03:00`) to debug and make sure the training is working at first. - If you see `ModuleNotFoundError: No module named 'torchvision'`, make sure you loaded `pytorch-conda`. - If your job is pending for a long time, check the queue: `squeue -p gpuA100x4`. Jobs usually start within a few minutes. You can see the *estimated* time of start of your job using `scontrol show job <jobid>` and looking at `StartTime` attribute. - Don't forget to download CIFAR-100 dataset first in `/projects/bgvu/<your-ncsa-id>` using `module load pytorch-conda/2.8 && python -c "from torchvision import datasets; datasets.CIFAR100('./data', download=True)"`.

2.8 What You Must Submit

Submit the following three files to Gradescope:

1. **`hw8_p2_resnet.py`** – Your completed training script with the ResNet implementation.
2. **`resnet_model.pt`** – Your trained model, saved with:

   ```
   torch.save(model, 'resnet_model.pt')
   ```

   This saves the entire model (architecture + weights). The autograder loads it with `torch.load()` and evaluates on the CIFAR-100 test set. This means your layer names do not need to match ours exactly.
3. **`training_log.json`** – A JSON file containing a list of dictionaries, one per epoch. Each dictionary must have the keys:

   ```
   {
       "epoch": 1,
       "train_loss": 3.85,
       "train_acc": 0.08,
       "test_loss": 3.72,
   ```

```
    "test_acc": 0.10
}
```

There should be at least 150 entries (one per epoch). Accuracies should be fractions (0.68 means 68%).

2.9 Hints and Sanity Checks

1. **Parameter count sanity check.** ResNet-56 for CIFAR-100 should have approximately 861,620 parameters. Verify with:

   ```
   print(f"Parameters: {sum(p.numel() for p in model.parameters()):,}")
   ```

   If your count is dramatically different, check your channel sizes ([16, 32, 64]) and the number of blocks per stage (9 for ResNet-56).

2. **Skip connections are critical.** Without skip connections, your ResNet is just a plain convolutional network and will be much harder to optimize. Common mistakes:

   - Forgetting to add the shortcut when `in_channels == out_channels` (identity case)
   - Applying ReLU before the addition instead of after
   - Not using stride in the shortcut's 1x1 conv when the spatial size changes

3. **BatchNorm placement.** The correct order in a BasicBlock is:

   - First branch: Conv -> BN -> ReLU -> Conv -> BN
   - Skip branch: (identity) or (Conv1x1 -> BN)
   - Add the two branches, then apply ReLU
   - The ReLU comes *after* the addition, not before it.

4. **Use `bias=False` in conv layers.** When a conv is followed by BatchNorm, the bias is redundant (BatchNorm subtracts the mean). Omit it with `bias=False`.

5. **Global average pooling.** After the last stage, use `F.adaptive_avg_pool2d(x, 1)` or `F.avg_pool2d(x, x.size(3))` to reduce the spatial dimensions from 8×8 to 1×1. Then flatten before the FC layer. You can flatten using `out = out.view(out.size(0), -1)` for example.

6. **Weight initialization.** PyTorch's default initialization works reasonably well. For potentially better results, use Kaiming initialization for conv layers:

   ```
   for m in self.modules():
       if isinstance(m, nn.Conv2d):
           nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
       elif isinstance(m, nn.BatchNorm2d):
           nn.init.constant_(m.weight, 1)
           nn.init.constant_(m.bias, 0)
   ```

7. **Training time.** On an A100, expect roughly **9-12 seconds per epoch** for ResNet-56, for a total of about **30-40 minutes** for 200 epochs. If an epoch takes more than 30 seconds, something is wrong: check that you are on a GPU (not CPU), that `num_workers` in the DataLoader matches `--cpus-per-task` in your SLURM script, and that you did not use `module purge` (which removes the CUDA toolkit).

