Image from NASA:
https://commons.wikimedia.org/wiki/File:Earth_Western_Hemisphere_transparent_background.png#filelinks

Geospatial Deep Learning

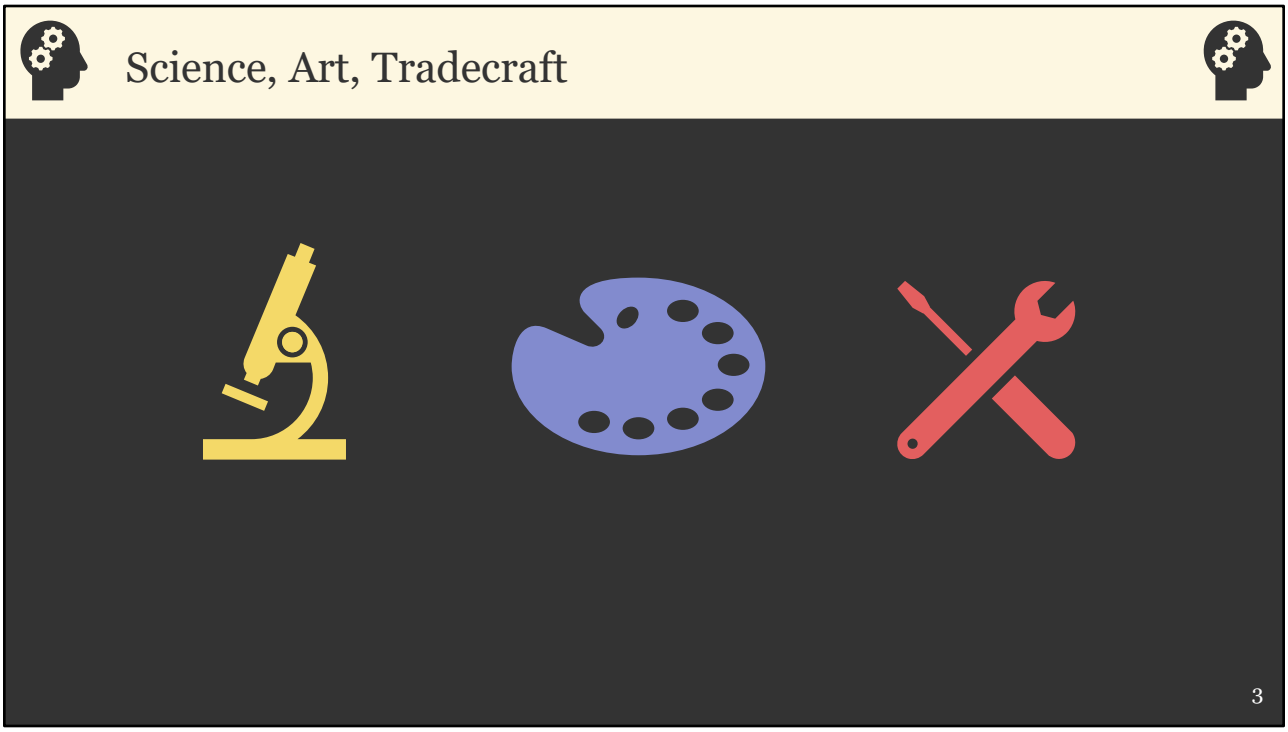
Improving Models

In this module, we will explore methods for potentially improving model performance by drawing from recent studies and best practices. Some of this material is a review from prior modules; however, I wanted to provide a single resource that covers a wide variety of means to improve models.

Module Topics

1. Using multiple GPUs
2. Reproducibility using PyTorch
3. Mini-batch size
4. Data augmentation
5. Loss metric selection and parameterization
6. Label smoothing
7. Unified focal loss framework
8. Batch normalization
9. Activation function selection
10. Dropouts
11. Residual connections
12. Dense connections
13. Atrous (dilated) convolution and the ASPP module
14. Branching
15. Feature reduction
16. Depthwise separable convolution
17. Squeeze and excitation
18. Attention gates
19. Convolution block attention module
20. Deep supervision
21. Transfer learning
22. Schedulers
23. Learning rate finder
24. Gradient accumulation
25. Gradient clipping
26. Exponential moving average
27. Mixed precision training

These are the topics covered in this module.



Deep learning is science, art, and trade knowledge. Analyst and developers often need to work with specific datasets, code libraries, and problem types in order to build up an intuition as to how best to prepare data, define architectures, and configure the learning process. Building up these skillsets often requires many hours of hands-on experience working on a variety of projects.

For deep learning specifically, how data are prepared and how the training process is configured can have a big impact on the accuracy, transferability, and generalizability of the resulting model.

If your goal is to become a well rounded geospatial deep learning practitioner, it will take time to develop this skillset, so don't get discouraged.

Multiple GPUs

- Use larger batch size
- Decrease training time
- Train on larger datasets

```
model = nn.DataParallel(model)
```

https://www.run.ai/guides/multi-gpu/pytorch-multi-gpu-4-techniques-explained

https://lambdalabs.com/

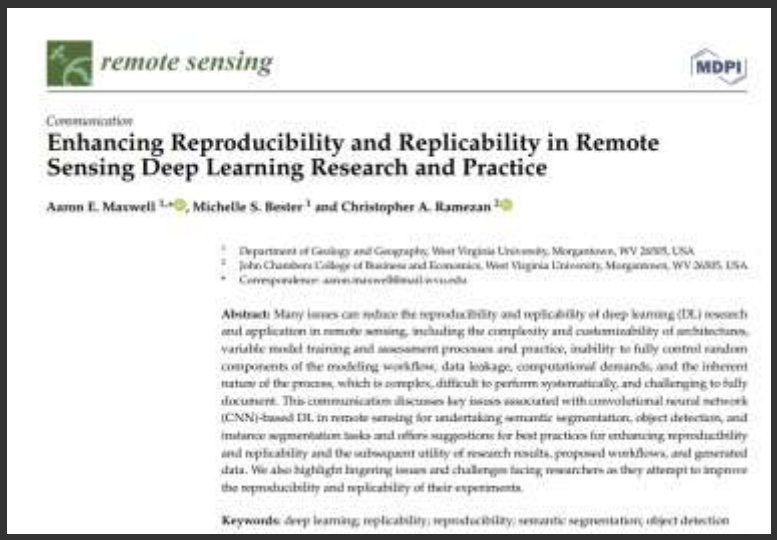
4

It is possible to train models using multiple GPUs, either within the same machine or spread across multiple machines. You can even train models using GPU clusters. This can allow for faster training and larger mini-batch sizes.

