A Technical Report of Convolutional Neural Network

Xu Chen 2257453

December 10, 2025

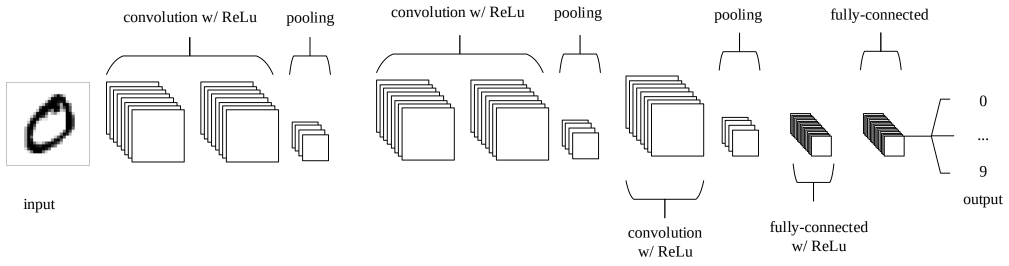


Figure 1: A typical architecture of a Convolutional Neural Network (CNN) from [O'Shea and Nash, 2015].

1. Two key components in CNN

1.1. Convolutional Kernels

Convolutional kernels (also called filters) are small learnable tensors that slide across an input image to extract local features. The operation, known as convolution, effectively transforms the input data into feature maps. Each feature map highlights a specific type of local pattern, such as edges, textures, or corners, learned by the corresponding kernel.

Formally, for an input feature map $\mathbf{X} \in \mathbb{R}^{H\times W\times C}$ (where H, W, and C are height, width, and number of channels) and a kernel $\mathbf{K} \in \mathbb{R}^{h\times w\times C}$ (with kernel height h and width w), the convolution at spatial location (i, j) is computed as:

$$(\mathbf{X} * \mathbf{K})(i,j) = \sum_{c=1}^{C}\sum_{u=1}^{h}\sum_{v=1}^{w} \mathbf{X}(i+u-1,\, j+v-1,\, c)\, \mathbf{K}(u,v,c).$$

The key mechanism of a CNN is parameter sharing (or weight sharing): the same kernel $\mathbf{K}$ is used across all spatial locations (i, j) of the input, drastically reducing the number of free parameters compared to a fully connected layer. This also provides the network with translation invariance, meaning a feature can be detected regardless of its position in the image.

1.2. Loss Function: Cross-Entropy

For a C-class classification problem, the cross-entropy loss measures the discrepancy between the predicted distribution and the one-hot ground-truth label $\mathbf{y}$. It is defined as:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{C} y_i \log p_\theta(i \mid \mathbf{x}),$$

where the predicted class probabilities are obtained through a softmax transformation:

$$p_\theta(i \mid \mathbf{x}) = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)},$$

and $(z_1, \ldots, z_C)$ are the logits generated by the CNN. The purpose of the loss function is to quantify the model's error, which is then used by the backpropagation algorithm to update the network's weights (θ) during training.

2. Training and Evaluation on CIFAR-10 [Krizhevsky, 2009]

2.1. Experimental Setup

The training and evaluation of the CNN framework were conducted using the following hardware and software configuration:

- **GPU Hardware:** NVIDIA GeForce RTX 4090 24GB
- **Deep Learning Framework:** torch 2.8.0 with torchvision 0.23.0
- **Operating System:** Linux (Ubuntu 20.04)
- **CUDA version:** 13.0
- **Python version :** 3.9

2.2. Training and Inference Steps Calculation

The total number of steps is calculated based on the training set split (40,000 for training, 10,000 for validation/testing) and the Batch Size of 1024.

- **Batch Size: 1024**
- **Optimizer:** SGD and Adam [Kingma and Ba, 2017].
- **Learning Rate Schedule:** Cosine Annealing Scheduler.
- **Loss Function:** Cross-Entropy Loss
- **Epochs:** 100.

Training Steps The total number of optimizer updates (optimizer.step()) across all epochs:

$$\text{Steps}_{\text{epoch}}^{\text{train}} = \lceil \frac{\text{Training Samples}}{\text{Batch Size}} \rceil = \lceil \frac{40,000}{1024} \rceil = 40 \text{ steps}$$

$$\text{Total Training Steps} = \text{Steps}_{\text{epoch}}^{\text{train}} \times \text{Epochs} = 40 \times 100 = \mathbf{4,000} \text{ steps}$$

Inference Steps (Validation) The total number of forward passes performed on the validation set during the training process:

$$\text{Steps}_{\text{epoch}}^{\text{val}} = \lceil \frac{\text{Validation Samples}}{\text{Batch Size}} \rceil = \lceil \frac{10,000}{1024} \rceil = 10 \text{ steps}$$

$$\text{Total Validation Steps} = \text{Steps}_{\text{epoch}}^{\text{val}} \times \text{Epochs} = 10 \times 100 = \mathbf{1,000} \text{ steps}$$

Final Test Steps The number of forward passes required for the single, final evaluation on the 10,000 image test set:

$$\text{Final Test Steps} = \lceil \frac{\text{Test Samples}}{\text{Batch Size}} \rceil = \lceil \frac{10,000}{1024} \rceil = \mathbf{10} \text{ steps}$$

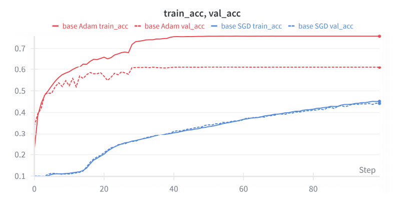
2.3. Network Architecture and Results

The final model achieved an overall test accuracy of **61**% on the CIFAR-10 test set. This performance was achieved using the Basic Net architecture on Github.