8. **Move everything to GPU.** Both the model and each batch of data must be on the same device:

```
model = ResNet(n=9, num_classes=100).to(device)
# In training loop:
images, labels = images.to(device), labels.to(device)
```

9. **Saving the model.** Save the entire model (not just state_dict) so the autograder can load any architecture:

```
torch.save(model, 'resnet_model.pt')
```

10. These torch.nn layers will be important: `nn.Conv2d, nn.BatchNorm2d,`

11. **Training log format.** Make sure accuracies are fractions (e.g., 0.68), not percentages (e.g., 68). The autograder checks for this.

2.10 Grading

Criterion	Points
Model architecture correct (loads into ResNet with n=9, correct param count)	10
BasicBlock has skip connection (forward pass test)	10
ResNet-56 forward pass produces correct output shape: [1, 100] for [1, 3, 32, 32] input	10
Test accuracy $> 50\%$	15
Test accuracy $> 60\%$	20
Test accuracy $> 68\%$	20
Training log submitted with 150+ epochs and required keys	15
Total	**100**

The accuracy thresholds are cumulative: if you achieve $> 68\%$, you automatically get the points for $> 60\%$ and $> 50\%$ as well.

3 (GROUP SUBMISSION ALLOWED) Problem 3: Bias-Variance Tradeoff and Double Descent

In Lecture 21, we introduced the bias-variance tradeoff: simple models underfit (high bias), complex models overfit (high variance), and the sweet spot is somewhere in between. This classical picture suggests that making a model more complex always eventually hurts generalization. But modern deep learning contradicts this: models with billions of parameters (and more recently trillions), far more than the number of training examples, often generalize nicely. In this problem, you will first prove the bias-variance decomposition rigorously, then run an experiment that reveals the double descent phenomenon, where test error first increases then decreases again as model complexity grows past the interpolation threshold.

3.1 Part A: Prove the Bias-Variance Decomposition (50 points)

Consider a regression setting where we observe data from the model $y = f(x) + \varepsilon$, where f is the true function and ε is noise with $\mathbb{E}[\varepsilon] = 0$ and $\text{Var}(\varepsilon) = \sigma^2$. The noise ε is independent of x.

Given a training set $\mathcal{D}$ sampled from our dataset, we learn a predictor $\hat{f}_{\mathcal{D}}(x)$. Different training sets produce different predictors. We want to understand the expected prediction error at a fixed test point x:

$$\mathbb{E}_{\mathcal{D},\varepsilon}\left[(y - \hat{f}_{\mathcal{D}}(x))^2\right]$$

where the expectation is over both the random training set $\mathcal{D}$ and the test noise ε.

3.1.1 A1. (25 points) Prove that the expected squared error decomposes as:

$$\mathbb{E}_{\mathcal{D},\varepsilon}\left[(y - \hat{f}_{\mathcal{D}}(x))^2\right] = \underbrace{\left(f(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(x)]\right)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\left[(\hat{f}_{\mathcal{D}}(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(x)])^2\right]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible noise}}$$

Hints: - Start by substituting $y = f(x) + \varepsilon$ and expanding the square. - Add and subtract $\mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(x)]$ inside the squared term to separate the bias and variance. - Use the fact that ε is independent of $\mathcal{D}$ and $\hat{f}_{\mathcal{D}}(x)$. - Use $\mathbb{E}[\varepsilon] = 0$ to eliminate cross-terms involving ε.

3.1.2 A2. (10 points) Interpret each term:

For each of the three terms (Bias^2, Variance, σ^2), give a one-sentence explanation of what it measures and what causes it to be large. Give a concrete example of a model that has high bias and one that has high variance (you can use the polynomial fitting example from Lecture 21).

3.1.3 A3. (15 points) The tradeoff:

Explain, in 3-5 sentences, why Bias^2 and Variance typically move in opposite directions as model complexity increases. Specifically: why does a more complex model class (e.g. higher-degree polynomial, wider neural network) tend to have lower bias but higher variance? Your answer should connect to the number of parameters, the flexibility of the function class, and the dependence on the specific training set.