In PyTorch, this can be accomplished using nn.DataParallel().

There are some drawbacks associated with using a large mini-batch size. We will discuss this later in this module.

Reproducibility and Replicability



remote sensing

MDPI

Communication

Enhancing Reproducibility and Replicability in Remote Sensing Deep Learning Research and Practice

Aaron E. Maxwell [1,*], Michelle S. Bester [1] and Christopher A. Ramezan [2]

[1] Department of Geology and Geography, West Virginia University, Morgantown, WV 26505, USA
[2] John Chambers College of Business and Economics, West Virginia University, Morgantown, WV 26505, USA
* Correspondence: aaron.maxwell@mail.wvu.edu

Abstract: Many issues can reduce the reproducibility and replicability of deep learning (DL) research and application in remote sensing, including the complexity and customizability of architectures, variable model training and assessment processes and practice, inability to fully control random components of the modeling workflow, data leakage, computational demands, and the inherent nature of the process, which is complex, difficult to perform systematically, and challenging to fully document. This communication discusses key issues associated with convolutional neural network (CNN)-based DL in remote sensing for undertaking semantic segmentation, object detection, and instance segmentation tasks and offers suggestions for best practices for enhancing reproducibility and replicability and the subsequent utility of research results, proposed workflows, and generated data. We also highlight lingering issues and challenges facing researchers as they attempt to improve the reproducibility and replicability of their experiments.

Keywords: deep learning; replicability; reproducibility; semantic segmentation; object detection

Maxwell, A.E., Bester, M.S. and Ramezan, C.A., 2022. Enhancing Reproducibility and Replicability in Remote Sensing Deep Learning Research and Practice. *Remote Sensing*, *14*(22), p.5760.

5

This paper, which is free and available in open access, discusses issues of reproducibility and replicability in remote sensing deep learning research and practice.

Reproducibility and Replicability

Engineering, M. and National Academies of Sciences, Engineering, and Medicine, 2019. Reproducibility and replicability in science.

- Reproducibility relates to "computational reproducibility—obtaining consistent results using the same input data, computational methods, and conditions of analysis."
- Replicability relates to "whether applying the same methods to the same scientific question produces similar results"

6

This slide explains the difference between reproducibility and replicability. To increase the value of scientific research, we should strive for enhanced reproducibility and replicability. One component of this is making use of open-source software tools and making code and data readily available for use by others.

Reproducibility in PyTorch

```
def set_seed2(seed=2019):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.random.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmark=False
```

Alahmari, S.S., Goldgof, D.B., Mouton, P.R. and Hall, L.O., 2020. Challenges for the repeatability of deep learning models. *IEEE Access*, *8*, pp.211860-211868.

7

Due to the complexity of neural network architectures and their implementations, obtaining reproducible results, where the same model can be obtained repeatedly, is not simple. The paper referenced on this slide provides code that sets multiple random seeds and other setting that should allow for reproducible results using PyTorch.

Input Data Considerations

Developing Training Data

- Spatially explicit
- Representative of the population
- Adequate number of samples
- Adequate number of samples per category
- Accurate

9

This slide reviews key considerations for developing training samples. In my opinion, other than the inherent difficulty of the classification task being undertaken, the quality and characteristics of the training samples are the most important factors in model performance.

Supervised learning requires training samples. These samples represent examples of the classes or values you are trying to predict.

The algorithm will then use these examples and the predictor variables at these locations to make a model that can be used to predict new locations.

When creating training samples, they need to be spatial explicit, or you need to know where they exist in the map space.

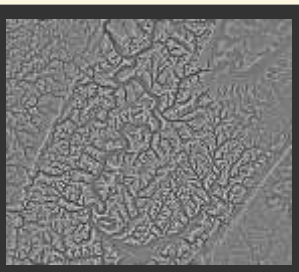
They also must be representative of the population. For example, if you are trying to predict where a certain tree might grow and it is known to grow on both ridges and in floodplains, then you need to give the algorithm examples of known locations where the tree has been found to grow in both floodplains and along ridges.

You need to provide an adequate number of samples and an adequate number of samples per class. The number of samples required will vary based on the

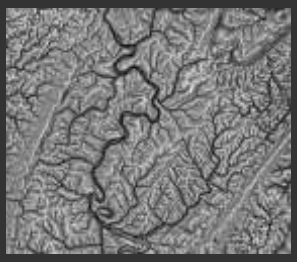
complexity of the problem and the methods used. I generally try to provide as many quality training samples as possible given limitations and constraints. Collecting a large number of training samples can be difficult due to time, cost, and access constraints.

Lastly, the data should be accurate. If the algorithm is given mislabeled or poor examples, it will have difficulty creating a useful model. Garbage in, garbage out.

Developing Predictor Variables



Slope Position



Dissection

Roughness

- Spatially explicit
- Accurately georeferenced
- Accurate/precise
- Timely
- Adequate spatial resolution
- Available

10

It is important to use predictor variables that characterize and differentiate the classes of interest. If the provided predictor variables, or feature space, is inadequate for the task at hand, it is unrealistic to expect accurate results even when using a powerful algorithm.

One issue I commonly see in remote sensing is using spectral bands, such as those provided by the instruments onboard the Landsat satellites, to attempt to map a variable or differentiate classes that are not characterized by spectral signatures. In such cases, the spectral data are generally being used because they are available and allow for predictions to be made at every pixel location since the imagery can be acquired over large spatial extents. However, if spectral information is not helpful for the task at hand, this is fundamentally flawed.