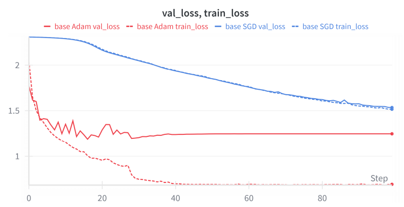
This model is a classic LeNet-style CNN structure, comprising two convolutional layers, two max-pooling layers, and three fully connected layers. Its structure is detailed in Table 1.

Table 1: BaseNet Architecture Details on CIFAR-10 (32×32 Input)

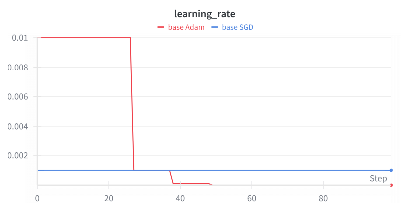
Layer (L)	**Type**	**Core Parameter**	**Input Size**	**Output Size**
$\mathbf{L}_1$	Conv2D	5×5, 6 filters	$3 \times 32 \times 32$	$6 \times 28 \times 28$
$\mathbf{P}_1$	MaxPool2D	2×2	$6 \times 28 \times 28$	$6 \times 14 \times 14$
$\mathbf{L}_2$	Conv2D	5×5, 16 filters	$6 \times 14 \times 14$	$16 \times 10 \times 10$
$\mathbf{P}_2$	MaxPool2D	2×2	$16 \times 10 \times 10$	$16 \times 5 \times 5$
$\mathbf{F}$	Flatten	-	$16 \times 5 \times 5$	400
$\mathbf{L}_3$	Linear (FC1)	-	400	120
$\mathbf{L}_4$	Linear (FC2)	-	120	84



(a) Training and Validation Accuracy Curve



(b) Training and Validation Loss Curve



(c) Learning Rate Schedule

Figure 2: Visual Summary of Model Optimization. (a) illustrates training and validation accuracy; (b) shows loss reduction, which often stabilizes faster with Adam than SGD; and (c) displays the dynamic adjustment of the learning rate.

2.4. Why Adam is More Effective

Standard SGD Update Standard SGD updates the weights (θ) based only on the current gradient ($\mathbf{g}_t$) and a fixed global learning rate (α):

$$\theta_{t+1} = \theta_t - \alpha \cdot \mathbf{g}_t$$

SGD struggles with features that require dramatically different learning rates (e.g., dense vs. sparse features).

Adam Update Adam calculates a separate, adaptive learning rate for each parameter based on two key components: the moving average of the gradient (first moment, $\mathbf{m}_t$) and the moving average of the squared gradient (second moment, $\mathbf{v}_t$).

1. **Moment Calculation:** At time step t, Adam computes exponential moving averages of the gradient and the squared gradient:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \quad \text{(First Moment / Mean)}$$
$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \quad \text{(Second Moment / Uncentered Variance)}$$

2. **Bias Correction:** To account for the initialization of $\mathbf{m}_0$ and $\mathbf{v}_0$ to zero, Adam applies bias correction, especially important in the early training steps:

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \quad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}$$

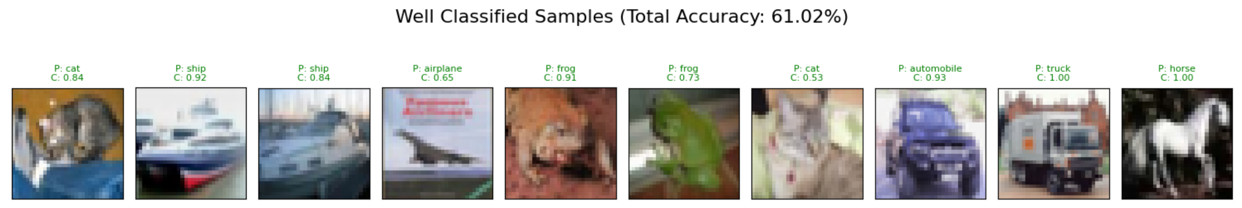
3. **Parameter Update Rule:** The final update for the weights is a step in the direction of the mean gradient $\hat{\mathbf{m}}_t$, scaled inversely by the square root of the variance $\sqrt{\hat{\mathbf{v}}_t}$.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \cdot \hat{\mathbf{m}}_t$$