3.2 Part B: Double Descent Experiment (50 points)

The classical bias-variance picture predicts a U-shaped test error curve. But Belkin et al. (2019) showed that for overparameterized models, test error can decrease again beyond the interpolation threshold, creating a "double descent" curve.

3.2.1 The setup

You will use **random ReLU features** with a **linear classifier solved in closed form** (no gradient descent needed). The idea:

1. **Generate d random feature vectors** $w_1, \ldots, w_d \in \mathbb{R}^{784}$ (sampled once from $\mathcal{N}(0, I/784)$).
2. **Transform each input** $x \in \mathbb{R}^{784}$ into a d-dimensional feature vector: $\phi(x) = [\text{ReLU}(w_1^\top x), \ldots, \text{ReLU}(w_d^\top x)]$.
3. **Solve for the optimal linear classifier** on the transformed features using least squares: $\hat{W} = \arg\min_W \|\Phi W - Y\|^2$, where Φ is the matrix of transformed training inputs and Y is the one-hot label matrix.

The key: `numpy.linalg.lstsq` gives the **minimum-norm** solution. When $d < n$ (fewer features than samples), the system is overdetermined and lstsq finds the best fit. When $d > n$, the system is underdetermined and lstsq finds the solution with the smallest weight norm. When $d \approx n$, the system is exactly determined and is forced to fit every training point exactly, including noisy ones, leading to poor generalization.

3.2.2 What to do

Use $n = 4{,}000$ training samples from MNIST (randomly subsampled), the full 10K test set, and sweep d (number of random features) over values from 50 to 20,000, with dense sampling around $d = n = 4{,}000$.

For each d: 1. Generate a random matrix $W \in \mathbb{R}^{d \times 784}$. 2. Compute $\Phi_{\text{train}} = \text{ReLU}(X_{\text{train}} W^\top)$ and $\Phi_{\text{test}} = \text{ReLU}(X_{\text{test}} W^\top)$. 3. Solve $\hat{W} =$ `np.linalg.lstsq`$(\Phi_{\text{train}}, Y_{\text{train}})$. 4. Predict: $\hat{y} = \arg\max(\Phi_{\text{test}} \hat{W})$. 5. Record train accuracy, test accuracy, and $\|\hat{W}\|$.

3.2.3 Starter code

We provide `hw8_p3_double_descent.py` with data loading, the experiment loop, and plotting already written. You only need to fill in two functions: `random_relu_features` and `solve_and_evaluate`.

3.2.4 Your plot should have

- X-axis: number of random features d (log scale)
- Y-axis: test error (1 - test accuracy)
- A vertical dashed line at $d = n = 4{,}000$ (interpolation threshold)
- You should also plot the weight norm $\|\hat{W}\|$ (which should spike at the threshold)

3.2.5 In your discussion, address

- What happens to test error as d increases from small to $d \approx n$?
- What happens as d grows far beyond n?

- How does the weight norm behave near the threshold, and why (remember what Lasso/Ridge regression did? Do you see why regularization is important?)?
- How does this relate to the bias-variance decomposition you proved in Part A?

3.3 Submission

Submit a single `writeup.pdf` to Gradescope containing: - **Part A:** your proof (A1, show all steps), interpretations (A2), and tradeoff explanation (A3) - **Part B:** your double descent plot(s), the code you used to generate them, and a 3-5 sentence discussion

Show your work. For Part A, write out the full derivation step by step. For Part B, include your code (inline or appendix) so we can verify you ran the experiment.

This problem is graded manually (no autograder).

3.4 Grading

Criterion	Points
A1: Correct proof of bias-variance decomposition	25
A2: Correct interpretation of each term with examples	10
A3: Clear explanation of the tradeoff	15
B: Plot shows double descent shape with labeled axes and threshold	25
B: Discussion correctly interprets the results and connects to Part A	25
Total	**100**