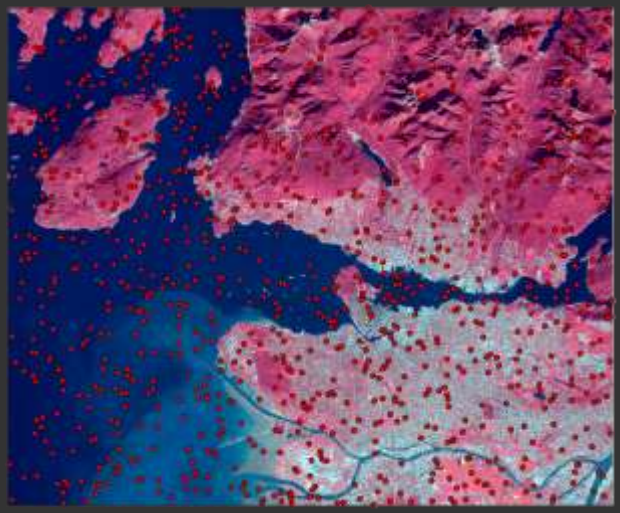
It would make sense to provide as much information or as many predictor variables as possible. For example, if you are buying a car, more information will make you a more informed consumer. However, this has generally been found to not be the case in predictive modeling. This is known as the Hughes Phenomenon or “Curse of Dimensionality”. This suggests that adding more predictor variables can actually decrease the performance of the model. This is because, even though more information is being provided, the complexity of the problem increases. This problem is generally more pronounced when a

small training set is used. This is because more samples will need to be provided to deal with the complex dimensionality of the problem. Fortunately, some algorithms are fairly robust to this problem. Also, methods are available to reduce the number of variables or select the most useful variables. This is outside the scope of this course.

Deep learning algorithms are generally not provided a large number of predictor variables or a large feature space. Instead, the deep network is expected to learn data abstractions from the input predictor variables.

Developing Testing or Validation Data

- Unbiased
- Randomized
- Non-overlapping with your training data
- Correctly proportioned
- Accurate



11

Model assessment can be misleading if the validation data are biased or otherwise inadequate. Remember that validation data refer to withheld samples used to assess the model at the end of each training epoch while testing data are used to assess the final model.

In order to produce an unbiased estimate of the performance, the testing and validation data must be randomized in some way. Also, the training and validation data should not overlap with each other or the testing samples, or the same samples should not be included for both training and assessment. This is because algorithms tend to do a better job of predicting the training samples as opposed to new locations, a phenomenon known as overfitting. So, including training samples as testing or validation samples could inflate the reported performance.

It is also recommended that testing and validation samples be correctly proportioned relative to the map. For example, if you are mapping land cover and 70% of the mapped area is forest, then 70% of the testing and validation samples should also be forest.

Lastly, testing and validation data should be accurate. The goal here is to compare the map product to reference data of higher quality. It is generally assumed that no data are perfect. So, even the testing and validation data will

have some error. We try to avoid using the terms “ground truth” or “ground truthing” for this reason.

Developing Training, Testing, or Validation Data

- Limited sample size is a common issue
- DL algorithms are prone to overfitting to small training sets
- Obtaining labeled data is a key bottleneck for supervised learning
- Data are not always well documented
- All data have uncertainty/errors (no such thing as "ground truth")
- Data may not be representative of the population of interest
- Data may be biased
- Some features are inherently difficult to map
- Boundaries can be fuzzy or gradational
- Data can be outdated or not temporally aligned with predictor variables

12

This slide summarizes some key points or considerations for developing training, testing, and validation data. Again, the characteristics of these dataset are very important for obtaining accurate, unbiased, and useful models and associated assessments.

Mini-Batch Size

- Large mini-batch size can result in reduced model generalization performance, as measured by predicting to the withheld validation data ("generalization gap")
- He et al. (2019): ratio of the mini-batch size to the learning rate has a negative correlation with generalization ability; if a large mini-batch size is used, you should consider using a larger learning rate
- Masters and Luschi (2018): increasing the mini-batch size results in decreasing the range of learning rates that support a stable training process; thus, choosing a learning rate may be of particular importance with larger mini-batch sizes
- Keskar et al. (2017): first trained with a small mini-batch size followed by training on a larger mini-batch size; or gradually increase the mini-batch size throughout the learning process

He, F., Liu, T., Tao, D., 2019. Control batch size and learning rate to generalize well: Theoretical and empirical evidence. Advances in neural information processing systems 32.

Masters, D., Luschi, C., 2018. Revisiting small batch training for deep neural networks. arXiv preprint arXiv:1804.07612.

Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P., 2016. On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836.

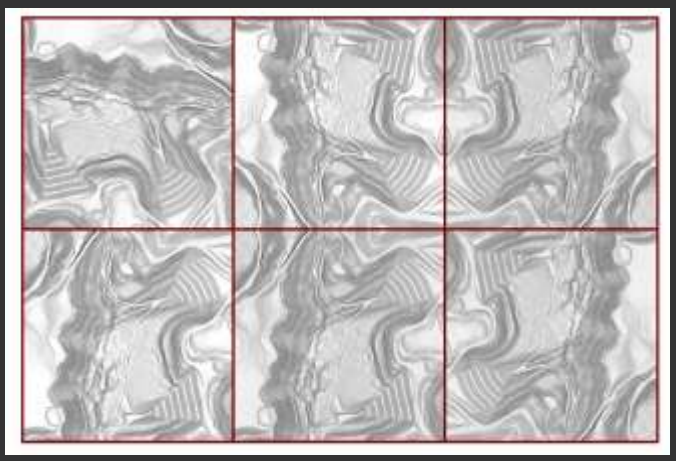
13

Remember that deep learning algorithms are commonly trained using mini-batch gradient descent or one of its derivatives (e.g., RMSProp or Adam). Parameter updates are made after processing each mini-batch as opposed to after processing all of the training data, or at the end of one training epoch.

Studies have documented that using a large mini-batch size can be problematic. Key issues and associated references are provided on this slide. As a result, I generally recommend not using large mini-batch sizes. If a large mini-batch size is used, the choice of a base learning rate and/or use of learning rate schedulers may have a larger impact on your learning process than if a smaller mini-batch size is used. In other words, there tends to be more sensitivity to learning rates when a larger mini-batch size is used.

Some studies have suggested modifying the mini-batch size during the training process (for example, Keskar et al. (2017)).

Data Augmentation



14

As mentioned in prior modules, data augmentation is often used to combat overfitting and improve model generalization and transferability. When defining data transformations, you have control over what transformations to consider (e.g., flips, rotations, contrast changes, and blurring), the probability of applying the transformation, and the degree of alteration (e.g., maximum allowed changes to contrast or blur).