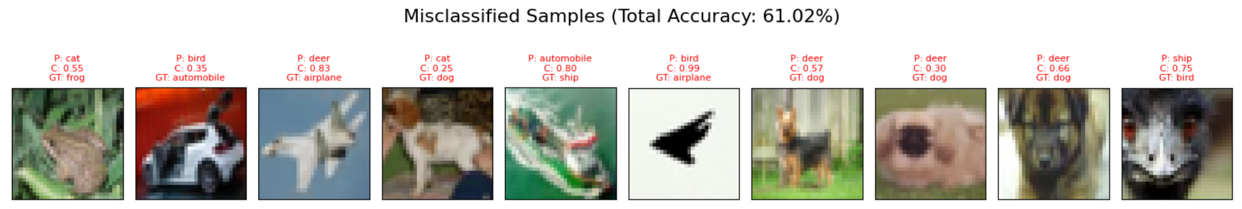
The term $\frac{\alpha}{\sqrt{\hat{\mathbf{v}}_t}+\epsilon}$ represents the adaptive learning rate. Parameters that have experienced high-variance (large squared gradients in $\hat{\mathbf{v}}_t$) are penalized with a smaller step size, preventing oscillations. Conversely, parameters with stable, low-variance gradients can take larger steps. This parameter-specific learning rate scaling allows Adam to find the optimal path in the loss landscape much more efficiently and reliably than the fixed, global learning rate of SGD, thereby contributing to the faster convergence observed in complex CNN models.

2.5. Classified and Misclassified Images

To gain insights into the model's decision-making process, we analyze representative samples from the CIFAR-10 test set, focusing on the model's confidence in its predictions.



(a) Well-Classified Samples (High Confidence)


Misclassified Samples (Total Accuracy: 61.02%)
P: cat C: 0.55 GT: frog
P: bird C: 0.35 GT: automobile
P: deer C: 0.83 GT: airplane
P: cat C: 0.25 GT: dog
P: automobile C: 0.80 GT: ship
P: bird C: 0.99 GT: airplane
P: deer C: 0.57 GT: dog
P: deer C: 0.30 GT: dog
P: deer C: 0.66 GT: dog
P: ship C: 0.75 GT: bird

(b) Misclassified Samples (Low/Confused Confidence)

Figure 3: Qualitative Analysis of model Performance. (a) shows samples where the predicted label (P) matches the ground truth (GT) with high confidence. (b) shows errors, where the predicted label (P) is incorrect, often exhibiting lower confidence or confusion between similar classes. The reported confidence (C) is the maximum Softmax probability.

2.6. Evaluation Metrics

The qualitative analysis of image samples is grounded in both macro-level quantitative metrics and granular, per-sample indicators. The following metrics were employed to contextualize the **61%** overall accuracy achieved by the BaseNet:

1. **Overall Test Accuracy and Per-Class Accuracy:** The achieved overall accuracy of 61% serves as the performance baseline. More critically, the analysis utilizes the **Per-Class Accuracy** to identify specific categories where the model performs poorly (e.g., Cat vs. Dog). These low-accuracy classes often correspond directly to the types of images found in the misclassified samples of Figure 3b.

2. **Classification Confidence (C):** The most crucial metric for the image analysis is the classification confidence, defined as the maximum probability output by the Softmax function (σ) over the network's final output logits ($\mathbf{z}$).

   $$C = \max_i(\sigma(\mathbf{z}))_i$$

   This metric provides a measure of the model's certainty in its prediction and is indispensable for distinguishing the samples:

   - **Well-Classified Samples:** These samples are typically accompanied by a high confidence value ($C > 0.7$), validating the model's strong certainty in its correct prediction.
   - **Misclassified Samples:** These failures are often marked by a low confidence value ($C < 0.4$) or similar probabilities across multiple confused classes, highlighting the model's ambiguity when facing ambiguous or complex features.

2.7. Findings and Discussions

2.7.1 Optimization Efficiency: The Mathematical Advantage of Adam

From Figure 2b and Section 2.4, the utilization of the Adam optimizer was a critical decision for achieving stable and efficient training, as supported by its theoretical foundation and the smooth convergence trajectory observed. This mechanism is mathematically advantageous for navigating the complex and rugged loss landscapes inherent to deep convolutional neural networks. By automatically dampening oscillations in high-variance directions and accelerating steps in stable directions, Adam ensures that the Net architecture reaches a better local minimum more reliably and significantly faster than a standard fixed-rate optimizer.

2.7.2 Architectural Limitations

The overall test accuracy achieved by **61%**, while significantly better than random guessing, represents a performance ceiling imposed by the intrinsic limitations of the Net architecture. As a shallow network containing only two convolutional layers, the model fundamentally lacks the representational capacity required to learn the complex hierarchical semantic features necessary for fine-grained discrimination in the CIFAR-10 dataset. The network successfully extracts primary low-level features (e.g., edges and colors). However, it fails to learn the abstract feature compositions needed to accurately distinguish between visually similar classes (e.g., differentiating between a cat and a dog). The model exhibits clear signs of **architectural underfitting**, indicating that the primary bottleneck to higher performance is the depth of the model, necessitating a migration to a more sophisticated architecture, such as ResNet, to unlock further accuracy gains.

