DeepChem-Variant: A Modular Open Source Framework for Genomic Variant Calling

Ankita Vaishnobi Bisoi [1 2] **Shreyas V** [1 2] **Jose Siguenza** [2] **Bharath Ramsundar** [2]

Abstract

Variant calling is a fundamental task in genomic research for detecting genetic variations such as single nucleotide polymorphisms (SNPs) and insertions or deletions (indels). This paper presents an enhancement to DeepChem (Ramsundar et al., 2019), a widely used open source drug discovery framework, through the integration of DeepVariant (Poplin et al., 2018). We introduce DeepChem-Variant, a variant calling pipeline that leverages DeepVariant's convolutional neural network (CNN) architecture to improve variant detection accuracy and reliability. DeepChem-Variant has stages for realignment of sequencing reads, candidate variant detection, and pileup image generation, followed by variant classification using either the original modified Inception V3 model or our novel MobileNetV2 implementation. We performed 3 case studies to validate our approach. Our work also contributes optimized utility functions for genomic data formats, including enhanced DataLoaders for BAM, SAM, and CRAM files, and an optimized FASTALoader. These implementations collectively provide a modular and extensible variant calling framework within DeepChem, enabling tighter integration of DeepChem's drug discovery infrastructure with bioinformatics pipelines for future research.

1. Introduction

Variant calling identifies single nucleotide polymorphisms (SNPs) and insertions/deletions (indels) from sequencing data, foundational for population genetics, disease etiology, and precision medicine applications including risk prediction and therapeutic interventions. Standard approaches like GATK (McKenna et al., 2010) and SAMtools (Li et al., 2009) use probabilistic models that struggle with ambiguous or low-quality data. These methods face challenges in noisy genomic regions, reducing sensitivity and specificity in low-coverage areas or regions with complex structural variations.

DeepVariant (Poplin et al., 2018; Poplin, 2017), developed by Google, uses a Convolutional Neural Network (CNN) (Krizhevsky et al., 2012) to reframe the variant calling task as an image classification problem. Pileup images (Section 2.2) are generated from sequencing reads and analyzed by the CNN to distinguish true variants from sequencing errors. This approach outperforms traditional heuristic-based methods, achieving higher accuracy in variant detection, particularly across diverse sequencing platforms and in regions where conventional tools exhibit reduced performance.

However, while the code for DeepVariant is accessible on platforms such as GitHub, part of the code is written in C++ and is challenging to modify or extend. The architecture and components are fixed within the provided framework, making it difficult for researchers to adapt DeepVariant to explore novel hypotheses or improve specific sub-components for their experimental needs.

To address the need for modular open-source implementations of computational genomic tools, we integrate an implementation of DeepVariant as DeepChem-Variant into the DeepChem (Ramsundar et al., 2019) framework. DeepChem, an open-source Python library designed for scientific machine learning and deep learning, has established itself as a versatile platform for applications in molecular machine learning ranging from the MoleculeNet benchmark suite (Wu et al., 2018) to protein-ligand interaction modeling (Gomes et al., 2017), and generative modeling of molecules (Frey et al., 2022), among others.

Deepchem's modular architecture, comprising components such as data loaders, featurizers, splitters, models, and metrics, provides an extensible system that supports the development of custom workflows. The DeepChem community of developers and contributors actively maintains all implementations in this system, which are validated through continuous integration and delivery (CI/CD) pipelines. The incorporation of DeepChem-Variant into DeepChem significantly broadens its functionality, enabling variant calling workflows to be conducted entirely within an open-source Python ecosystem. We anticipate this infrastructure will enable subsequent computational work exploring the intersection of drug discovery and bioinformatics.

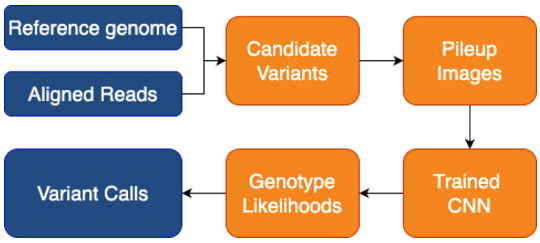

Figure 1. DeepVariant workflow: Reference genome and aligned reads generate candidate variants, which are converted to pileup images, processed by a trained CNN to produce genotype likelihoods, and finally output as variant calls.

2. Implementations

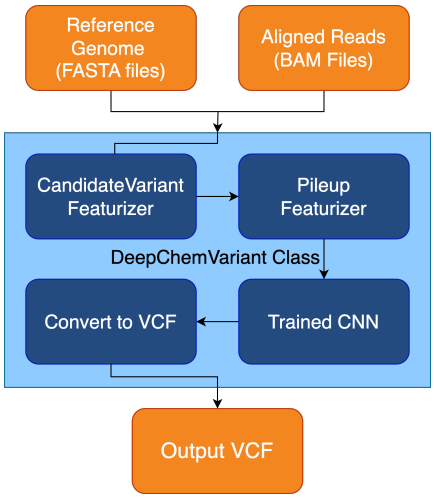

Figure 2. DeepChem-Variant workflow: Reference genome (FASTA) and aligned reads (BAM) feed into the DeepVariant class containing candidate variant featurizer, pileup featurizer, trained CNN (MobileNetV2 or InceptionV3), and VCF converter to produce output Variant Call Format (VCF) files.

DeepChem-Variant has three primary components: candidate variant detection, pileup image generation, and a deep learning model designed for variant calling. These components are implemented through modular featurizers and a custom convolutional neural network (CNN). To efficiently handle various sequence alignment formats, we developed specialized utility classes including `BAMLoader`, `SAMLoader`, and `CRAMLoader`, enabling seamless integration with diverse genomic datasets. We also enhanced the `FASTALoader` for efficient reference genome access and significantly optimized the `FASTAFeaturizer` to process raw nucleotide sequences directly, rather than converting to one-hot encoded representations, resulting in a 210-fold acceleration in processing speed.

2.1. Candidate Variant Detection

The first stage involves realigning input sequencing reads and identifying candidate variants through the `CandidateFeaturizer` class. Reads are provided in BAM(Li et al., 2009) format, storing compressed alignments of sequencing reads to a reference genome. The featurizer supports optional realignment for improved variant detection accuracy, optional multiprocessing (achieving 8× speedup), and optional labeling for training purposes when VCF ground truth is provided.

The realignment process introduces haplotype awareness through realignment using the `realign_read` method, using Striped Smith Waterman algorithm (Zhao et al., 2013) to improve read positioning. The `count_alleles` method then tallies base occurrences at each position from aligned reads, accounting for CIGAR operations (which encode how sequencing reads align to the reference genome, indicating matches, insertions, deletions, and other alignment operations) to handle insertions, deletions, and matches. The `detect_candidates` method identifies potential variants by comparing observed bases against the reference, flagging positions where alternative alleles exceed the minimum count and frequency thresholds. Finally, the `left_align_indel` method standardizes indel representations by shifting them to their leftmost valid positions, ensuring consistent variant notation across samples.