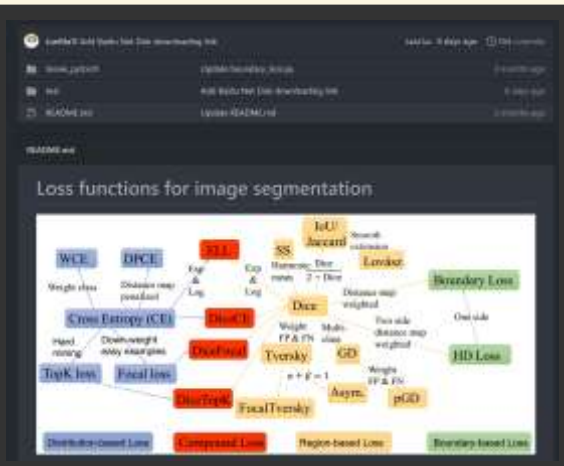
You may want to experiment with different augmentation techniques to assess how model performance is impacted.

I generally used conservative modifications/augmentations; however, others may disagree.

Another key point is that some augmentations are not appropriate for all data. For example, hue and saturation changes are only applicable to RGB data.

Loss Metrics

Choosing a Loss Metric



- Problem type
- Class imbalance
- Class weights

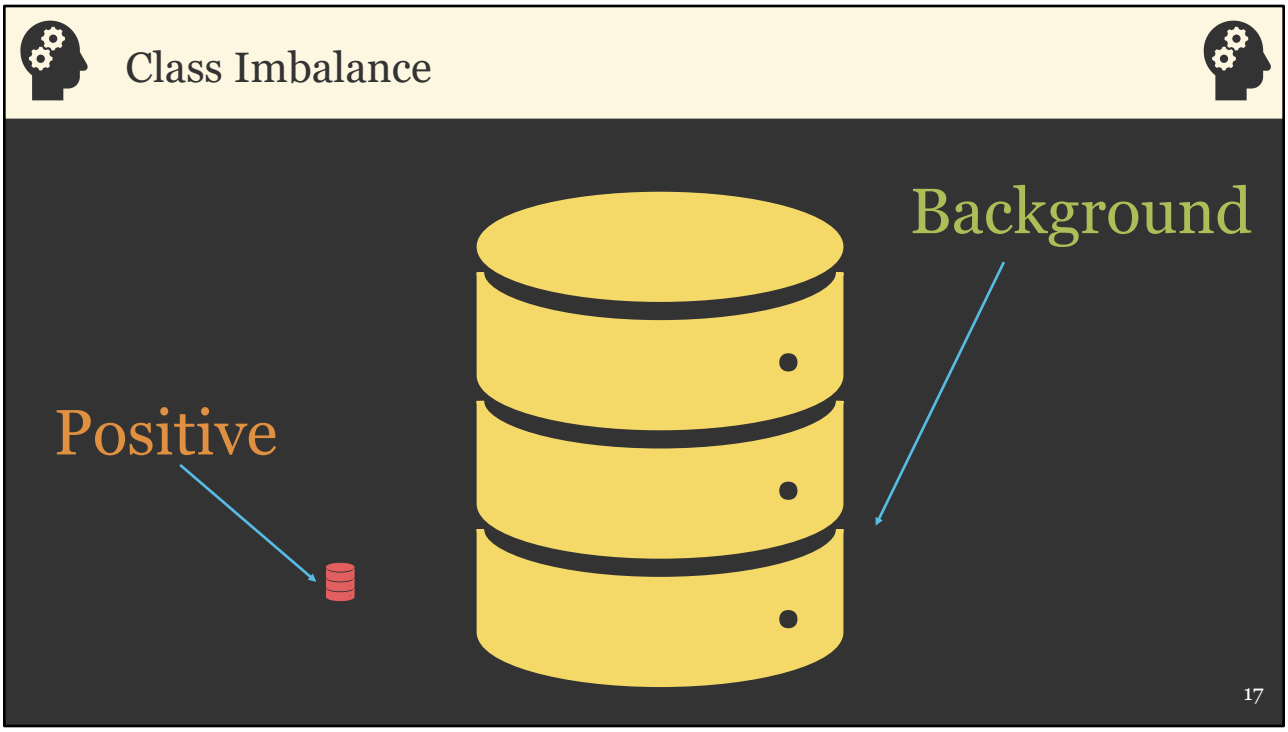
Ma J, Chen J, Ng M, Huang R, Li Y, Li C, Yang X, Martel AL. Loss odyssey in medical image segmentation. *Med Image Anal.* 2021 Jul;71:102035. doi: 10.1016/j.media.2021.102035. Epub 2021 Mar 19. PMID: 33813286.

16

Choosing a loss metric is very important. As discussed in the prior modules, the appropriate loss metric will partially depend on the problem type. For example, binary cross entropy is appropriate for a two-class problem where a single logit is returned whereas cross entropy is used when two or more logits are returned. Mean square error is appropriate for regression whereas Dice loss is used for classification.

Another key consideration is the impact of data imbalance. For example, cross entropy loss often performs poorly when classes are imbalanced, or when there are large differences in the number of training samples per class. In such cases, it is often appropriate to apply class weightings to obtain a weighted cross entropy loss. For example, some practitioners will weight classes based on the inverse of their abundance in the training set. You could also use Dice loss, which is more robust to data imbalance.

As discussed in prior modules, you can also apply Top K or focal losses to focus on predicting difficult samples. You can even combine multiple loss metrics. For very specific use cases, you can even define custom loss measures.



Data imbalance is one of the most challenging issues during the training process. For example, it is common for the feature of interest to take up a small proportion of the landscape. For example, if you are interested in extracting ponds from aerial imagery, ponds may constitute less than 1% of the area being investigated or mapped.

For a multiclass problem, some classes often are much more abundant on the landscape than others.

In the following slides, we discuss loss metric considerations when classes are imbalanced.

Weighted CE

- Augmentation CE loss that incorporates different weights for each class
- w_j = class weightings
- Weightings inversely proportional to class abundance
- Can be useful in cases of class imbalance

$$= -\frac{1}{\sum_{i=1}^{n} w_j} \sum_{i=1}^{n} \sum_{j=1}^{C} w_j \cdot y_{ij} \cdot \log\left(\widehat{y_{ij}}\right)$$

y_{ij} = current sample for current class
$\widehat{y_{ij}}$ = softmax probability for current class
n = number of samples
C = number of classes
w_j = class weightings

18

Cross entropy loss can be negatively impacted by class imbalance. Since the more abundant classes have more samples, they will have a larger impact on the loss calculation. This can cause the model to focus more on the abundant classes and less on the less abundant classes. This can result in poorer prediction accuracies for the less abundant class, overpredicting the abundance of the more abundant class, and/or underpredicting the abundance of the less abundant classes.