2.7.3 The Need for Augmentation

The qualitative analysis of the misclassified samples in Figure 3b provides critical evidence on the failure modes of the model and the quality of the input data. A significant portion of the misclassified images are visually ambiguous even under human inspection, often suffering from low resolution (32×32), severe occlusion, or challenging viewpoints inherent in the dataset. Furthermore, errors occur predominantly between visually confused classes, often showing **low classification confidence** across multiple labels. This suggests that the standard data preprocessing (normalization) is insufficient to render the model robust against these ambiguities. To enhance the generalization of the model and force it to focus on essential distinguishing features, the implementation of more aggressive **Data Augmentation** techniques (such as random cropping, affine transformations, or color jittering) is strongly recommended for future iterations, with the aim of improving the model's invariance to minor distortions and increasing its robustness.

3. Model Improvement

Due to the relatively simple network structure and less-than-ideal performance, I'm more focused on achieving better performance rather than reducing model size. ResNet is a popular approach for this.

3.1. Why Introduce ResNet?

The initial results using the shallow Net architecture (Section 2.3 demonstrated a performance ceiling of **61**% accuracy, a limitation attributed to **architectural underfitting**. The key to surpassing this limit is increasing model depth to capture complex features. However, simply stacking more layers in a traditional network leads to the Vanishing Gradient and Degradation problems.

The Residual Network (**ResNet**) [He et al., 2015] solves this through the introduction of the Residual Block. Instead of learning a direct mapping $H(x)$, the block learns a residual function $F(x)$, where $H(x) = F(x) + x$. This identity shortcut connection ensures two critical advantages:

- **Gradient Flow:** It provides an unimpeded path for gradients to flow backward, mitigating the Vanishing Gradient problem and enabling the training of very deep networks.
- **Degradation Prevention:** It prevents the Degradation Problem by allowing layers to easily learn the identity mapping ($F(x) = 0$), guaranteeing that adding more layers does not hurt performance.

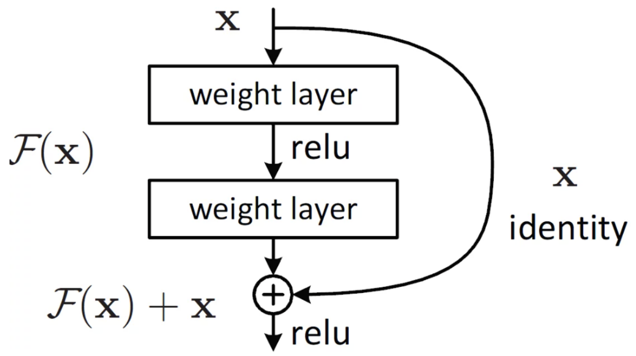
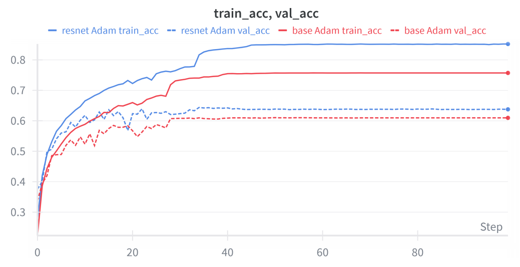

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

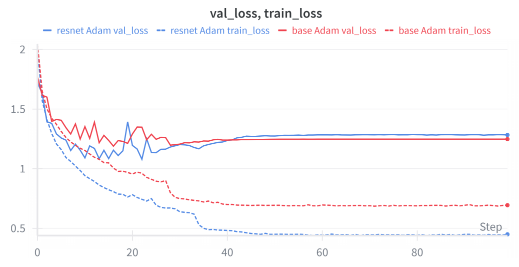
Figure 4: Residual learning: a building block. For Real Implements see Appendix 1

3.2. Ablation Study: IS ResNet useful on the Shallow BaseNet?

No. To isolate the impact of the residual connection, an ablation study was performed by modifying only the **conv2** block of the shallow BaseNet architecture into a Residual Block, keeping the rest of the model depth and width constant see Appendix 1. The intention was to observe the benefit of the residual connection even in a small setting, but the result in test data accuracy only improve form 61 to 63. This limited improvement suggests that the primary bottleneck was not the gradient flow or degradation in the shallow BaseNet, but rather than the sufficient depth itself. ResNet's true advantage—its ability to scale effectively—was suppressed by the small size of the original network. The residual connection, while beneficial, provides minimal gain when the overall model capacity is too low to extract complex features.



(a) Training and Validation Accuracy Curve



(b) Training and Validation Loss Curve

Figure 5: Although Resnet have a good training curve , it dose not perform well on validation data.

3.3. Motivation: Necessity of Model Scaling and Data Augmentation

As the model lacked the intrinsic capacity to learn the highly abstract and non-linear feature hierarchies required for complex CIFAR-10 image discrimination. While the attempted ResNet ablation study confirmed the importance of deep learning concepts, the fundamental solution was scaling. Transitioning from the BaseNet's minimal structure to a high-capacity Deep Convolutional Network (DeepNet) was essential to provide the millions of parameters needed to fully map the input space and break the 61% performance barrier.

The qualitative analysis of misclassified Figure 3b and Figure 7b showed that the model failed primarily on samples exhibiting subtle variations, poor lighting, or challenging orientations. This suggested the BaseNet suffered from poor generalization and robustness. To force the high-capacity DeepNet model to learn the invariant features of objects (e.g., recognizing a car regardless of its angle or size) and prevent it from overfitting its large parameter set to the limited training data, a strategy of aggressive **Data Augmentation** was introduced. This step ensures that the model learns essential semantic information rather than memorizing noise or background clutter.