An optimized Smith-Waterman alignment algorithm performs realignment using PyTorch's GPU-accelerated tensor operations. Unlike traditional optimized implementations in C/C++, this approach exploits Python's high-level interface while maintaining competitive performance through PyTorch's CUDA backend. Alignment scores are computed using vectorized operations with substitution scores derived from binary match/mismatch masks.

Left-alignment of indels proves critical for variant standardization. Without left-alignment, candidate detection yielded inconsistent results across samples (average: 342,536 variants), while left-alignment produced 1,727,087 average standardized candidates, demonstrating the importance of variant normalization.

The `min_count` parameter, which is the minimum number of reads that must support an alternate allele for it to be considered a candidate variant, significantly impacts candidate sensitivity and computational efficiency. It helps filter out sequencing errors and reduces false positives by requiring multiple independent observations of the same change.

The default `min_count=2` provides optimal balance, as reducing to 1 substantially increases downstream processing without meaningful recall improvement. This stage is implemented using the pysam (Gilman et al., 2019) library for efficient BAM and FASTA file manipulation.

Table 1. Impact of min_count parameter on candidate detection using HG004 (GIAB consortium data, using Novaseq whole exome sequencing with IDT capture at 100x coverage)

min_count	Total Candidates	Computational Impact
1	8,420,830	High downstream execution time
2 (default)	1,770,108	Optimal balance

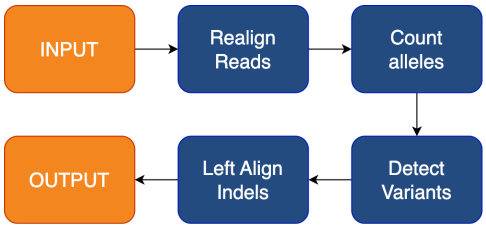

Figure 3. `CandidateFeaturizer` workflow: Input data flows through realignment of reads, allele counting at genomic positions, variant detection based on frequency thresholds, and left-alignment of indels to produce standardized candidate variants as output.

2.2. Pileup Image Generation

Once candidate variants have been identified, the next stage involves generating pileup images, a core feature of DeepVariant. A pileup image represents aligned sequencing reads at a genomic region centered on a candidate variant position. Each row corresponds to a read, and each column represents a genomic coordinate within the pileup window.

The `PileupFeaturizer` is responsible for creating these images, with six channels encoding different features of the sequencing data. Channel 0 encodes base intensities, while Channel 1 captures base quality information. Channel 2 encodes mapping quality, and Channel 3 represents the strand orientation (i.e., whether the read is from the forward or reverse strand). Channel 4 indicates whether the read supports a variant, and Channel 5 encodes the difference between the read and the reference sequence. These multi-channel images provide a rich representation of the underlying sequencing data, enhancing the ability of the deep learning model to distinguish between true variants and sequencing artifacts.

2.3. Modified CNN for Variant Calling

The deep learning model employed in this workflow is a custom CNN, derived from either the (Szegedy et al., 2015) architecture or MobileNetV2 (Sandler et al., 2018), which is specifically tailored for genomic variant calling from pileup images. We integrated both Inception V3 and MobileNetV2 architectures into DeepChem's core model library. The Inception V3 model's convolutional layers are modified to handle the six-channel input format, while our MobileNetV2 implementation leverages its efficient inverted residual structure and linear bottlenecks for improved computational efficiency without sacrificing accuracy. Both models output a probability score for each candidate variant, indicating whether it is a true variant or a sequencing error. The models integrate into DeepChem's model library, allowing users to easily swap between Inception V3 and MobileNetV2 implementations if desired or integrate variant calling into larger machine learning workflows, such as multi-task learning frameworks or pipelines incorporating other types of genomic data.

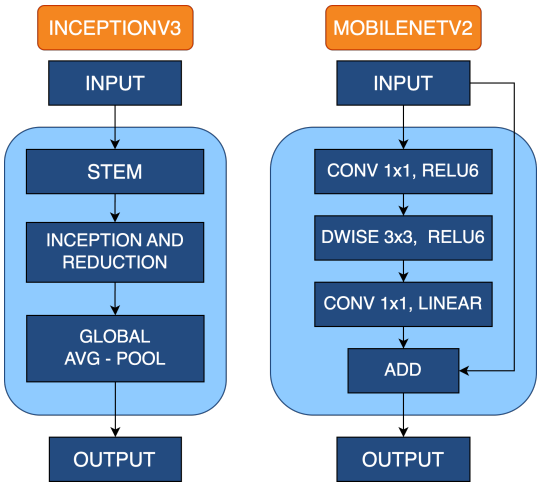

Figure 4. Architectures of Inception V3 and MobileNetV2 as used in DeepVariant.

3. Case Studies

We validated DeepChem-Variant across three genomic contexts using DeepVariant as baseline, since no established ground truth exists for these specialized datasets. DeepChem-Variant values in Table 2 represent the subset of DeepVariant calls that our method successfully detected. Sensitivity measures DeepChem-Variant's ability to recover DeepVariant's calls, calculated as the percentage of DeepVariant variants successfully identified by DeepChem-Variant. VCF outputs from both methods were compared by intersecting variant positions and alleles between them.

CRISPR Off-Target Detection: Synthetic datasets simulated CRISPR-Cas9 (Jinek et al., 2012) off-target effects at PAM sites (protospacer adjacent motifs) with insertion, deletion, and random edit patterns across NNNNGATT and NGG motifs. Off-target detection is critical for CRISPR therapeutic safety, as unintended edits can cause harmful mutations.

Ancient DNA Analysis: Simulated characteristic ancient DNA damage patterns including C→T transitions, fragmentation, and age-dependent preservation effects spanning 100 to 50,000 years. Ancient DNA analysis enables evolutionary

studies and population history reconstruction but requires specialized variant calling due to extreme degradation.

Population Genomics: Analyzed whole exome sequencing from adult female, adult male, and pediatric male samples to assess demographic-specific variant detection. Population genomics applications require consistent variant calling across diverse samples for disease association studies and personalized medicine.

Table 2. Variant detection sensitivity across genomic contexts

Context	Sample/Type	DeepVariant	DeepChem-Variant	Sensitivity (%)
CRISPR	NNNNGATT Insertion	2103	1717	81.6
	NNNNGATT Deletion	2284	1819	79.6
	NNNNGATT Random	2051	1623	79.1
	NGG Insertion	284	251	88.4
	NGG Deletion	312	282	90.4
	NGG Random	298	274	91.9
Ancient DNA	Recent (100y)	32,847	30,942	94.2
	Medieval (800y)	51,293	47,251	92.1
	Neanderthal (50,000y)	78,164	68,503	87.6
Population	Adult Female	61,245	55,732	91.0
	Adult Male	59,874	54,605	91.2
	Pediatric Male	60,298	54,992	91.2

4. Discussion

DeepChem-Variant marks a progression in the development of open-source tools for genomic variant calling. By embedding advanced deep learning techniques within a flexible machine learning framework, this integration improves accessibility and customizability of variant calling infrastructure for a wide range of research applications. DeepChem's modular architecture enables easy adaptation, allowing researchers to explore new methodologies, within genomic data analysis workflows.