One means to potentially combat this issue is to apply class weightings. It is common to weight the relative influence of each class based on the inverse of its abundance in the dataset. This can result in a more balanced influence between the classes and, potentially, improved model performance.

Dice Loss

$$\text{Binary Dice loss} = 1 - \left(\frac{(2 \times \Sigma\hat{p}_{TP}) + \varepsilon}{(2 \times \Sigma\hat{p}_{TP}) + \Sigma\hat{p}_{FN} + \Sigma\hat{p}_{FP} + \varepsilon}\right)$$

$$\text{Multiclass macro-averaged Dice loss} = \frac{1}{C}\sum_{j=1}^{C}\left(1 - \left(\frac{(2 \times \Sigma\hat{p}_{TP}) + \varepsilon}{(2 \times \Sigma\hat{p}_{TP}) + \Sigma\hat{p}_{FN} + \Sigma\hat{p}_{FP} + \varepsilon}\right)\right)$$

$$\text{Log-cosh Dice loss} = \log(\cosh(\text{Dice loss}))$$

$$\text{Weighted macro-averaged Dice loss} = \frac{1}{\sum_{j=1}^{C} w_j}\sum_{j=1}^{C} w_j\left(1 - \left(\frac{(2 \times \Sigma\hat{p}_{TP}) + \varepsilon}{(2 \times \Sigma\hat{p}_{TP}) + \Sigma\hat{p}_{FN} + \Sigma\hat{p}_{FP} + \varepsilon}\right)\right)$$

$\hat{p}_{xx}$ = softmax probability for current class
C = number of classes
ε = smoothing factor
w_j = class weightings

19

Dice-based losses are commonly used as an alternative to cross entropy-based losses when classes are imbalanced. If Dice loss is used to combat class imbalance in a multiclass classification, it is important that macro-averaging as opposed to micro-averaging is used to aggregate the class-level losses. It is also possible to specify class weighting to further control the loss implementation.

Remember that, in contrast to the Dice coefficient when used as an assessment metric, class probabilities are used as opposed to hard counts in the Dice loss metric so that it is differentiable.

Tversky Loss

- Similar to Dice loss
- Provide different weightings for FNs (α) and FPs (β)
- Or, provide different weightings for recall and precision

$$\text{Binary Tversky loss} = 1 - \left(\frac{\Sigma \hat{p}_{TP} + \varepsilon}{\Sigma \hat{p}_{TP} + \alpha \Sigma \hat{p}_{FN} + \beta \Sigma \hat{p}_{FP} + \varepsilon}\right)$$

$$\text{Multiclass Tversky loss} = \frac{1}{C} \sum_{j=1}^{C} (1 - TI_j)$$

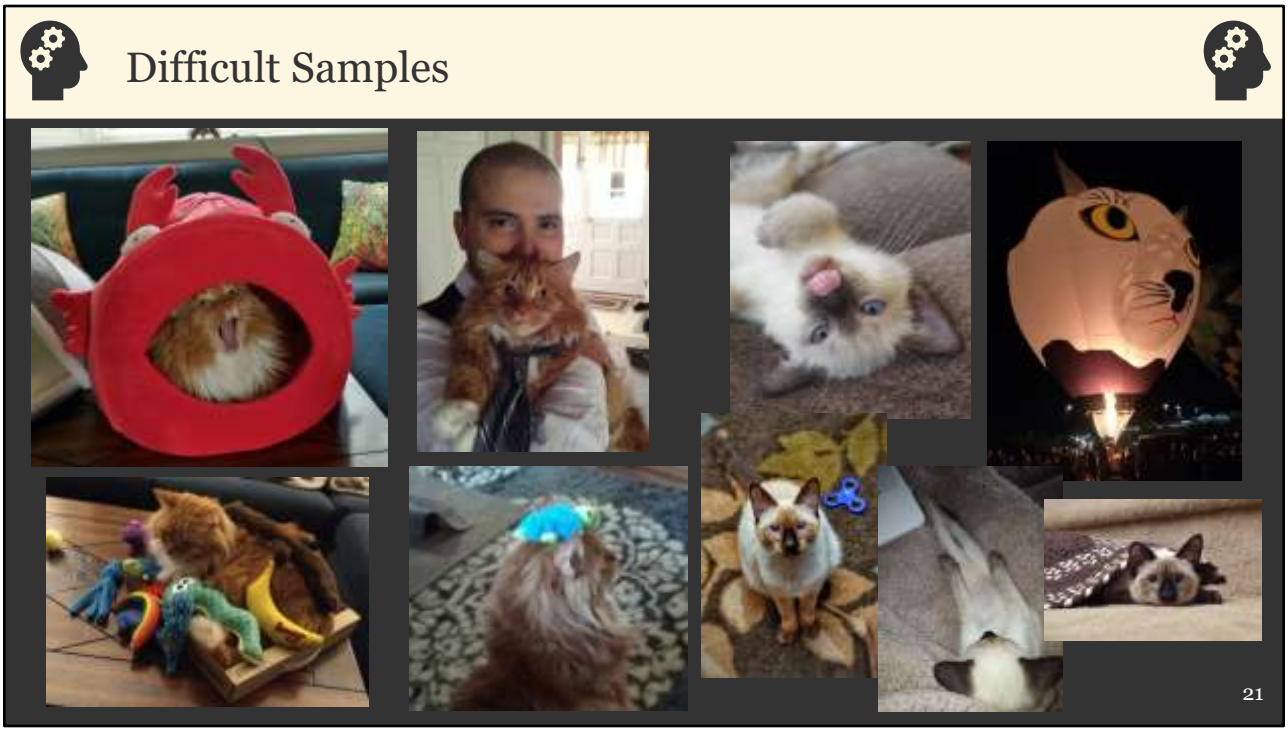
$\hat{p}_{xx}$ = softmax probability for current class
C = number of classes
ε = smoothing factor
α = coefficient for FNs
β = coefficient for FPs

20

The Tversky loss is an augmentation of the Dice loss and allows for applying different weightings to false negative (FN) and false positive (FP) errors relative to each class. In other words, different weightings can be applied to recall and precision. The key here is that the user can prioritize different types of error during model training: omission and commission errors do not need to be equally weighted. Tversky loss is also often a good choice when classes are imbalanced.