3.4. Implementation: Data Augmentation Strategy and DeepNet Architecture

Data Augmentation Implementation A composite set of data augmentation techniques was applied to the training dataset. These transformations were carefully selected to simulate typical real-world variations, thereby boosting the model's resistance to minor shifts in input data. The specific augmentation pipeline is detailed in Table 2.

Table 2: Training Data Augmentation Pipeline: Functional Description

Transformation	**Purpose**
Random Flip	Improves invariance to minor changes in object pose and size.
Tensor Conversion	Prepares data for GPU processing and scales pixel values to $[0, 1]$.
Normalization	Standardizes data distribution across channels to $[-1, 1]$.

DeepNet Architecture The final high-performance model, a Deep Convolutional Network (DeepNet), features significant increases in depth, width, and incorporates strong regularization techniques. This architecture successfully achieved an accuracy of approximately **90%**. Table 3 provides a summary of its structure, which notably relies on Batch Normalization (BN) and Dropout for stabilization and regularization. See Appendix 1a

Table 3: DeepNet Architecture Details (Final High-Performance Model)

Layer / Block	**Operation**	**Output Feature Map**	**Key Regularization**
Conv Block 1	$\text{Conv}(5 \times 5, 128) \times 2$	$128 \times 16 \times 16$	BN $\to$ MaxPool $\to$ Dropout2D(0.25)
Conv Block 2	$\text{Conv}(3 \times 3, 256) \times 2$	$256 \times 8 \times 8$	BN $\to$ MaxPool $\to$ Dropout2D(0.25)
Flatten	$\text{View}(-1, 16384)$	16384	-
FC Block 1	$\text{Linear}(16384 \to 1024)$	1024	BN $\to$ Dropout(0.5)
FC Block 2	$\text{Linear}(1024 \to 512)$	512	BN $\to$ Dropout(0.5)
Output	$\text{Linear}(512 \to 10)$	10 (Classes)	-

1. **Batch Normalization (BN)**: BN is applied to stabilize the distributions of layer inputs throughout training, specifically counteracting the *Internal Covariate Shift*. For a mini-batch $\mathcal{B} = \{x_1, \ldots, x_m\}$, the output y_i of BN for input x_i is given by:

$$y_i = \gamma \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta$$

   where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ are the mean and variance computed over the mini-batch, and γ and β are learnable scaling and shifting parameters, respectively.

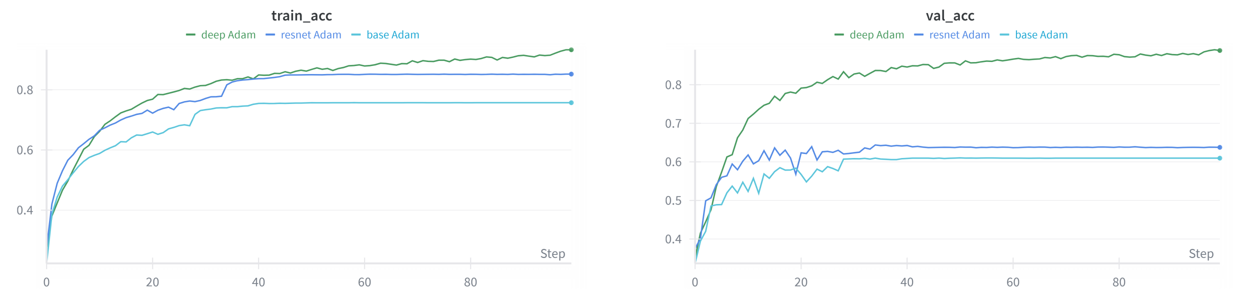
2. **Dropout**: To prevent co-adaptation of features in the deep layers and mitigate overfitting, Dropout is applied to both convolutional and dense layers. During training, for a layer output vector z, the Dropout operation generates a masked vector z':

$$z'_i = \begin{cases} \frac{z_i}{1-p} & \text{with probability } 1-p \\ 0 & \text{with probability } p \end{cases}$$

   where p is the dropout rate (0.25 or 0.5). By randomly setting a fraction p of the activations to zero, Dropout forces the network to learn a more robust, non-redundant set of features.

3.5. Performance Comparison

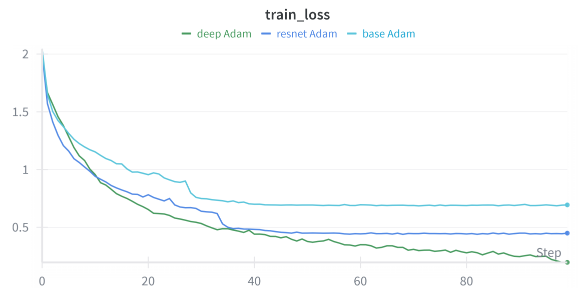
1. **DeepNet vs. BaseNet:** The 29% performance gap (61% to 90%) directly confirms that the primary bottleneck was **architectural underfitting**. The DeepNet's increased capacity (18.6 Million parameters), combined with its greater depth and width, was essential for learning the complex feature representations that the simple BaseNet could not capture.



