Learning Dynamics and Representation in Convolutional Neural Networks (CNNs) using CIFAR-10

Course: Deep Learning and Pattern Classification
Institution: IIIT Vadodara

Team: Neural Ninjas

- **Shubham Gupta** — 202251126
- **Sourabh Patil** — 202251095
- **Neel Madhav Padhi** — 202251074
- **Tanishka Sharma** — 202251140

Project Overview

This notebook implements and analyzes the learning dynamics and internal representations of **Convolutional Neural Networks (CNNs)** trained on a subset of the **CIFAR-10 dataset**.
The project is divided into five key parts, exploring both theoretical and experimental aspects:

1. **Core Implementation:**
   Building and training a CNN from scratch, verifying gradients, and visualizing filters.
2. **Optimization Dynamics:**
   Studying how learning rate, optimizer choice, and network depth affect convergence and stability.
3. **Visualizing Representations:**
   Exploring feature evolution using PCA/t-SNE and layer-wise activation analysis.
4. **Loss Landscape Exploration:**
   Examining CNN loss surface geometry, curvature, and flatness for generalization insights.
5. **Regularization and Generalization:**
   Comparing effects of dropout, weight decay, and data augmentation on performance and loss sharpness.

Key Deliverables

- Correct CNN implementation (forward & backward passes verified)
- Gradient analysis (analytical vs numerical)
- Training and validation logs with visualization
- Feature representation plots (filters, PCA/t-SNE)
- Loss landscape and Hessian curvature visualizations
- Regularization and generalization study

Tools & Frameworks

- **Language:** Python 3
- **Framework:** PyTorch
- **Libraries:** NumPy, Matplotlib, tqdm, scikit-learn

Setup, imports, and configuration

```
import os
import math
import random
import time
from pathlib import Path


import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm


import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Subset, random_split
from torchvision import datasets, transforms, utils


from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.metrics.pairwise import cosine_similarity


# Set random seeds for reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)


# Use GPU if available
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Device:', device)
```

```
Device: cuda
```

Configuration

```
CONFIG = {
'batch_size': 128,
'val_batch_size': 256,
'num_workers': 2,
'num_epochs': 30,
'subset_train_size': 10000, # reduced training subset
'lr': 0.01,
'momentum': 0.9,
'weight_decay': 0.0,
'optimizer': 'SGD', # 'SGD', 'SGD_mom', 'Adam'
'model_depth': 4, # choose from 2,4,8
'dropout': 0.0,
'augment': True,
}


# Create results dir
RESULTS_DIR = Path('results')
RESULTS_DIR.mkdir(exist_ok=True)
```

```
# Data preparation (CIFAR-10) with optional augmentation and subset
```

```
mean = (0.4914, 0.4822, 0.4465)
std = (0.2470, 0.2435, 0.2616)


train_transforms = [transforms.RandomCrop(32, padding=4), transforms.RandomH


train_transform = transforms.Compose(train_transforms)
val_transform = transforms.Compose([transforms.ToTensor(), transforms.Normal
```

```
# Download CIFAR-10
train_dataset_full = datasets.CIFAR10(root='data', train=True, download=True
val_dataset_full = datasets.CIFAR10(root='data', train=False, download=True,
```

```
100%|██████████| 170M/170M [00:03<00:00, 47.5MB/s]
```

```
# Use subset for faster experiments
subset_size = min(CONFIG['subset_train_size'], len(train_dataset_full))
train_subset, _ = random_split(train_dataset_full, [subset_size, len(train_d
```

```
train_loader = DataLoader(train_subset, batch_size=CONFIG['batch_size'], shu
val_loader = DataLoader(val_dataset_full, batch_size=CONFIG['val_batch_size'


print('Train subset size:', len(train_subset), 'Val size:', len(val_dataset_
```

```
Train subset size: 10000 Val size: 10000
```

Model: parameterized CNN with variable depth keeping parameter count roughly constant

```
class SimpleCNN(nn.Module):
    def __init__(self, num_layers=4, base_channels=32, num_classes=10, dropc
        super().__init__()
        layers = []
        in_ch = 3
        channels = base_channels
        for l in range(num_layers):
            conv = nn.Conv2d(in_ch, channels, kernel_size=3, padding=1, bias
            layers += [conv, nn.BatchNorm2d(channels), nn.ReLU(inplace=True)
            if (l % 2) == 1:
                layers += [nn.MaxPool2d(2)]
            if dropout > 0:
                layers += [nn.Dropout(dropout)]
            in_ch = channels
            # double channels slightly every two layers to keep capacity
            if l % 2 == 1:
                channels = min(channels * 2, 512)

        self.features = nn.Sequential(*layers)
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(in_ch, 256),
            nn.ReLU(inplace=True),
            nn.Dropout(dropout),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

```
# Utility to instantiate models with roughly similar parameter counts by adj
def make_model(depth=4, dropout=0.0, target_params=None):
    # Heuristic: choose base_channels so parameter count ~ target
    base = 32
```

```
    model = SimpleCNN(num_layers=depth, base_channels=base, dropout=dropout)
    if target_params is None:
        return model
    # scale base to hit target roughly
    for base_try in [16,24,32,48,64,96]:
        model_try = SimpleCNN(num_layers=depth, base_channels=base_try, drop
        n = sum(p.numel() for p in model_try.parameters())
        if n >= target_params:
            return model_try
    return model
```

```
# Quick test
m = make_model(depth=CONFIG['model_depth'], dropout=CONFIG['dropout'])
print('Model params:', sum(p.numel() for p in m.parameters()))
```

```
Model params: 85162
```

Training utilities: train loop, evaluate, gradient norm monitor