4.1. Performance and efficiency

The original DeepVariant combined C++ and Python, creating complexity requiring proficiency in both languages. DeepChem-Variant's all-Python implementation (approximately 1,500 lines versus DeepVariant's 35,000 as mentioned in Table 3) simplifies development, reduces barriers to entry, and leverages Python's scientific computing ecosystem for easier extensibility and rapid prototyping. DeepChem-Variant offers two CNN architectures: Inception V3 and MobileNetV2 Our models were trained on HG001, HG002, HG004, and HG005 WES deduplicated samples at 100x coverage from IDT NovaSeq (300 million rows), compared to production DeepVariant models trained on 8-9 fold larger multi-technology GIAB datasets (2.6 billion rows) (Zook et al., 2016) and all models were validated on HG003. The code was implemented in PyTorch (Paszke et al., 2019) on Google Colab (Colab). More details about hyperparameters and system specifications are mentioned in Appendix B, details about datasets are mentioned in Appendix C.

Table 3. Comparison of lines of code between DeepVariant and DeepVariant (DeepChem)

Method	Language(s)	Lines of Code (approx.)
DeepVariant	C++/Python	35,000
DeepChem-Variant	Python	1,500

Table 4. Performance comparison of variant calling methods

Method	Variant Type	Recall	Precision	F1-Score
DeepVariant	INDEL	0.971	0.993	0.982
DeepVariant	SNP	0.988	0.998	0.993
DeepChem-Variant (MobileNetV2)	INDEL	0.912	0.934	0.923
DeepChem-Variant (MobileNetV2)	SNP	0.922	0.941	0.943
DeepChem-Variant (InceptionV3)	INDEL	0.923	0.951	0.933
DeepChem-Variant (InceptionV3)	SNP	0.931	0.954	0.939

4.2. Limitations and future work

An evaluation of performance metrics (Table 4) indicates some discrepancies between the original DeepVariant implementation and DeepChem-Variant. This is due to limited training data diversity in our experiments compared to production DeepVariant (discussed in section 4.1). The MobileNetV2 (3.4 million parameters) results are particularly notable given its significantly fewer parameters compared to InceptionV3 (24 million parameters) and lower ImageNet accuracy, yet achieving competitive performance in genomic variant calling. Despite these limitations, our modular open-source Python implementation allows users to easily swap components, such as the CNN architecture or realignment algorithm, as new methods and technologies emerge.

5. Conclusion

In this work, we introduce DeepChem-Variant which enables researchers to utilize advanced deep learning methods for genomics within a customizable Python framework, expanding machine learning applications in genomics. While performance differences compared to the original CNN implementation were observed, due to training on smaller datasets and architectural choices, the integration within DeepChem facilitates rapid future improvements. This will also allow for the easier incorporation of novel genomic analysis methodologies. We note that the observed performance reflects the current training data and model architecture, and leave further optimization for future work. As an open-source tool, we anticipate community contributions will drive further enhancements, ultimately benefiting areas such as personalized medicine and population genetics.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Colab, G. Google colaboratory. https://colab.research.google.com/.

Frey, N. C., Gadepally, V., and Ramsundar, B. Fastflows: Flow-based models for molecular graph generation, 2022.

Gilman, P., Janzou, S., Guittet, D., Freeman, J., DiOrio, N., Blair, N., Boyd, M., Neises, T., and Wagner, M. Pysam (python wrapper for system advisor model "sam") [swr-19-57]. *OSTI OAI (U.S. Department of Energy Office of Scientific and Technical Information)*, aug 2019. doi: 10.11578/dc.20190903.1. URL https://www.osti.gov/biblio/1559931.

Gomes, J., Ramsundar, B., Feinberg, E. N., and Pande, V. S. Atomic convolutional networks for predicting protein-ligand binding affinity, 2017.

Jinek, M., Chylinski, K., Fonfara, I., Hauer, M., Doudna, J. A., and Charpentier, E. A programmable dual-rna–guided dna endonuclease in adaptive bacterial immunity. *Science*, 337(6096):816–821, 2012. doi: 10.1126/science.1225829.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, pp. 1097–1105. Curran Associates, Inc., 2012. URL https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., and Subgroup, . G. P. D. P. The sequence alignment/map format and samtools. *bioinformatics*, 25(16):2078–2079, 2009.

McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M., et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Poplin, D. P. Deepvariant: Highly accurate genomes with deep neural networks, 2017. URL https://research.google/blog/deepvariant-highly-accurate-genomes-with-deep-neural-networks/. Google Research Blog.

Poplin, R., Chang, P.-C., Alexander, D., Schwartz, S., Colthurst, T., Ku, A., Newburger, D., Dijamco, J., Nguyen, N., Afshar, P. T., et al. A universal snp and small-indel variant caller using deep neural networks. *Nature biotechnology*, 36(10):983–987, 2018.

Ramsundar, B., Eastman, P., Walters, P., Pande, V., Leswing, K., and Wu, Z. *Deep Learning for the Life Sciences*. O'Reilly Media, 2019. https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/1492039837.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4510–4520, 2018.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision, 2015. URL https://arxiv.org/abs/1512.00567.

Wu, Z., Ramsundar, B., Feinberg, E. N., Gomes, J., Geniesse, C., Pappu, A. S., Leswing, K., and Pande, V. Moleculenet: A benchmark for molecular machine learning, 2018.

Zhao, M., Lee, W.-P., Garrison, E. P., and Marth, G. T. SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications. *PLOS ONE*, 8(12):e82138, December 2013. doi: 10.1371/journal.pone.0082138.

Zook, J. M., Catoe, D., McDaniel, J., Vang, L., Spies, N., Sidow, A., Weng, Z., Liu, Y., Mason, C. E., Alexander, N., et al. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific Data*, 3(1):160025, 2016. doi: 10.1038/sdata.2016.25.

A. Implementation Details

This project added significant functionality to DeepChem through new classes and enhancements spanning featurizers, data loaders, models, and variant calling infrastructure. Table 5 summarizes the key contributions.