The relative weightings of FN and FP outcomes are controlled by user-defined α and β terms. α is the weighting for FNs, and β is the weighting for FPs. It has been suggested that applying these weightings can be of particular importance when classes are heavily imbalanced. Specifically, for a binary classification where the positive case is not abundant, it may be useful to use a higher α relative to β to improve recall.

The slide provides Tversky loss equations for both binary and multiclass classification.



Other than class imbalance, another issue is whether or not to weight all samples the same regardless of their level of difficulty. For example, the different pictures of my cats, or objects that look like my cats, would have varying levels of difficulty; thus, it may not be desired to treat them all equally in the loss calculations.

Focal Loss

- Focal loss allows for the prioritization of or focus on difficult-to-classify samples over multiple iterations over the training data
- Add γ (larger = more weight on difficult-to-classify samples)

$$= -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C}\left[\left(1-\widehat{y_{ij}}\right)^{\gamma}\cdot y_{ij}\cdot\log\left(\widehat{y_{ij}}\right)\right]$$

y_{ij} = current sample for current class
$\widehat{y_{ij}}$ = softmax probability for current class
n = number of samples
C = number of classes
γ = focal weightings

22

Focal losses allow for the prioritization of difficult-to-classify samples. Over multiple iterations over the training data, the weight updates will be made such that the classification of difficult samples, or those with lower probabilities of membership to the correct class, are prioritized. There are focal versions of both cross entropy and Dice/Tversky loss available.

Larger values of γ will result in putting more weight on difficult training samples. A γ of 0 equates to not implementing a focal adjustment (i.e., the result will be equivalent to CE loss).

Focal Tversky Loss

- Provide different weightings for FNs (α) and FPs (β)
- Focal loss allows for the prioritization of or focus on difficult-to-classify samples over multiple iterations over the training data (γ)
- Applied per class as opposed to per sample

$$\text{Focal multiclass Tversky loss} = \frac{1}{C}\sum_{j=1}^{C}(1 - TI_j)^{1/\gamma}$$

C = number of classes
γ = focal weighting

23

Focal Tverksy loss adds a γ parameter, which can be used to control the relative impact of difficult-to-classify samples in the aggregated loss, similar to focal CE loss.

When γ is larger than 1, difficult samples take on a larger relative weight in the loss calculation.

Top *K* Loss

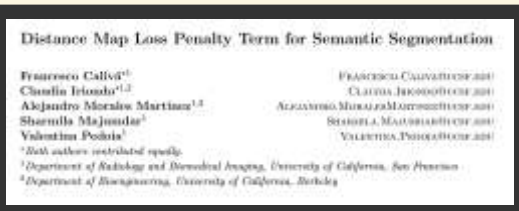
- Another way to focus on difficult-to-classify samples
- Only considers top *K* number of samples contributing to the loss
- Top *K* BCE/CE
- Top *K* Dice

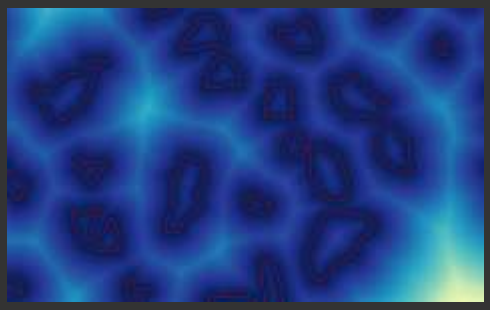
24

Another means to weight difficult-to-classify samples is to only use the difficult-to-classify samples in the calculation of the loss. In other words, the *K* number of samples that contribute most to the loss will be used to guide the weight/parameter updates. All other samples will be ignored. Similar to focal loss, there are Top K versions of cross entropy and Dice loss.

Boundaries

- Can provide higher weight to cells near class boundaries
- Assumes edge cells are more difficult to predict than interior cells


Distance Map Loss Penalty Term for Semantic Segmentation



Caliva, F., Iriondo, C., Martinez, A.M., Majumdar, S. and Pedoia, V., 2019. Distance map loss penalty term for semantic segmentation. *arXiv preprint arXiv:1908.03679*.

25

It is common for pixels or cells nearer to the margin of classes to be more difficult to classify than those nearer to the interior of classes. This is especially true when boundaries are fuzzy or uncertain.

It has been suggested that applying a higher weight to cells that are nearer to class boundaries can be used to improve model performance. This is similar to using a focal loss except that cells nearer to class margins are assumed to be more difficult to predict than those with a lower predicted probability relative to their correct class.

Implementing such a technique requires generating grids that characterize the distance from class margins. This could be accomplished using Euclidean distance, a cost-weighted Euclidean distance, or a Euclidean distance that implements inverse distance weighting.

Label Smoothing

- Hard one-hot encoded labels → soft targets
- Smoothed Label (Current Class) = (1 − α) + (α / Number of Classes)
- Smoothed Label (Other Class) = α / Number of Classes
- α = smoothing strength (0.05 to 0.1)
- If α = 0.1: [0, 1, 0, 0] → [α/4, (1−α) + (α/4) , α/4, α/4] → [0.25, 0.925, 0.25, 0.25]
- Why?
  - Model regularization/reduce overfitting
  - Prevent model from becoming overconfident in predictions
  - Smooth decision boundaries
  - May help when classes are imbalanced or training set is small

26

Label smoothing is a regularization technique that replaces hard one-hot encoded targets with soft targets, preventing the model from becoming overconfident in its predictions and encouraging smoother decision boundaries. Instead of assigning probability 1.0 to the correct class and 0.0 to all others, label smoothing distributes a small fraction of probability mass to incorrect classes. This regularization discourages the model from assigning extreme confidence values during training, which reduces overfitting because the model cannot achieve arbitrarily low loss by memorizing training examples.

Label smoothing is especially beneficial on smaller datasets or with class imbalance, where overconfidence on training samples often degrades validation performance.