```
from collections import defaultdict


def get_optimizer(model, cfg):
    if cfg['optimizer'] == 'SGD':
        return optim.SGD(model.parameters(), lr=cfg['lr'], weight_decay=cfg[
    elif cfg['optimizer'] == 'SGD_mom':
        return optim.SGD(model.parameters(), lr=cfg['lr'], momentum=cfg['mom
    elif cfg['optimizer'] == 'Adam':
        return optim.Adam(model.parameters(), lr=cfg['lr'], weight_decay=cfg
    else:
        raise ValueError('Unknown optimizer')
```

```
def compute_gradient_norm(model):
    total_norm = 0.0
    for p in model.parameters():
        if p.grad is not None:
            param_norm = p.grad.data.norm(2)
            total_norm += param_norm.item() ** 2
    return math.sqrt(total_norm)
```

```
def train_one_epoch(model, loader, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    grad_norms = []
    for x,y in loader:
```

```
        x = x.to(device)
        y = y.to(device)
        optimizer.zero_grad()
        logits = model(x)
        loss = F.cross_entropy(logits, y)
        loss.backward()
        gn = compute_gradient_norm(model)
        grad_norms.append(gn)
        optimizer.step()

        running_loss += loss.item() * x.size(0)
        preds = logits.argmax(dim=1)
        correct += (preds == y).sum().item()
        total += x.size(0)
    return running_loss/total, correct/total, np.mean(grad_norms)
```

```
def evaluate(model, loader, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for x,y in loader:
            x = x.to(device)
            y = y.to(device)
            logits = model(x)
            loss = F.cross_entropy(logits, y)
            running_loss += loss.item() * x.size(0)
            preds = logits.argmax(dim=1)
            correct += (preds == y).sum().item()
            total += x.size(0)
    return running_loss/total, correct/total
```

Gradient check (finite differences) for a convolutional layer

We'll implement a numeric gradient check for a single conv layer parameter (weight) comparing

analytical gradient from backprop vs finite difference estimate.

```
def numeric_grad_check_conv(model, sample_input, sample_target, eps=1e-4, pa
    # Find parameter
    param = None
    for name, p in model.named_parameters():
        if param_name_substr in name:
            param = p
            p_name = name
            break
    if param is None:
        raise ValueError('param not found')

    model.zero_grad()
    logits = model(sample_input)
    loss = F.cross_entropy(logits, sample_target)
    loss.backward()
    analytic = param.grad.detach().cpu().numpy().copy()

    # numeric estimate for a small slice: choose a few indices to estimate
    numeric = np.zeros_like(analytic)
    it = np.nditer(analytic, flags=['multi_index'], op_flags=['readwrite'])
    # sample up to 10 indices
    indices = []
    rng = np.random.default_rng(SEED)
    flat_size = analytic.size
    picks = rng.choice(flat_size, size=min(10, flat_size), replace=False)
    for flat_idx in picks:
        multi = np.unravel_index(flat_idx, analytic.shape)
        indices.append(multi)

    for multi in indices:
        # plus
        orig = param.data[multi].item()
        param.data[multi] = orig + eps
        lp = F.cross_entropy(model(sample_input), sample_target).item()
        param.data[multi] = orig - eps
        lm = F.cross_entropy(model(sample_input), sample_target).item()
        param.data[multi] = orig
        numeric[multi] = (lp - lm) / (2*eps)

    # compute relative error
    rel_errors = []
    for multi in indices:
        a = analytic[multi]
        n = numeric[multi]
        denom = max(1e-8, abs(a)+abs(n))
        rel_errors.append(abs(a-n)/denom)
    return indices, analytic, numeric, rel_errors
```

Training runs: function to run experiments and save logs

```
def run_experiment(cfg, exp_name='exp'):
    logs = defaultdict(list)
    model = make_model(depth=cfg['model_depth'], dropout=cfg.get('dropout',6
    optimizer = get_optimizer(model, cfg)
    print('Experiment', exp_name, 'Model params:', sum(p.numel() for p in mc

    for epoch in range(cfg['num_epochs']):
        t0 = time.time()
        train_loss, train_acc, train_gn = train_one_epoch(model, train_loade
        val_loss, val_acc = evaluate(model, val_loader, device)
        t1 = time.time()
        logs['epoch'].append(epoch)
        logs['train_loss'].append(train_loss)
        logs['train_acc'].append(train_acc)
        logs['train_grad_norm'].append(train_gn)
        logs['val_loss'].append(val_loss)
        logs['val_acc'].append(val_acc)
        print(f"Epoch {epoch:02d}  train_loss={train_loss:.4f} train_acc={tr

    # Save model and logs
    torch.save(model.state_dict(), RESULTS_DIR / f'{exp_name}_model.pth')
    np.savez(RESULTS_DIR / f'{exp_name}_logs.npz', **{k:np.array(v) for k,v
    return model, logs
```

7. Visualization utilities: filters, activations, PCA/t-SNE, RSM

```
def show_filters(conv_layer, ncols=8, title='filters'):
    w = conv_layer.weight.detach().cpu().clone()
    # Normalize to [0,1] per filter for visualization
    w_min, w_max = w.min(), w.max()
    w = (w - w_min) / (w_max - w_min + 1e-8)
    n_filters = w.shape[0]
    n_display = min(n_filters, 32)
    grid = utils.make_grid(w[:n_display], nrow=ncols, padding=1)
    plt.figure(figsize=(8,8))
    plt.imshow(grid.permute(1,2,0).numpy())
    plt.axis('off')
    plt.title(title)
    plt.show()
```