Table 5. Code contributions to DeepChem framework

Class Name	Parent Class	Description
`SAMFeaturizer`	`Featurizer`	Processes SAM alignment files
`BAMFeaturizer`	`Featurizer`	Processes BAM alignment files
`CRAMFeaturizer`	`Featurizer`	Processes CRAM alignment files
`FASTAFeaturizer`	`Featurizer`	Enhanced sequence processing with 210× speedup
`SAMLoader`	`DataLoader`	Loads SAM format files
`BAMLoader`	`DataLoader`	Loads BAM format files
`CRAMLoader`	`DataLoader`	Loads CRAM format files
`FASTALoader`	`DataLoader`	Loads reference genome sequences
`MobileNetV2Model`	`TorchModel`	Efficient CNN for variant classification
`InceptionV3Model`	`TorchModel`	High-accuracy CNN for variant classification
`CandidateFeaturizer`	`Featurizer`	Identifies potential variant sites
`PileupFeaturizer`	`Featurizer`	Generates multi-channel alignment images
`DeepChemVariant`	`TorchModel`	Complete variant calling pipeline

B. Hyperparameters and System Specifications

DeepChem-Variant (both InceptionV3 and MobileNetV2) utilized a pileup image representation with a window size of 221 base pairs, capturing 100 read depths across 6 channels encoding base identity, base quality, mapping quality, strand orientation, variant support, and reference match indicators. Training was conducted using the Adam optimizer with a learning rate of 1e-3, batch size of 128, and 10 epochs. The model was trained on Google Colab's L4 GPU infrastructure with 8 data loader workers for parallel data processing. Variant candidates were filtered using minimum allele count and frequency thresholds of 2 reads and 1% respectively, ensuring adequate support for downstream classification while maintaining computational efficiency.

C. Datasets

DeepChem-Variant (both InceptionV3 and MobileNetV2) was trained on a constrained dataset comprising HG001, HG002, HG004, and HG005 WES deduplicated samples at 100x coverage from a single sequencing platform (IDT NovaSeq, 12 GB total). In contrast, production DeepVariant models utilize extensive multi-technology GIAB datasets that are 8-9 fold larger, incorporating data from diverse sequencing platforms including HiSeqX, NovaSeq, and PCR-positive samples across multiple WES capture kits, different sequencing depths, and various sample preparation methods (Zook et al., 2016). This substantial difference in training data scale, technological diversity, and sample heterogeneity results in reduced performance due to our model's limited exposure to the full spectrum of sequencing artifacts and variant patterns present in real-world genomic data.

D. Computation of Candidate Variants

D.1. CandidateVariantFeaturizer

The CandidateVariantFeaturizer algorithm processes genomic data in sliding windows to identify potential variant sites. For each genomic region, it extracts aligned reads and reference sequences, optionally performs Smith-Waterman realignment to improve accuracy, then counts allele frequencies at each position. Candidate variants are detected by comparing observed alleles against the reference using minimum count and frequency thresholds. Finally, indel variants are left-aligned to ensure standardized representation. The algorithm returns an array of candidate variants with associated metadata for downstream

analysis.

Algorithm 1 CandidateVariantFeaturizer

Input: BAM file B, FASTA file F
Output: Candidate variant array
for each region $(chrom, start, end)$ **do**
  $reads \leftarrow \texttt{fetch}(B, chrom, start, end)$ /* extract aligned reads */
  $ref\_seq \leftarrow \texttt{fetch}(F, chrom, start, end)$ /* extract reference sequence */
  if realign enabled **then**
    $reads \leftarrow \texttt{smith\_waterman\_realign}(reads, ref\_seq)$ /* improve alignment accuracy */
  end if
  $counts \leftarrow \texttt{count\_alleles}(reads, ref\_seq)$ /* tally base frequencies */
  $variants \leftarrow \texttt{detect\_candidates}(counts, \text{min\_count}, \text{min\_frac})$ /* identify variant sites */
  for each variant $v \in variants$ **do**
    $v \leftarrow \texttt{left\_align\_indel}(v)$ /* standardize representation */
    `output.append`(v) /* add to result set */
  end for
end for
Return: candidate variants with metadata

D.2. Realignment of Reads

Algorithm 2 Smith-Waterman Alignment

Input: Query sequence Q, reference sequence R
Output: Aligned query sequence
$H, E, F \leftarrow \texttt{zeros}(|Q|+1, |R|+1)$ /* alignment, gap matrices */
$pointer \leftarrow \texttt{zeros}(|Q|+1, |R|+1)$ /* traceback directions */
$max\_score, max\_pos \leftarrow 0, (0,0)$ /* track optimal alignment */
for $i = 1$ to $|Q|$ **do**
  for $j = 1$ to $|R|$ **do**
    $match \leftarrow H[i-1, j-1] + \texttt{score}(Q[i], R[j])$ /* diagonal score */
    $E[i,j] \leftarrow \max(H[i-1,j] + gap\_open, E[i-1,j] + gap\_extend)$ /* vertical gap */
    $F[i,j] \leftarrow \max(H[i,j-1] + gap\_open, F[i,j-1] + gap\_extend)$ /* horizontal gap */
    $H[i,j] \leftarrow \max(0, match, E[i,j], F[i,j])$ /* local alignment score */
    if $H[i,j] > max\_score$ **then**
      $max\_score, max\_pos \leftarrow H[i,j], (i,j)$ /* update maximum */
    end if
  end for
end for
$aligned \leftarrow \texttt{traceback}(pointer, max\_pos, Q)$ /* reconstruct alignment */
Return: $aligned$

The Smith-Waterman algorithm performs optimal local sequence alignment using dynamic programming. It initializes three scoring matrices: H for alignment scores, E and F for gap penalties. The algorithm fills these matrices by calculating match/mismatch scores and gap costs, maintaining traceback pointers for reconstruction. It identifies the maximum local alignment score during matrix filling, then traces back from this position to reconstruct the optimal alignment path. This implementation uses PyTorch tensors for vectorized operations, providing GPU acceleration while maintaining the algorithm's quadratic time complexity.

D.3. Counting Alleles

The `count_alleles` funtion tallies base frequencies at each genomic position by processing aligned reads. It initializes a dictionary array to store counts per position, then iterates through each read's CIGAR string to handle matches, insertions,

and deletions appropriately. For matched regions, it extracts base calls and increments corresponding position counters. The algorithm advances position pointers based on CIGAR operations, ensuring proper coordinate mapping between read sequences and reference positions.