Unified Focal Loss Framework

❖Combine distribution- and region-based losses into one framework

	Distribution-based loss	Compound loss	Region-based loss
Parameters	$\lambda = 0$	$\lambda > 0$ & $\lambda < 1$	$\lambda = 1$
$\gamma < 1$ & $\delta \neq 0.5$ All classes not treated equal	Asymmetric focal CE loss	Asymmetric unified focal loss	Asymmetric focal Tversky loss
$\gamma < 1$ $\delta \neq 0.5$ All classes treated equal	Symmetric focal CE loss	Symmetric unified focal loss	Symmetric focal Tversky loss
$\gamma = 1$ $\delta \neq 0.5$	CE loss	Tversky + CE loss	Tversky loss
$\gamma = 1$ $\delta = 0.5$	CE loss	CE + Dice loss	Dice loss

λ = relative weightings of distribution- and region-based losses
γ = focal weighting
δ = relative FN and FP weightings

Yeung, M., Sala, E., Schönlieb, C.B. and Rundo, L., 2022. Unified focal loss: Generalising dice and cross entropy-based losses to handle class imbalanced medical image segmentation. *Computerized Medical Imaging and Graphics*, *95*, p.102026.

27

Yueng et al. (2022) suggested a framework for integrating focal-versions of distribution- and region-based loss metrics into a single metric, which they termed the unified focal loss. Distribution-based losses are those derived from cross entropy (cross entropy, weighted cross entropy, focal cross entropy, and weighted focal cross entropy) while region-based losses are those derived from the Dice loss (Dice loss, weighted Dice loss, focal Dice loss, weighted focal Dice loss, Tversky loss, weighted Tversky loss, focal Tversky loss, and weighted focal Tversky loss).

Altering the parameterization of the unified focal loss allows for obtaining a variety of loss metrics including CE, focal CE, Dice, Tversky, focal Tversky, CE + Dice, CE + Tversky, and focal CE + focal Tversky. The symmetric version of the loss allows for treating each class equally in regards to the application of the focal corrections while the asymmetric version allows for treating classes differently when applying the focal corrections.

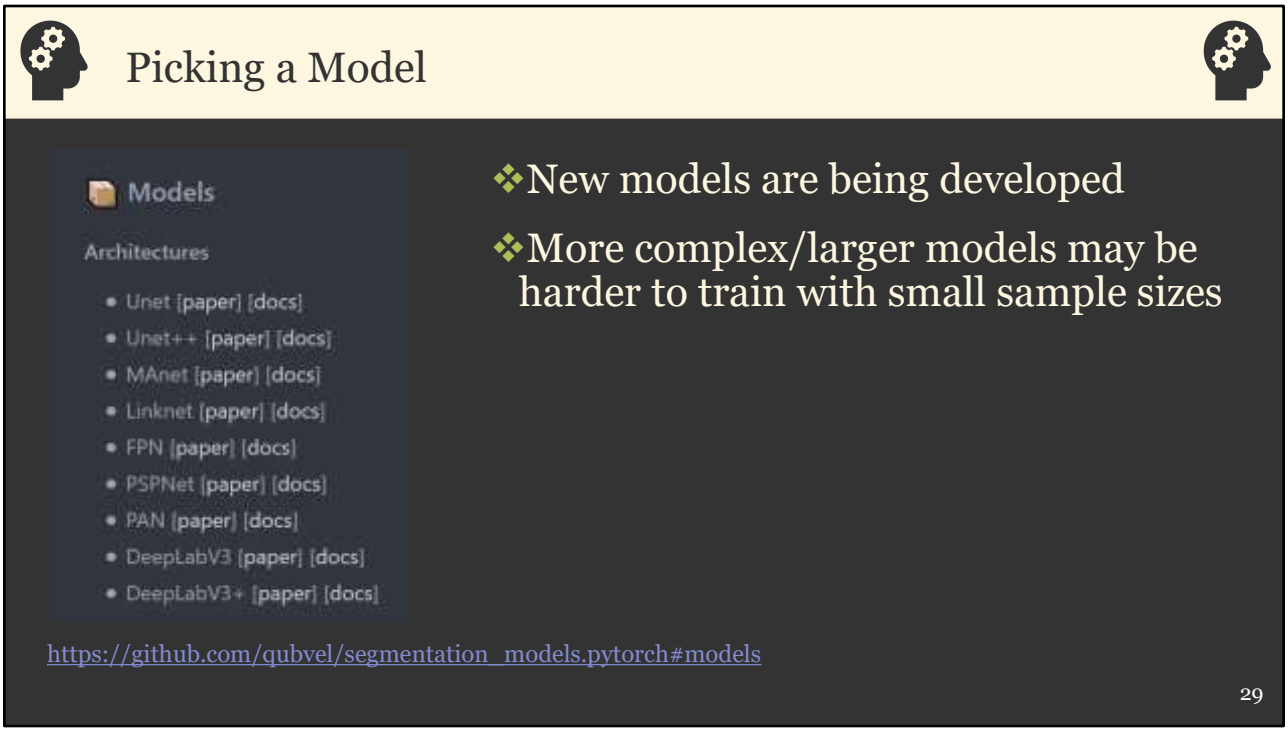
This unified loss is parameterized by three user-defined parameter: λ, γ, and δ. λ controls the relative weight of the distribution and region-based losses. When $\lambda = 0$, only the distribution-based loss is considered. When $\lambda = 1$, only the region-based loss is considered. If $\lambda = 0.5$, the losses are equally weighted. γ controls the focal adjustment, or the relative weighting applied to difficult-to-predict samples. If $\gamma = 1$, then no focal weighting is applied. Smaller values of γ

result in increased weighting applied to difficult-to-predict classes or samples. The δ parameter controls the relative weighting of false negative (FN) and false positive (FP) errors. A value larger than 0.5 puts more weight on FN as opposed to FP errors while a value less than 0.5 puts more weight on FP errors relative to FN errors. If δ = 0.5, FN and FP are equally weighted. For the region-based loss, this would equate to using Dice as opposed to Tversky.

Architectural Considerations

28

In this section, we explore issues associated with choosing model architectures and augmenting model architectures.



One of the first decisions required is which algorithm or architecture to use. This will be at least partially dictated by the problem type. For example, a UNet is appropriate for semantic segmentation but not object detection.