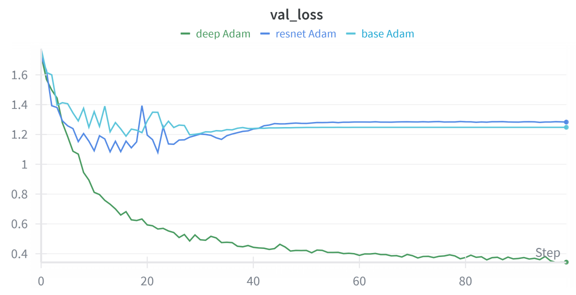
(a) Training Accuracy Curve

(b) Validation Accuracy Curve

Figure 6: The training curves between BaseNet, BaseNet with Residual and DeepNet



(a) Training Loss Curve



(b) Validation Loss Curve

Figure 7: The loss curves between BaseNet, BaseNet with Residual and DeepNet

2. **Stability:** Achieving high accuracy with a high-capacity model requires aggressive regularization. The integration of **Batch Normalization**(**BN**) stabilized the training of the deep architecture, while **Dropout** and **Data Augmentation** provided the necessary robustness to mitigate the severe overfitting that would otherwise occur with such a large parameter set.

3. **Comparison to SOTA:** While the $\sim 90\%$ accuracy is highly competitive for a custom Deep CNN implementation, state-of-the-art (SOTA) results on CIFAR-10 often exceed 95% using highly optimized architectures like deep ResNets (e.g., ResNet-50) or specialized designs like DenseNet. The remaining gap between 90% and 95%+ suggests that further gains would require adopting canonical residual frameworks or more sophisticated regularization techniques like Cutout or Mixup.

3.6. Findings of multi-classification tasks

Multi-class classification fundamentally transitions the learning objective from predicting a single scalar (as in binary tasks) to estimating a conditional probability distribution $\mathbf{P}(Y|X)$, where the model must assign a probability to each of the $K > 2$ classes. This shift introduces significant complexities:

1. **High-Dimensional Decision Boundary Partitioning:** The core mathematical challenge is partitioning the high-dimensional feature space ($\mathbb{R}^D$) into K distinct regions. The model must learn K complex, non-linear decision boundaries simultaneously. The statistical risk of **Overlapping Distributions** (i.e., high visual or semantic similarity between classes, such as 'cat' and 'dog' on CIFAR-10) increases rapidly with K, requiring the model's feature map to be exceptionally discriminative to minimize classification error.

2. **Statistical Variance and Imbalance:** In statistical learning, the difficulty of distinguishing between classes is fundamentally measured by the $\mathbf{Kullback - Leibler}(\mathbf{KL})$ **Divergence** between their feature distributions, $\mathbf{P}_k(\mathbf{X})$, where k is the class index. For two distributions, $\mathbf{P}$ (True Distribution) and $\mathbf{Q}$ (Model Prediction), the KL Divergence is defined as:

$$D_{KL}(\mathbf{P}||\mathbf{Q}) = \sum_x \mathbf{P}(x) \log\left(\frac{\mathbf{P}(x)}{\mathbf{Q}(x)}\right)$$

Low D_{KL} (high overlap) mandates models with vast capacity to learn subtle decision boundaries. This is directly linked to the empirical risk minimization through the standard $\mathbf{Cross - Entropy}(\mathbf{CE})$ **Loss** function, which is minimized when $D_{KL}(\mathbf{P}||\mathbf{Q})$ is minimized. For a multi-class task with K classes, the total empirical CE Loss ($\mathcal{L}_{CE}$) over a dataset $\mathcal{D}$ is the expected value of the negative log-likelihood:

$$\mathcal{L}_{CE} = E_{(X,Y)\sim\mathcal{D}}[-\log(\mathbf{Q}(Y|X))] = -\sum_{(x_i,y_i)\in\mathcal{D}} \log(\mathbf{Q}(y_i|x_i))$$

If the prior probabilities $\mathbf{P}(Y)$ are highly skewed (**Class Imbalance**), the expected loss is dominated by the abundant classes. Let $\mathbf{P}(Y=k)$ be the true probability of sampling class k. The empirical loss can be approximated by weighting the class-conditional expected loss $E_{X|Y=k}$ by the class prior:

$$\mathcal{L}_{CE} \approx \sum_{k=1}^{K} \mathbf{P}(Y=k) \cdot E_{X|Y=k}[-\log(\mathbf{Q}(Y=k|X))]$$

In a severely imbalanced scenario, the probability of sampling the majority class, $\mathbf{P}(Y = \text{majority})$, is significantly larger than $\mathbf{P}(Y = \text{minority})$. This **statistical weighting** causes the total loss gradient to be overwhelmed by the majority class samples, statistically ignoring the minority classes. Consequently, the model optimizes its parameters primarily to achieve high accuracy on the abundant classes, leading to poor generalization and recall for the vital, yet infrequent, minority classes. This phenomenon is why specialized algorithms like **Focal Loss** are necessary to re-weight the contribution of easy, abundant samples.

3. **The Curse of Dimensionality:** For high-dimensional inputs (like images or long sequences), the volume of the feature space grows exponentially, making the training data statistically sparse. This necessitates powerful algorithms capable of efficient representation learning—projecting the high-dimensional input into a lower-dimensional, highly-separable latent space.