Algorithm 3 Count Alleles

Input: Reads R, reference sequence ref, region start $start$
Output: Allele counts per position
$counts \leftarrow$ `empty_dict_array`$(|ref|)$ /* one dict per position */
for each read $r \in R$ **do**
  if r is unmapped or duplicate **then**
    continue /* skip low-quality reads */
  end if
  $ref\_pos, query\_pos \leftarrow r.start, 0$ /* initialize positions */
  for each $(operation, length)$ in $r.cigar$ **do**
    if $operation ==$ `MATCH` **then**
      for $i = 0$ to $length - 1$ **do**
        $pos \leftarrow ref\_pos + i - start$ /* convert to region coordinates */
        if $0 \leq pos < |ref|$ and $query\_pos + i < |r.sequence|$ **then**
          $base \leftarrow r.sequence[query\_pos + i]$ /* extract base call */
          $counts[pos][base] \leftarrow counts[pos][base] + 1$ /* increment count */
        end if
      end for
      $ref\_pos, query\_pos \leftarrow ref\_pos + length, query\_pos + length$ /* advance both */
    else if $operation ==$ `INSERTION` **then**
      $query\_pos \leftarrow query\_pos + length$ /* advance query only */
    else if $operation ==$ `DELETION` **then**
      $ref\_pos \leftarrow ref\_pos + length$ /* advance reference only */
    end if
  end for
end for
Return: $counts$

D.4. Detecting candidates

The `detect_candidates` function identifies potential variant sites by applying frequency-based filtering to allele counts. It examines each genomic position with coverage, calculates total read depth, and compares observed alleles against the reference. Variants are flagged as candidates if they differ from the reference allele and exceed both minimum count and frequency thresholds, helping reduce false positives from sequencing errors while retaining true biological variants.

D.5. Left Aligning Indels

The left align indels algorithm standardizes indel representation by shifting variants to their leftmost valid position. It first checks if the variant is a SNP (equal lengths or different first bases), returning unchanged if so. For indels, it trims common prefix and suffix sequences, then iteratively shifts the variant leftward by comparing flanking bases from the reference genome until no further movement is possible, ensuring consistent variant notation across different calling methods.

E. Pileup image generation

The `PileupFeaturizer` algorithm converts genomic variants into multi-channel images for CNN processing. It creates a 6-channel tensor where each channel captures different alignment properties: base identity (A/C/G/T encoded as intensity values), base quality scores, mapping quality, read strand direction, and matches to alternate/reference alleles. The algorithm centers a fixed-width window around each variant position, extracts aligned reads from the BAM file, sorts them by quality, and populates the image tensor row-by-row. The reference sequence occupies the bottom row with maximum quality values, while aligned reads fill remaining rows based on their CIGAR alignment coordinates. This transforms raw sequencing data

Algorithm 4 Detect Candidate Variants

Input: Allele counts $counts$, reference ref, thresholds min_count, min_frac
Output: Candidate variant list
$candidates \leftarrow$ `empty_list()` /* initialize result list */
for $i = 0$ to $|counts| - 1$ **do**
  $total \leftarrow \sum(counts[i].values())$ /* sum all allele counts */
  if $total == 0$ **then**
    continue /* skip positions with no coverage */
  end if
  $ref\_base \leftarrow ref[i]$ /* get reference allele */
  for each $(base, count)$ in $counts[i]$ **do**
    if $base \neq ref\_base$ and $count \geq min\_count$ and $\frac{count}{total} \geq min\_frac$ **then**
      $candidate \leftarrow (i, ref\_base, base, count, total)$ /* create variant record */
      $candidates$`.append(`$candidate$`)` /* add to candidates */
    end if
  end for
end for
Return: $candidates$

Algorithm 5 Left Align Indels

Input: Chromosome $chrom$, position pos, reference allele ref, alternate allele alt, FASTA $fasta$
Output: Left-aligned position and alleles
if $|ref| == |alt|$ or $ref[0] \neq alt[0]$ **then**
  Return: (pos, ref, alt) /* SNP, no alignment needed */
end if
$seq, seq\_alt, left \leftarrow ref, alt, pos$ /* initialize working variables */
while $|seq| > 1$ and $|seq\_alt| > 1$ and $seq[-1] == seq\_alt[-1]$ **do**
  $seq, seq\_alt \leftarrow seq[:-1], seq\_alt[:-1]$ /* trim common suffix */
end while
while $|seq| > 1$ and $|seq\_alt| > 1$ and $seq[0] == seq\_alt[0]$ **do**
  $seq, seq\_alt \leftarrow seq[1:], seq\_alt[1:]$ /* trim common prefix */
  $left \leftarrow left + 1$ /* adjust position */
end while
while $left > 1$ **do**
  $prev\_base \leftarrow fasta$`.fetch(`$chrom, left - 2, left - 1$`)` /* get preceding base */
  if deletion and $seq[-1] == prev\_base$ **then**
    $seq \leftarrow prev\_base + seq[:-1]$ /* shift deletion left */
    $left \leftarrow left - 1$ /* update position */
  else if insertion and $seq\_alt[-1] == prev\_base$ **then**
    $seq\_alt \leftarrow prev\_base + seq\_alt[:-1]$ /* shift insertion left */
    $left \leftarrow left - 1$ /* update position */
  else
    break /* cannot shift further */
  end if
end while
Return: $(left, seq, seq\_alt)$

into structured image format suitable for deep learning variant classification.

Algorithm 6 PileupFeaturizer

Input: BAM file B, FASTA file F, candidates C
Output: Multi-channel pileup images dataset
$n \leftarrow |C|$ /* number of candidate variants */
$X \leftarrow \texttt{zeros}(n, channels, height, window)$ /* image tensor */
$y \leftarrow \texttt{zeros}(n)$ if labeled else None /* labels if training */
for $i = 0$ to $n - 1$ **do**
  $chrom, pos, ref, alt \leftarrow C[i][0:4]$ /* extract variant info */
  $start \leftarrow pos - window/2$ /* define window boundaries */
  $end \leftarrow pos + window/2 + 1$
  $ref\_seq \leftarrow \texttt{fetch\_reference}(F, chrom, start, end)$ /* get reference */
  $reads \leftarrow \texttt{fetch\_reads}(B, chrom, start, end)$ /* get aligned reads */
  $reads \leftarrow \texttt{sort}(reads,$ by mapping quality$)$ /* prioritize high-quality reads */
  $pile \leftarrow \texttt{zeros}(channels, height, window)$ /* initialize image */
  for $col = 0$ to $window - 1$ **do**
    $pile[0, height - 1, col] \leftarrow \texttt{base\_to\_intensity}(ref\_seq[col])$ /* reference base */
    $pile[1:5, height - 1, col] \leftarrow [1.0, 1.0, 1.0, \texttt{alt\_match}]$ /* reference row */
  end for
  for $row = 0$ to $height - 2$ **do**
    $read \leftarrow reads[row]$ /* process each read */
    for each aligned position $(qpos, rpos)$ in $read$ **do**
      if $start \leq rpos < end$ **then**
        $col \leftarrow rpos - start$ /* column in image */
        $base \leftarrow read.sequence[qpos]$ /* read base */
        $pile[0, row, col] \leftarrow \texttt{base\_to\_intensity}(base)$ /* base identity */
        $pile[1, row, col] \leftarrow read.quality[qpos]/40.0$ /* base quality */
        $pile[2, row, col] \leftarrow read.mapping\_quality/60.0$ /* mapping quality */
        $pile[3, row, col] \leftarrow 0.0$ if reverse else 1.0 /* strand */
        $pile[4, row, col] \leftarrow 1.0$ if $base == alt$ else 0.0 /* alt match */
        $pile[5, row, col] \leftarrow 1.0$ if $base == ref$ else 0.0 /* ref match */
      end if
    end for
  end for
  $X[i] \leftarrow pile$ /* store completed image */
  if labeled **then**
    $y[i] \leftarrow C[i][-1]$ /* extract label if training */
  end if
end for
Return: $\texttt{NumpyDataset}(X, y)$

F. DeepChem-Variant

The DeepChem-Variant algorithm implements a complete variant calling pipeline using deep learning. It first extracts candidate variants from aligned reads using frequency thresholds, then generates multi-channel pileup images around each candidate site. These images are processed through a convolutional neural network (MobileNetV2) in batches to predict genotype probabilities (reference, heterozygous, or homozygous alternate). The algorithm computes genotype quality scores from prediction confidence, filters out reference calls, and writes remaining variants to a standard VCF file with proper formatting and metadata.