```
def get_activations(model, loader, layer_name='features.0'):
    # Return activations for a single batch
    model.eval()
    activations = None
    hooks = []
    def hook_fn(module, input, output):
        nonlocal activations
        activations = output.detach().cpu()
    # find layer
```

```
    target = dict(model.named_modules()).get(layer_name, None)
    if target is None:
        raise ValueError('layer not found')
    h = target.register_forward_hook(hook_fn)
    with torch.no_grad():
        x,y = next(iter(loader))
        _ = model(x.to(device))
    h.remove()
    return x, y, activations
```

```
def plot_pca_tsne(features, labels, method='pca', ncomp=2, sample=2000):
    # features: [N, D]
    N = features.shape[0]
    idx = np.random.choice(N, min(N, sample), replace=False)
    X = features[idx]
    y = labels[idx]
    if method == 'pca':
        p = PCA(n_components=ncomp)
        Z = p.fit_transform(X)
    else:
        ts = TSNE(n_components=2, init='pca', learning_rate='auto')
        Z = ts.fit_transform(X)
    plt.figure(figsize=(6,6))
    for c in np.unique(y):
        sel = y==c
        plt.scatter(Z[sel,0], Z[sel,1], label=str(c), s=8)
    plt.legend(bbox_to_anchor=(1.05,1), loc='upper left')
    plt.title(f'{method.upper()} projection of features')
    plt.show()
```

```
def compute_rsm(features):
    # features: [N, D]
    sims = cosine_similarity(features)
    return sims
```

Hessian top eigenvalue estimate via power iteration using Hessian-vector product

```
def hessian_vector_product(loss, model, v):
    # v should be a list of tensors matching model.parameters()
    grads = torch.autograd.grad(loss, model.parameters(), create_graph=True)
    flat_grads = torch.cat([g.contiguous().view(-1) for g in grads])
    flat_v = torch.cat([vv.contiguous().view(-1) for vv in v])
    grad_v = (flat_grads * flat_v).sum()
    Hv = torch.autograd.grad(grad_v, model.parameters(), retain_graph=True)
    return [h.detach() for h in Hv]
```

```
def flatten_tensors(tensors):
    return torch.cat([t.contiguous().view(-1) for t in tensors])
```

```
def unflatten_like(flat, template_params):
    params = []
    offset = 0
    for p in template_params:
        num = p.numel()
        params.append(flat[offset:offset+num].view_as(p).detach())
        offset += num
    return params
```

```
def estimate_top_hessian_eigenvalue(model, data_batch, target_batch, iters=2
    model.zero_grad()
    model.train()
    outputs = model(data_batch)
    loss = F.cross_entropy(outputs, target_batch)
    # initialize v with random vector
    params = [p for p in model.parameters() if p.requires_grad]
    flat_size = sum(p.numel() for p in params)
    v = torch.randn(flat_size, device=loss.device)
    v = v / v.norm()
    for i in range(iters):
        v_list = unflatten_like(v, params)
        Hv_list = hessian_vector_product(loss, model, v_list)
        Hv = flatten_tensors(Hv_list)
        eigen = torch.dot(v, Hv).item()
        Hv_norm = Hv.norm()
        v = Hv / (Hv_norm + 1e-12)
    return eigen
```

Loss landscape: compute loss on a grid along two random directions in parameter space

```
def sample_random_directions(model):
    params = [p for p in model.parameters() if p.requires_grad]
    flat_size = sum(p.numel() for p in params)
    d1 = torch.randn(flat_size, device=next(model.parameters()).device)
    d2 = torch.randn(flat_size, device=next(model.parameters()).device)
    # normalize
    d1 = d1 / d1.norm()
    d2 = d2 - d1 * torch.dot(d1, d2)
    d2 = d2 / d2.norm()
    return d1, d2
```

```
def get_flat_params(model):
    return flatten_tensors([p.data for p in model.parameters() if p.requires
```

```
def set_flat_params(model, flat):
    params = [p for p in model.parameters() if p.requires_grad]
    offset = 0
    for p in params:
        num = p.numel()
        p.data.copy_(flat[offset:offset+num].view_as(p))
        offset += num
```