Challenges and Solutions The design of a robust multi-class algorithm must also address the intrinsic statistical nature of the data modality:

- **Image Data (Spatial Invariance):** The challenge is maintaining spatial and geometric invariance. A high-performing algorithm must identify an object regardless of its position, scale, or rotation. The solution is the hierarchical feature extraction of **Convolutional Networks** combined with aggressive Data Augmentation (geometric and color transformations) to statistically enrich the training distribution.
- **Text Data (Sequential and Semantic Dependencies):** The challenge is modeling long-range sequential dependencies and semantic ambiguity (polysemy). Words are tokens, and their meaning is context-dependent. The solution involves sequence models (like RNNs/LSTMs) and, more powerfully, **Transformer Architectures** that use Attention Mechanisms to statistically weigh the importance of every word relative to every other word, thereby capturing global contextual dependencies for accurate classification.
- **Audio Data (Temporal and Spectral Dependencies):** The challenge involves processing time-frequency representations (spectrograms) where key features (pitch, timbre, duration) are distributed across time and frequency axes. The solution uses specialized Time Frequency Convolutional Networks or hybrid CNN-RNN models to capture both local spectral patterns and long-term temporal dependencies essential for tasks like speech recognition or music classification.

4. Conclusion

This study rigorously analyzed the impact of model capacity and regularization on multi-class image classification performance on the CIFAR-10 dataset. The initial BaseNet architecture, a shallow CNN, established a performance ceiling of **61**% accuracy, confirming the limitation of architectural underfitting in complex classification tasks. The subsequent ablation study demonstrated that merely adding the Residual Connection (ResNet principle) to the shallow network yielded only marginal gains, proving that the true bottleneck was a lack of scale, not training stability.

The final transition to the high-capacity DeepNet architecture, coupled with aggressive Batch Normalization and **Dropout** regularization, successfully overcame this limitation, achieving a robust test accuracy of approximately **90**%. This result affirms that achieving high performance in multi-class environments necessitates both sufficient model capacity to learn complex feature distributions and strong statistical stabilization techniques to mitigate overfitting. The challenge of future algorithm design must therefore focus on task-specific nuances, such as specialized attention mechanisms for Fine-Grained Classification and robust loss function design for Long Tailed Distribution problems.

References

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. URL https://api.semanticscholar.org/CorpusID:18268744.

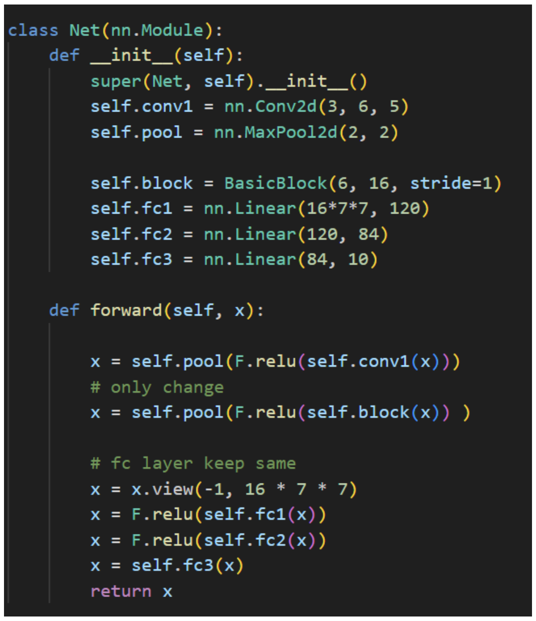
Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015. URL https://arxiv.org/abs/1511.08458.

A. Code and Experiments

See the Github Code and Wandb.