Algorithm 7 DeepChem-Variant

Input: BAM file B, FASTA file F, output VCF path O
Output: VCF file with variant calls
$candidates \leftarrow \texttt{CandidateFeaturizer}(B, F)$ /* extract potential variants */
if $candidates$ is empty **then**
  $\texttt{write\_empty\_vcf}(O, F)$ /* create empty VCF with header */
  Return: O
end if
$pileup\_images \leftarrow \texttt{PileupFeaturizer}(B, F, candidates)$ /* generate 6-channel images */
$predictions \leftarrow []$ /* initialize prediction array */
for $i = 0$ to $|pileup\_images|$ step $batch\_size$ **do**
  $batch \leftarrow pileup\_images[i : i + batch\_size]$ /* create batch */
  $batch\_preds \leftarrow \texttt{MobileNetV2}(batch)$ /* predict genotype probabilities */
  $predictions.\texttt{append}(batch\_preds)$ /* collect predictions */
end for
$all\_predictions \leftarrow \texttt{concatenate}(predictions)$ /* combine batches */
$genotypes \leftarrow \texttt{argmax}(all\_predictions)$ /* most likely genotype */
$gq \leftarrow \texttt{compute\_quality}(all\_predictions)$ /* genotype quality scores */
$\texttt{write\_vcf\_header}(O, F, sample\_name)$ /* write VCF header */
for each $(candidate_i, genotype_i, quality_i) \in \texttt{zip}(candidates, genotypes, gq)$ **do**
  if $genotype_i == 0$ **then**
    continue /* skip reference calls */
  end if
  $chrom, pos, ref, alt \leftarrow candidate_i[0:4]$ /* extract variant info */
  $gt\_string \leftarrow$ "0/1" if $genotype_i == 1$ else "1/1" /* format genotype */
  $\texttt{write\_vcf\_record}(O, chrom, pos, ref, alt, gt\_string, quality_i)$ /* write variant */
end for
Return: O

G. MobileNetV2

The complete MobileNetV2 architecture implements efficient mobile computer vision. It begins with standard convolution for initial feature extraction, then processes through configurable inverted residual blocks that balance accuracy and computational efficiency. Each block configuration [t, c, n, s] specifies expansion ratio, output channels, repetition count, and stride. Width multipliers enable scaling model capacity for different resource constraints. Final layers include high-dimensional feature mapping (1280 channels), global average pooling for spatial dimension reduction, and linear classification head.

Algorithm 8 MobileNetV2 Network

Input: Image I with C_{in} channels, input size $H \times W$
Params: Width multiplier α, class count N_{class}
$x \leftarrow \text{ConvBNReLU}(I, \text{out\_channels} = 32, \text{stride} = 2)$ /* initial feature extraction */
for each block configuration $[t, c, n, s]$ in settings **do**
  $C_{out} \leftarrow \lceil \frac{c \times \alpha}{8} \rceil \times 8$ /* apply width multiplier */
  for $i = 1$ to n **do**
    if $i = 1$ **then**
      $x \leftarrow \text{InvertedResidual}(x, C_{out}, \text{stride} = s, \text{expand\_ratio} = t)$ /* first block with stride */
    else
      $x \leftarrow \text{InvertedResidual}(x, C_{out}, \text{stride} = 1, \text{expand\_ratio} = t)$ /* subsequent blocks */
    end if
  end for
end for
$x \leftarrow \text{ConvBNReLU}(x, \text{out\_channels} = 1280)$ /* final feature mapping */
$x \leftarrow \text{Mean}(x, \text{over spatial dims})$ /* global average pooling */
$y \leftarrow \text{Linear}(x, \text{out\_features} = N_{\text{class}})$ /* classification layer */
Return: Class logits y

G.1. ConvBNReLU block

The ConvBNReLU block is a standard convolutional building block optimized for mobile inference. The 3×3 convolution extracts spatial features while batch normalization stabilizes training and inference. ReLU6 activation (min(max(0, x), 6)) provides bounded non-linearity that improves quantization precision for mobile deployment, reducing numerical precision requirements compared to unbounded ReLU.

Algorithm 9 ConvBNReLU Block

Input: Feature map x, input channels C_{in}, output channels C_{out}, stride s
$y \leftarrow \text{Conv2D}(x, \text{kernel} = 3 \times 3, \text{stride} = s, \text{padding} = 1, \text{bias} = False)$ /* convolution */
$y \leftarrow \text{BatchNorm}(y)$ /* normalize activations */
$y \leftarrow \text{ReLU6}(y)$ /* bounded activation function */
Return: y

G.2. Inverted Residual Block

The Inverted Residual block is a core MobileNetV2 innovation addressing traditional depthwise convolution limitations. The "inverted" design expands narrow input channels to higher dimensions (expansion phase), applies efficient depthwise convolution for spatial feature extraction (filtering phase), then compresses back to narrow output channels (projection phase). This pattern maintains information flow in high-dimensional space while keeping input/output narrow for efficiency. Linear bottlenecks (no activation after final projection) preserve information flow. Residual connections enable gradient flow and feature reuse when input/output dimensions align, following ResNet principles adapted for mobile efficiency.