```
def loss_on_grid(model, d1, d2, base_params, grid_x, grid_y, data_batch, tar
    losses = np.zeros((len(grid_x), len(grid_y)))
    base = base_params.to(device)
    for i, a in enumerate(grid_x):
        for j, b in enumerate(grid_y):
            vec = base + a * d1 + b * d2
            set_flat_params(model, vec)
            out = model(data_batch)
            losses[i,j] = F.cross_entropy(out, target_batch).item()
    # restore base
    set_flat_params(model, base)
    return losses
```

Putting it all together: run a full experiment and produce the requested plots and analyses

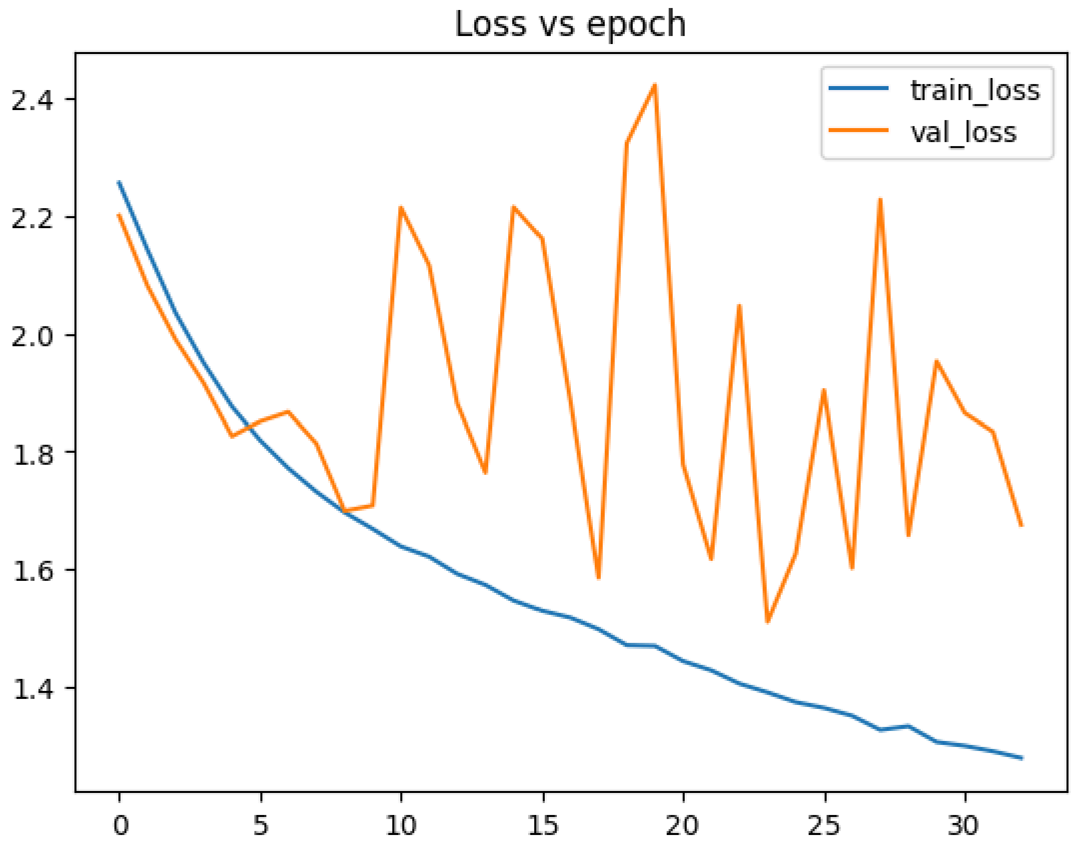
```
def full_run_and_analysis(cfg, name='run'):
    model, logs = run_experiment(cfg, exp_name=name)

    # Save and plot training logs
    epochs = np.array(logs['epoch'])
    plt.figure(); plt.plot(epochs, logs['train_loss'], label='train_loss');
    plt.figure(); plt.plot(epochs, logs['train_acc'], label='train_acc'); pl
    plt.figure(); plt.plot(epochs, logs['train_grad_norm'], label='train_gra

    # Show first conv filters before/after: (we saved after training only) -
    first_conv = dict(model.named_modules())['features.0']
    show_filters(first_conv, title='Conv1 filters (trained)')

    # Gradient check on a small batch
    xb, yb = next(iter(train_loader))
    xb_small = xb[:16].to(device)
    yb_small = yb[:16].to(device)
    inds, analytic, numeric, rel_errs = numeric_grad_check_conv(model, xb_sm
    print('Gradient check indices:', inds)
    print('Relative errors:', rel_errs)

    # Activations and PCA/t-SNE for conv1 and conv2 if exists
    try:
        x_batch, y_batch, act1 = get_activations(model, train_loader, layer_
        # act1 shape [B, C, H, W] -> flatten spatial
```

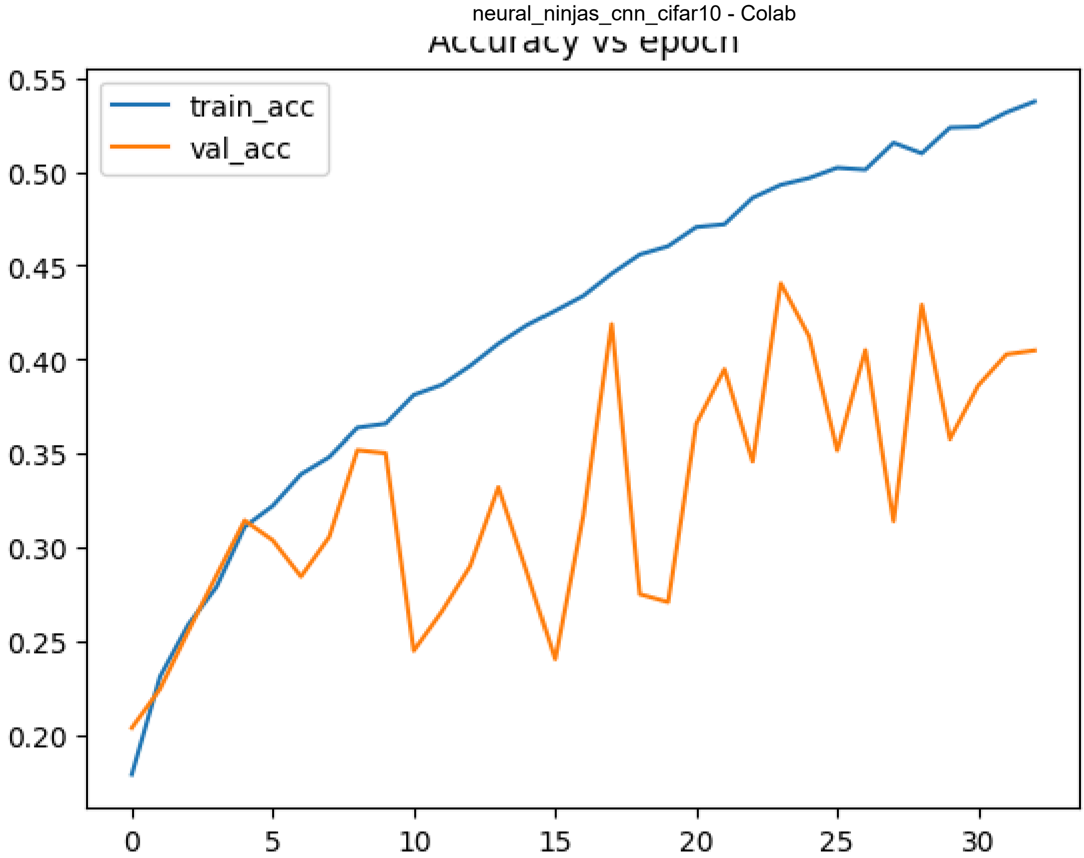
```
            feat1 = act1.view(act1.size(0), -1).numpy()
            labels = y_batch.numpy()
            plot_pca_tsne(feat1, labels, method='pca')
            plot_pca_tsne(feat1, labels, method='tsne')
        except Exception as e:
            print('Activations error', e)