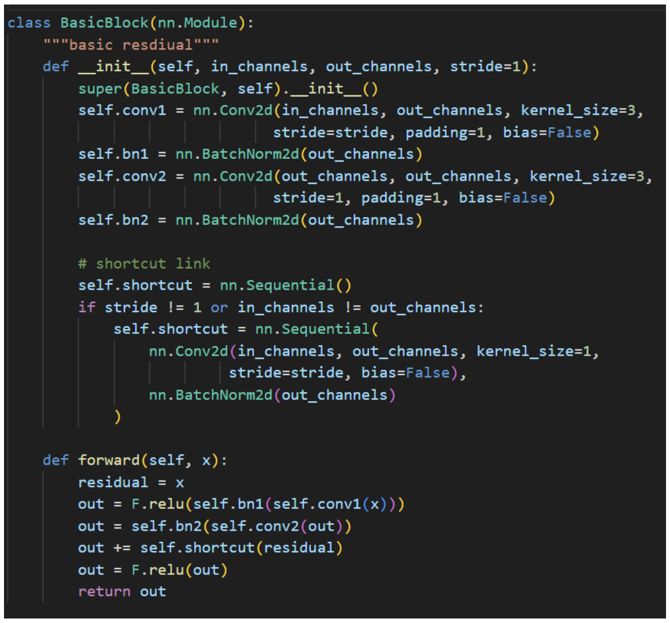
B. Implements of Resnet into Base Net

Figure 1: Architectural Variants Used in the Improvement Phase



```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)

        self.block = BasicBlock(6, 16, stride=1)
        self.fc1 = nn.Linear(16*7*7, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))
        # only change
        x = self.pool(F.relu(self.block(x)) )

        # fc layer keep same
        x = x.view(-1, 16 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

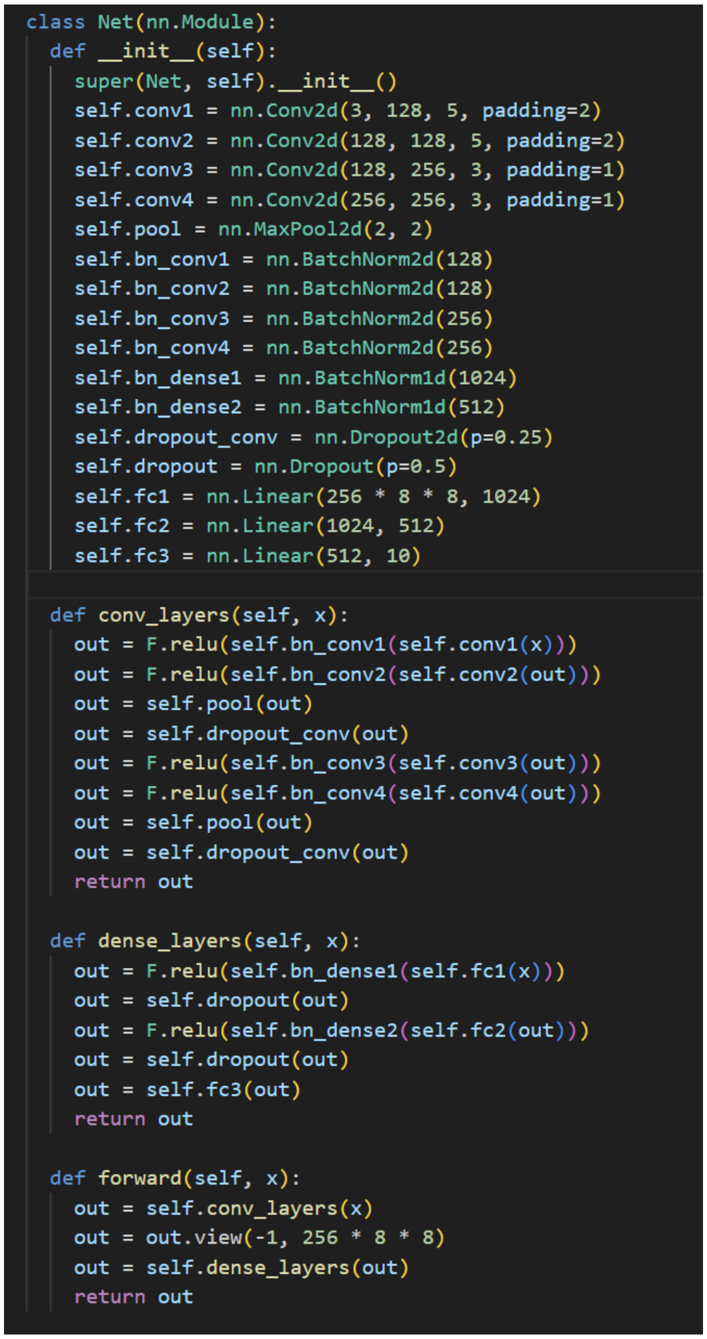
(a) conv2 Block replaced by a Residual



```python
class BasicBlock(nn.Module):
    """basic resdiual"""
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # shortcut link
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
                         stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(residual)
        out = F.relu(out)
        return out
```

(b) The BasicBlock

C. Implements of DeepNet



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, 5, padding=2)
        self.conv2 = nn.Conv2d(128, 128, 5, padding=2)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.conv4 = nn.Conv2d(256, 256, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.bn_conv1 = nn.BatchNorm2d(128)
        self.bn_conv2 = nn.BatchNorm2d(128)
        self.bn_conv3 = nn.BatchNorm2d(256)
        self.bn_conv4 = nn.BatchNorm2d(256)
        self.bn_dense1 = nn.BatchNorm1d(1024)
        self.bn_dense2 = nn.BatchNorm1d(512)
        self.dropout_conv = nn.Dropout2d(p=0.25)
        self.dropout = nn.Dropout(p=0.5)
        self.fc1 = nn.Linear(256 * 8 * 8, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 10)

    def conv_layers(self, x):
        out = F.relu(self.bn_conv1(self.conv1(x)))
        out = F.relu(self.bn_conv2(self.conv2(out)))
        out = self.pool(out)
        out = self.dropout_conv(out)
        out = F.relu(self.bn_conv3(self.conv3(out)))
        out = F.relu(self.bn_conv4(self.conv4(out)))
        out = self.pool(out)
        out = self.dropout_conv(out)
        return out

    def dense_layers(self, x):
        out = F.relu(self.bn_dense1(self.fc1(x)))
        out = self.dropout(out)
        out = F.relu(self.bn_dense2(self.fc2(out)))
        out = self.dropout(out)
        out = self.fc3(out)
        return out

    def forward(self, x):
        out = self.conv_layers(x)
        out = out.view(-1, 256 * 8 * 8)
        out = self.dense_layers(out)
        return out
```

(a) DeepNet