Algorithm 10 Inverted Residual Block

Input: Feature map x, input channels C_{in}, output channels C_{out}, stride s, expand ratio t
$C_{hidden} \leftarrow C_{in} \times t$ /* calculate expanded channels */
use_residual $\leftarrow (s = 1) \wedge (C_{in} = C_{out})$ /* residual only if dimensions match */
if $t = 1$ **then**
  $y \leftarrow \text{DepthwiseConv}(x, \text{kernel} = 3 \times 3, \text{stride} = s)$ /* no expansion needed */
  $y \leftarrow \text{BatchNorm}(y)$ /* normalize */
  $y \leftarrow \text{ReLU6}(y)$ /* activate */
  $y \leftarrow \text{PointwiseConv}(y, C_{out})$ /* project to output channels */
  $y \leftarrow \text{BatchNorm}(y)$ /* normalize projection */
else
  $y \leftarrow \text{PointwiseConv}(x, C_{hidden})$ /* expand channels */
  $y \leftarrow \text{BatchNorm}(y)$ /* normalize expansion */
  $y \leftarrow \text{ReLU6}(y)$ /* activate expanded features */
  $y \leftarrow \text{DepthwiseConv}(y, \text{kernel} = 3 \times 3, \text{stride} = s)$ /* spatial filtering */
  $y \leftarrow \text{BatchNorm}(y)$ /* normalize filtering */
  $y \leftarrow \text{ReLU6}(y)$ /* activate filtered features */
  $y \leftarrow \text{PointwiseConv}(y, C_{out})$ /* project to output */
  $y \leftarrow \text{BatchNorm}(y)$ /* normalize final projection */
end if
if use_residual **then**
  $y \leftarrow x + y$ /* add residual connection */
end if
Return: y

H. InceptionV3

InceptionV3 architecture implements the network-in-network design philosophy, where convolutional filters of various sizes (1×1, 3×3, 5×5) are applied in parallel within each module to capture features at multiple scales. The architecture systematically processes images through a hierarchical feature extraction pipeline that begins with stem convolutions for low-level feature extraction, progresses through InceptionA modules for multi-scale pattern recognition, utilizes reduction modules to compress spatial dimensions while expanding channel depth, employs InceptionC modules with factorized 7×7 convolutions for efficient mid-level feature processing, and concludes with InceptionE modules for high-level abstraction. The network incorporates factorized convolutions that break down larger convolutions into smaller, more efficient sequences (such as decomposing 3×3 into 1×3 and 3×1), significantly reducing computational complexity while maintaining representational power. Auxiliary classifiers are strategically placed to provide intermediate supervision during training, acting as regularizers that combat vanishing gradients in deep networks. Dimensionality reduction techniques are employed throughout to control computational complexity without sacrificing model expressiveness.

H.1. BasicConv2d

This fundamental building block combines three essential operations: convolution for spatial feature extraction, batch normalization for training stability and gradient flow optimization, and ReLU activation for introducing non-linearity while preserving gradient propagation. The bias-free convolution design leverages batch normalization's inherent bias handling, reducing parameter redundancy.

H.2. InceptionA

This multi-scale feature extraction module implements parallel processing branches with different receptive fields to capture diverse spatial patterns. The 1×1 branch captures point-wise features and cross-channel correlations, the 5×5 branch (preceded by 1×1 reduction) captures medium-scale spatial patterns, the double 3×3 branch efficiently approximates larger receptive fields while reducing computational cost, and the pooling branch preserves existing feature representations. Feature concatenation combines these diverse representations into a unified output tensor.

Algorithm 11 InceptionV3

Input: Image x with C_{in} channels, size 299×299
Output: Class logits and auxiliary logits (if training)
$x \leftarrow$ Conv2d_1a_3x3(x) /* initial stem convolution */
$x \leftarrow$ Conv2d_2a_3x3(x) /* stem progression */
$x \leftarrow$ Conv2d_2b_3x3(x) /* stem completion */
$x \leftarrow$ MaxPool2d($x, 3, 2$) /* spatial downsampling */
$x \leftarrow$ Conv2d_3b_1x1(x) /* channel reduction */
$x \leftarrow$ Conv2d_4a_3x3(x) /* feature extraction */
$x \leftarrow$ MaxPool2d($x, 3, 2$) /* spatial downsampling */
for $i = 5b$ to $5d$ **do**
  $x \leftarrow$ InceptionA(x) /* parallel multi-scale convolutions */
end for
$x \leftarrow$ InceptionB(x) /* reduction with stride 2 */
for $i = 6b$ to $6e$ **do**
  $x \leftarrow$ InceptionC(x) /* factorized 7×7 convolutions */
end for
if training and aux_logits **then**
  $aux \leftarrow$ InceptionAux(x) /* auxiliary classifier */
end if
$x \leftarrow$ InceptionD(x) /* reduction with stride 2 */
for $i = 7b$ to $7c$ **do**
  $x \leftarrow$ InceptionE(x) /* high-level feature extraction */
end for
$x \leftarrow$ AdaptiveAvgPool2d($x, (1, 1)$) /* global pooling */
$x \leftarrow$ Flatten(x) /* vectorize features */
$x \leftarrow$ Dropout(x) /* regularization */
$x \leftarrow$ Linear($x, num\_classes$) /* classification head */
Return: x (and aux if training)

Algorithm 12 BasicConv2d

Input: Feature map x, output channels C_{out}, kernel params
$x \leftarrow$ Conv2d(x, C_{out}, bias=False) /* convolution without bias */
$x \leftarrow$ BatchNorm2d(x) /* normalize activations */
$x \leftarrow$ ReLU(x) /* non-linear activation */
Return: x

Algorithm 13 InceptionA Module

Input: Feature map x, pool features count
$branch_{11} \leftarrow$ BasicConv2d($x, 64, 11$) /* direct 1×1 path */
$branch_{55} \leftarrow$ BasicConv2d($x, 48, 11$) /* 5×5 reduction */
$branch_{55} \leftarrow$ BasicConv2d($branch_{55}, 64, 55$) /* 5×5 convolution */
$branch_{33} \leftarrow$ BasicConv2d($x, 64, 11$) /* double 3×3 reduction */
$branch_{33} \leftarrow$ BasicConv2d($branch_{33}, 96, 33$) /* first 3×3 */
$branch_{33} \leftarrow$ BasicConv2d($branch_{33}, 96, 33$) /* second 3×3 */
$branch_{pool} \leftarrow$ AvgPool2d($x, 3, 1, 1$) /* pooling path */
$branch_{pool} \leftarrow$ BasicConv2d($branch_{pool}, pool\_features, 11$) /* pool projection */
$output \leftarrow$ Concatenate($[branch_{11}, branch_{55}, branch_{33}, branch_{pool}]$) /* combine paths */
Return: $output$

H.3. InceptionB

This architectural transition module reduces spatial resolution from 35×35 to 17×17 while expanding channel depth from 288 to 768. Multiple reduction paths maintain feature diversity during downsampling: direct 3×3 convolution with stride-2 for efficient reduction, double 3×3 path for complex pattern preservation, and max pooling for spatial downsampling. The increased channel count compensates for spatial information loss.