        # RSM for a deeper layer: pick last conv in features
        last_layer = None
        for name, module in model.named_modules():
            if isinstance(module, nn.Conv2d):
                last_layer = name
        if last_layer is not None:
            x_batch, y_batch, actL = get_activations(model, train_loader, layer_
            featL = actL.view(actL.size(0), -1).numpy()
            rsm = compute_rsm(featL[:200])
            plt.figure(figsize=(6,6)); plt.imshow(rsm); plt.title('RSM (last con

        # Hessian top eigenvalue estimate on a small batch
        xb2, yb2 = next(iter(train_loader))
        xb2 = xb2[:64].to(device); yb2 = yb2[:64].to(device)
        try:
            top_eig = estimate_top_hessian_eigenvalue(model, xb2, yb2, iters=10)
            print('Estimated top Hessian eigenvalue (approx):', top_eig)
        except Exception as e:
            print('Hessian estimate failed:', e)

        # Loss landscape around current params
        flat_base = get_flat_params(model).detach()
        d1, d2 = sample_random_directions(model)
        grid = np.linspace(-1.0, 1.0, 21)
        losses = loss_on_grid(model, d1, d2, flat_base, grid, grid, xb2, yb2)
        plt.figure(figsize=(6,5));
        cs = plt.contourf(grid, grid, losses.T, levels=30); plt.colorbar(cs); pl

        return model, logs
```

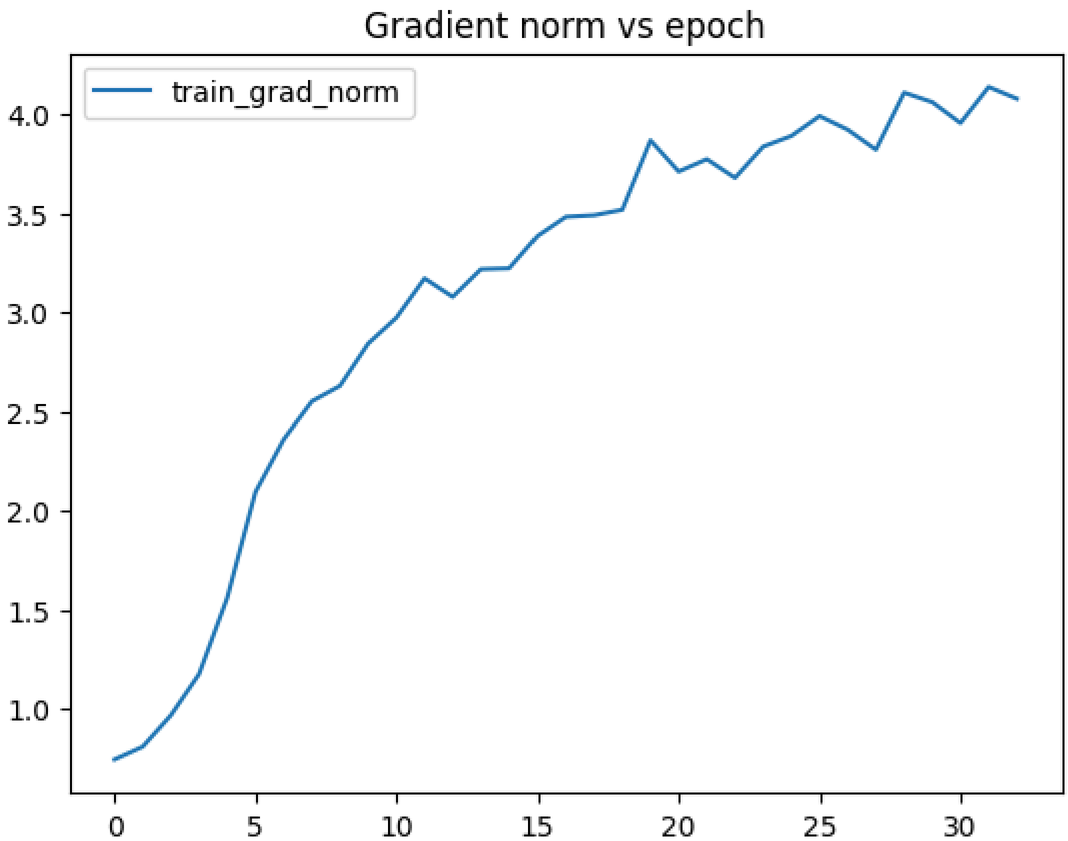
Run the main experiment

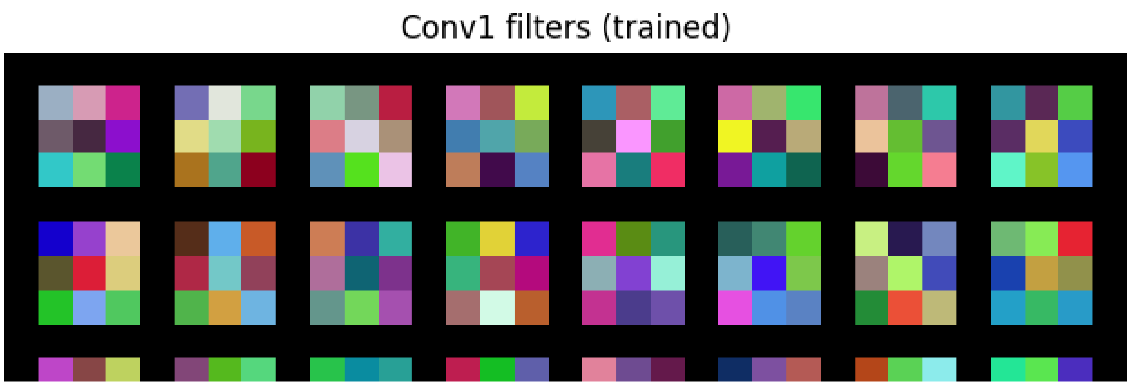
```
    if __name__ == '__main__':
        cfg = CONFIG.copy()
        cfg['num_epochs'] = 33
        model, logs = full_run_and_analysis(cfg, name='default_run')
```

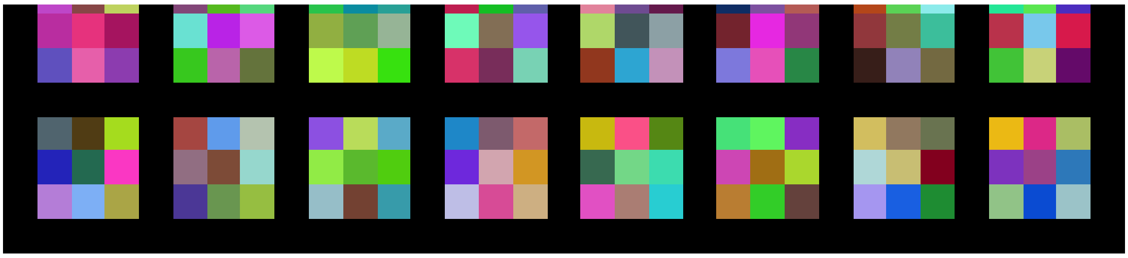
```
Experiment default_run Model params: 85162
Epoch 00  train_loss=2.2558 train_acc=0.1792 train_gn=7.4424e-01  val_loss=:
Epoch 01  train_loss=2.1426 train_acc=0.2313 train_gn=8.0878e-01  val_loss=:
Epoch 02  train_loss=2.0350 train_acc=0.2590 train_gn=9.6749e-01  val_loss=:
Epoch 03  train_loss=1.9499 train_acc=0.2790 train_gn=1.1754e+00  val_loss=:
Epoch 04  train_loss=1.8762 train_acc=0.3111 train_gn=1.5608e+00  val_loss=:
Epoch 05  train_loss=1.8189 train_acc=0.3223 train_gn=2.0942e+00  val_loss=:
Epoch 06  train_loss=1.7717 train_acc=0.3391 train_gn=2.3590e+00  val_loss=:
Epoch 07  train_loss=1.7315 train_acc=0.3482 train_gn=2.5508e+00  val_loss=:
Epoch 08  train_loss=1.6966 train_acc=0.3637 train_gn=2.6283e+00  val_loss=:
Epoch 09  train_loss=1.6684 train_acc=0.3658 train_gn=2.8418e+00  val_loss=:
Epoch 10  train_loss=1.6386 train_acc=0.3811 train_gn=2.9724e+00  val_loss=:
Epoch 11  train_loss=1.6213 train_acc=0.3865 train_gn=3.1709e+00  val_loss=:
Epoch 12  train_loss=1.5920 train_acc=0.3966 train_gn=3.0778e+00  val_loss=:
Epoch 13  train_loss=1.5732 train_acc=0.4084 train_gn=3.2174e+00  val_loss=:
Epoch 14  train_loss=1.5467 train_acc=0.4182 train_gn=3.2218e+00  val_loss=:
Epoch 15  train_loss=1.5294 train_acc=0.4259 train_gn=3.3854e+00  val_loss=:
Epoch 16  train_loss=1.5176 train_acc=0.4339 train_gn=3.4819e+00  val_loss=:
Epoch 17  train_loss=1.4979 train_acc=0.4457 train_gn=3.4898e+00  val_loss=:
Epoch 18  train_loss=1.4708 train_acc=0.4559 train_gn=3.5175e+00  val_loss=:
Epoch 19  train_loss=1.4695 train_acc=0.4603 train_gn=3.8680e+00  val_loss=:
Epoch 20  train_loss=1.4437 train_acc=0.4705 train_gn=3.7106e+00  val_loss=:
Epoch 21  train_loss=1.4280 train_acc=0.4720 train_gn=3.7727e+00  val_loss=:
Epoch 22  train_loss=1.4054 train_acc=0.4861 train_gn=3.6786e+00  val_loss=:
Epoch 23  train_loss=1.3905 train_acc=0.4930 train_gn=3.8366e+00  val_loss=:
Epoch 24  train_loss=1.3739 train_acc=0.4966 train_gn=3.8903e+00  val_loss=:
Epoch 25  train_loss=1.3641 train_acc=0.5021 train_gn=3.9906e+00  val_loss=:
Epoch 26  train_loss=1.3508 train_acc=0.5011 train_gn=3.9217e+00  val_loss=:
Epoch 27  train_loss=1.3267 train_acc=0.5154 train_gn=3.8208e+00  val_loss=:
Epoch 28  train_loss=1.3331 train_acc=0.5098 train_gn=4.1089e+00  val_loss=:
Epoch 29  train_loss=1.3062 train_acc=0.5235 train_gn=4.0609e+00  val_loss=:
Epoch 30  train_loss=1.2995 train_acc=0.5240 train_gn=3.9554e+00  val_loss=:
Epoch 31  train_loss=1.2904 train_acc=0.5316 train_gn=4.1376e+00  val_loss=:
Epoch 32  train_loss=1.2791 train_acc=0.5374 train_gn=4.0783e+00  val_loss=:
```



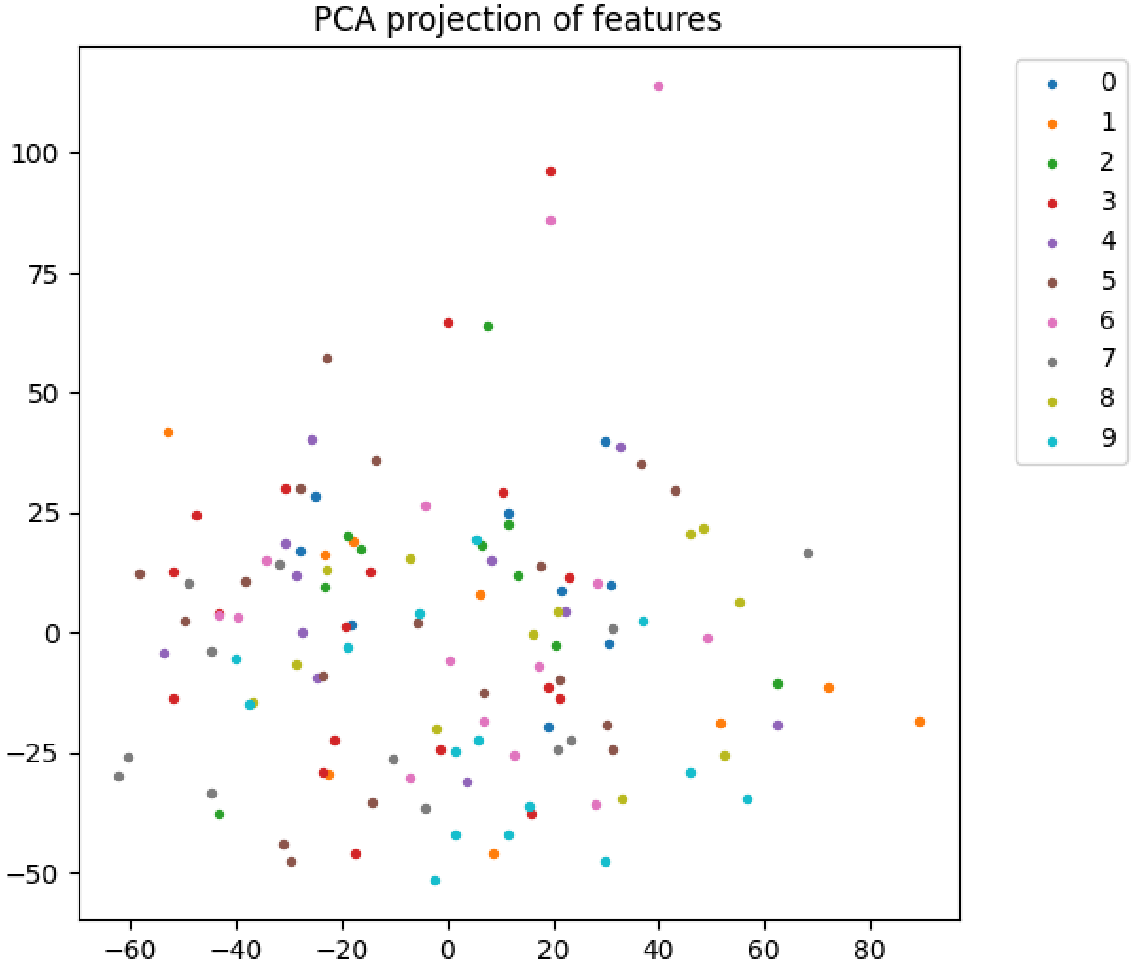

Accuracy vs epoch
train_acc
val_acc
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
5
10
15
20
25
30

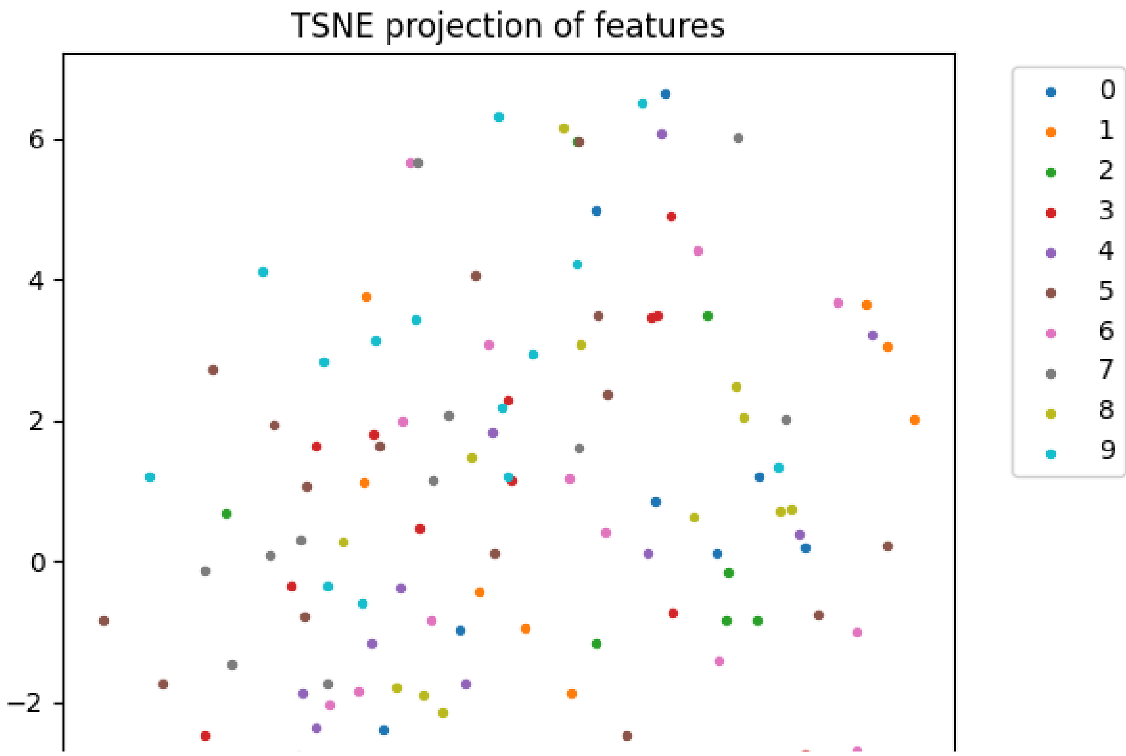

Gradient norm vs epoch
train_grad_norm
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
5
10
15
20
25
30

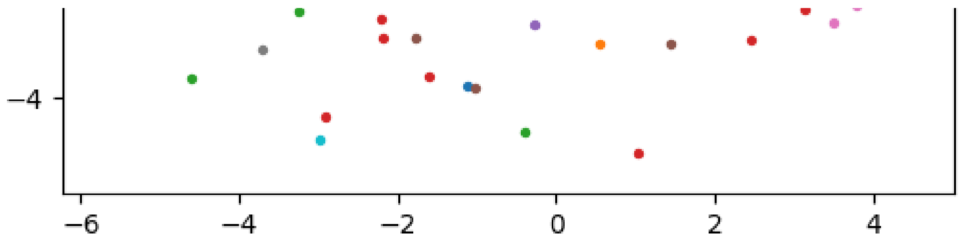

Conv1 filters (trained)

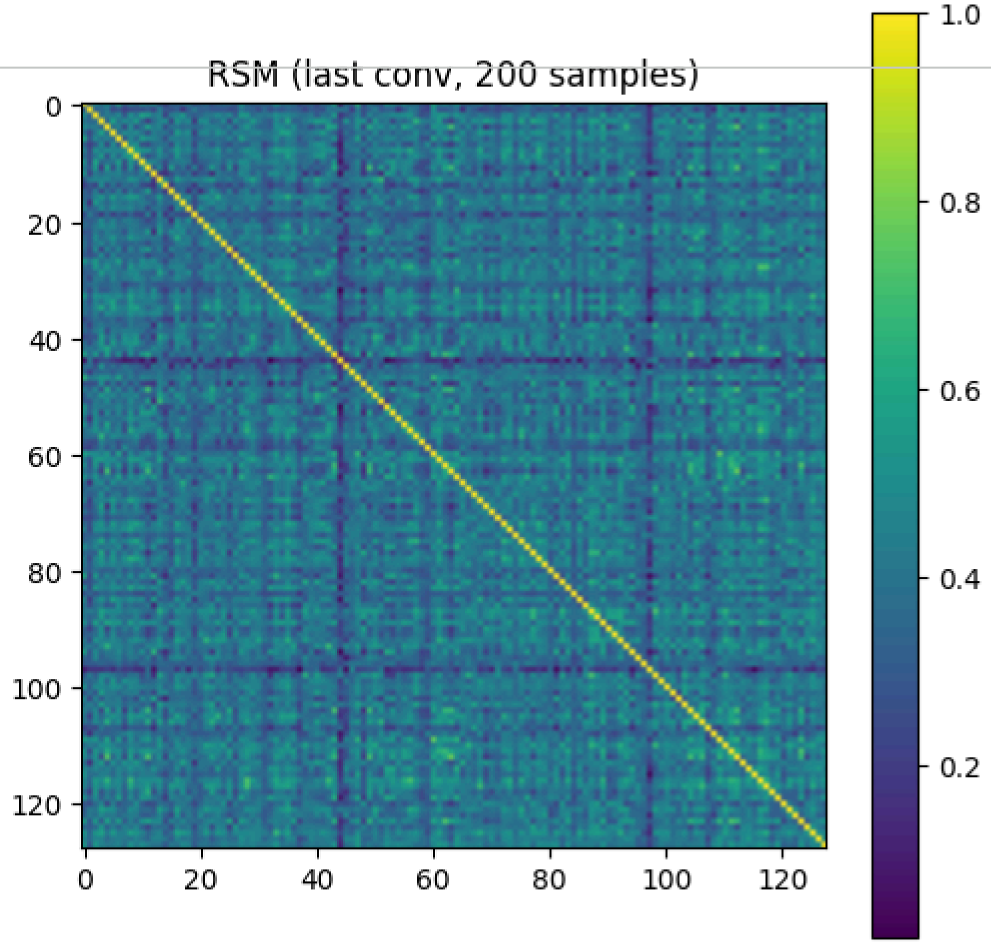


```
Gradient check indices: [(np.int64(2), np.int64(2), np.int64(0), np.int64(1
Relative errors: [np.float32(0.009731285), np.float32(0.0007628997), np.floa
```










```
Estimated top Hessian eigenvalue (approx): 214.97503662109375
```