Algorithm 14 InceptionB Module

Input: Feature map x
$branch_{33} \leftarrow$ BasicConv2d$(x, 384, 33, \text{stride=2})$ /* direct reduction */
$branch_{dbl} \leftarrow$ BasicConv2d$(x, 64, 11)$ /* double 3×3 path */
$branch_{dbl} \leftarrow$ BasicConv2d$(branch_{dbl}, 96, 33)$ /* expand channels */
$branch_{dbl} \leftarrow$ BasicConv2d$(branch_{dbl}, 96, 33, \text{stride=2})$ /* reduce spatial */
$branch_{pool} \leftarrow$ MaxPool2d$(x, 3, 2)$ /* pooling reduction */
$output \leftarrow$ Concatenate$([branch_{33}, branch_{dbl}, branch_{pool}])$ /* combine reductions */
Return: $output$

H.4. InceptionC

This computational optimization module factorizes expensive 7×7 convolutions into more efficient asymmetric sequences. The 1×7 followed by 7×1 factorization reduces parameters from 49 to 14 while maintaining equivalent receptive field coverage. The double factorization path provides additional feature diversity through repeated asymmetric convolutions, enabling complex pattern recognition with reduced computational overhead.

Algorithm 15 InceptionC Module

Input: Feature map x, 7×7 channel count
$branch_{11} \leftarrow$ BasicConv2d$(x, 192, 11)$ /* direct path */
$branch_{77} \leftarrow$ BasicConv2d$(x, channels_{77}, 11)$ /* 7×7 factorization */
$branch_{77} \leftarrow$ BasicConv2d$(branch_{77}, channels_{77}, 17)$ /* factorize to 1×7 */
$branch_{77} \leftarrow$ BasicConv2d$(branch_{77}, 192, 71)$ /* factorize to 7×1 */
$branch_{dbl} \leftarrow$ BasicConv2d$(x, channels_{77}, 11)$ /* double 7×7 path */
$branch_{dbl} \leftarrow$ BasicConv2d$(branch_{dbl}, channels_{77}, 71)$ /* first factorization */
$branch_{dbl} \leftarrow$ BasicConv2d$(branch_{dbl}, channels_{77}, 17)$ /* second factorization */
$branch_{dbl} \leftarrow$ BasicConv2d$(branch_{dbl}, channels_{77}, 71)$ /* third factorization */
$branch_{dbl} \leftarrow$ BasicConv2d$(branch_{dbl}, 192, 17)$ /* final factorization */
$branch_{pool} \leftarrow$ AvgPool2d$(x, 3, 1, 1)$ /* pooling path */
$branch_{pool} \leftarrow$ BasicConv2d$(branch_{pool}, 192, 11)$ /* pool projection */
$output \leftarrow$ Concatenate$([branch_{11}, branch_{77}, branch_{dbl}, branch_{pool}])$ /* combine paths */
Return: $output$

H.5. InceptionD

This second reduction stage transitions from 17×17 to 8×8 spatial resolution while expanding channels from 768 to 1280. The module combines direct 3×3 reduction for efficiency with complex 7×7×3 factorized paths that maintain information richness through sequential asymmetric convolutions followed by spatial reduction. This design preserves feature diversity while preparing for final high-level processing.

H.6. InceptionE

The module splits 3×3 convolutions into separate 1×3 and 3×1 branches, effectively doubling feature diversity by capturing horizontal and vertical patterns independently. Complex double-branch paths maximize representational capacity through parallel processing of complementary feature patterns, providing rich feature representations for final classification decisions.

Algorithm 16 InceptionD Module

Input: Feature map x
$branch_{33} \leftarrow$ BasicConv2d$(x, 192, 11)$ /* 3×3 reduction path */
$branch_{33} \leftarrow$ BasicConv2d$(branch_{33}, 320, 33, \text{stride=2})$ /* spatial reduction */
$branch_{773} \leftarrow$ BasicConv2d$(x, 192, 11)$ /* 7×7×3 path */
$branch_{773} \leftarrow$ BasicConv2d$(branch_{773}, 192, 17)$ /* factorize 1×7 */
$branch_{773} \leftarrow$ BasicConv2d$(branch_{773}, 192, 71)$ /* factorize 7×1 */
$branch_{773} \leftarrow$ BasicConv2d$(branch_{773}, 192, 33, \text{stride=2})$ /* final reduction */
$branch_{pool} \leftarrow$ MaxPool2d$(x, 3, 2)$ /* pooling reduction */
$output \leftarrow$ Concatenate$([branch_{33}, branch_{773}, branch_{pool}])$ /* combine reductions */
Return: $output$

Algorithm 17 InceptionE Module

Input: Feature map x
$branch_{11} \leftarrow$ BasicConv2d$(x, 320, 11)$ /* direct path */
$branch_{33} \leftarrow$ BasicConv2d$(x, 384, 11)$ /* 3×3 split preparation */
$branch_{33a} \leftarrow$ BasicConv2d$(branch_{33}, 384, 13)$ /* horizontal split */
$branch_{33b} \leftarrow$ BasicConv2d$(branch_{33}, 384, 31)$ /* vertical split */
$branch_{33} \leftarrow$ Concatenate$([branch_{33a}, branch_{33b}])$ /* combine splits */
$branch_{dbl} \leftarrow$ BasicConv2d$(x, 448, 11)$ /* double 3×3 path */
$branch_{dbl} \leftarrow$ BasicConv2d$(branch_{dbl}, 384, 33)$ /* expand features */
$branch_{dbla} \leftarrow$ BasicConv2d$(branch_{dbl}, 384, 13)$ /* horizontal split */
$branch_{dblb} \leftarrow$ BasicConv2d$(branch_{dbl}, 384, 31)$ /* vertical split */
$branch_{dbl} \leftarrow$ Concatenate$([branch_{dbla}, branch_{dblb}])$ /* combine splits */
$branch_{pool} \leftarrow$ AvgPool2d$(x, 3, 1, 1)$ /* pooling path */
$branch_{pool} \leftarrow$ BasicConv2d$(branch_{pool}, 192, 11)$ /* pool projection */
$output \leftarrow$ Concatenate$([branch_{11}, branch_{33}, branch_{dbl}, branch_{pool}])$ /* combine all paths */
Return: $output$

H.7. InceptionAux

This auxiliary classifier module addresses vanishing gradient problems in deep networks by providing intermediate supervision during training. Positioned at the network's midpoint, it processes intermediate features through spatial reduction, channel manipulation, and classification layers. The auxiliary loss signal improves gradient flow to earlier network layers, enhancing training convergence and preventing gradient degradation. During inference, this module is bypassed to maintain computational efficiency.

Algorithm 18 InceptionAux Module

Input: Feature map x, number of classes
$x \leftarrow$ AvgPool2d$(x, 5, 3)$ /* spatial reduction */
$x \leftarrow$ BasicConv2d$(x, 128, 11)$ /* channel reduction */
$x \leftarrow$ BasicConv2d$(x, 768, 55)$ /* feature expansion */
$x \leftarrow$ AdaptiveAvgPool2d$(x, (1, 1))$ /* global pooling */
$x \leftarrow$ Flatten(x) /* vectorize */
$x \leftarrow$ Linear$(x, num_classes)$ /* classify */
Return: x