Also, since deep learning is a rapidly advancing field the most state-of-the art models will change with time. Unfortunately, although more complex models may be able to model more complex patterns in data and, as a result, yield more accurate predictions, they may also commonly require more training data in order to avoid overfitting.

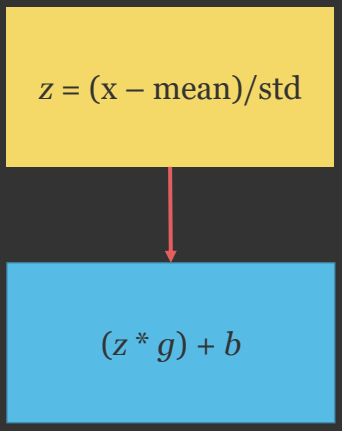
Generally, researchers and practitioners have found that simpler models may perform well if they are parameterized well, provided quality training examples, and trained using appropriate routines.

For example, if I am undertaking a semantic segmentation task, I often start with a UNet and move on to a more complex model, such as UNet++ or DeepLabv3+, only if the UNet does not perform well. I also sometimes explore different backbones (e.g., ResNet-101 vs. ResNet-34) for use with the same architecture. Note that larger backbones generally require larger training datasets to avoid overfitting. We will further discuss some of these considerations in the semantic segmentation modules.

Fortunately, using libraries such as Segmentation Tools, which is explored in our PyTorch modules, allows for a variety of algorithms to be applied with little alteration to the code.

Batch Normalization

- Does not allow activations to get really small or large in comparison to other activations
- Decreases the influence of large weights/stabilizes training process
- Reduces overfitting
- Can decrease training time
- Can use a larger learning rate
- Occurs on a batch-by-batch basis




```
nn.BatchNorm1d(hiddenSizes[0])
```

30

As mentioned in prior modules, batch normalization is commonly implemented throughout architectures. I generally recommend using batch normalization. The implementation of batch normalization layers is demonstrated in the PyTorch examples.

Batch Normalization

Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: *International Conference on Machine Learning*. PMLR, pp. 448–456.

Li, Y., Wang, N., Shi, J., Liu, J., Hou, X., 2016. Revisiting batch normalization for practical domain adaptation. arXiv preprint arXiv:1603.04779.

Bjorck, J., Gomes, C., Selman, B., Weinberger, K.Q., 2018. Understanding Batch Normalization. arXiv:1806.02375 [cs, stat].

Santurkar, S., Tsipras, D., Ilyas, A., Madry, A., 2018. How does batch normalization help optimization? *Advances in neural information processing systems* 31.

Luo, P., Wang, X., Shao, W., Peng, Z., 2018. Towards understanding regularization in batch normalization. arXiv preprint arXiv:1809.00846.

Thakkar, V., Tewary, S., Chakraborty, C., 2018. Batch Normalization in Convolutional Neural Networks—A comparative study with CIFAR-10 data, in: *2018 Fifth International Conference on Emerging Applications of Information Technology (EAIT)*. IEEE, pp. 1–5.

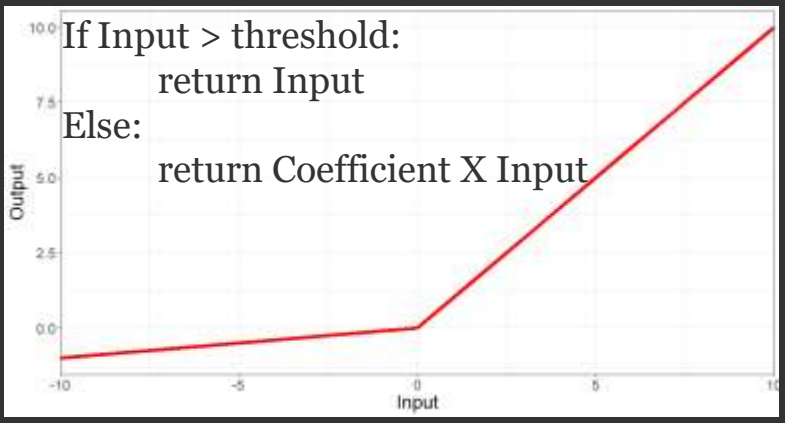
Li, X., Chen, S., Hu, X., Yang, J., 2019. Understanding the disharmony between dropout and batch normalization by variance shift, in: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. pp. 2682–2690.

Garbin, C., Zhu, X., Marques, O., 2020. Dropout vs. batch normalization: an empirical study of their impact to deep learning. *Multimed Tools Appl* 79, 12777–12815.

This slide lists papers that explore the use of batch normalization. Note that there is still some debate as to why it is effective.

Leaky Rectified Linear Unit (Leaky ReLU)

- Nonlinear
- More computationally efficient than sigmoid or tanh
- Widely used
- Does not reach gradient of zero like RelU
- Fix dying ReLU




```
nn.LeakyReLU(negative_slope=0.1, inplace=True)
```

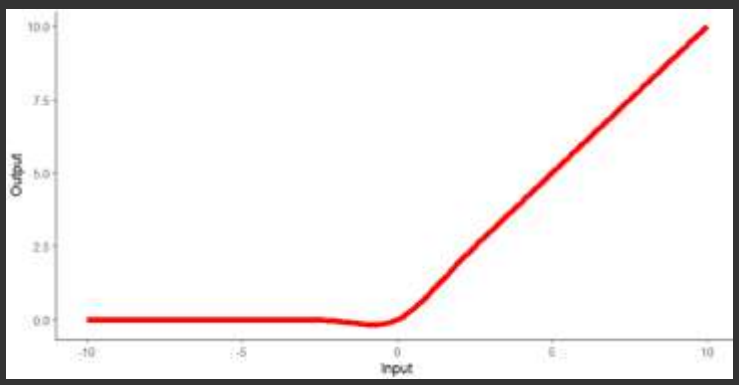
32

In order to combat the "dying ReLU" issue, leaky ReLU maintains a gradient for negative activations, as opposed to converting them to 0, by applying a slope term. Thus, the gradient does not become zero, but changes gradually.

I am generally a fan of using leaky ReLU. This is a simple change to make within architectures.

GELU

- Gaussian Error Linear Unit
- Cumulative distribution function of the standard Gaussian distribution
- Approximated as:



$$= 0.5x(1 + (1 + \tanh(\sqrt{2/\pi}(x + 0.044715\, x^3)))$$

33

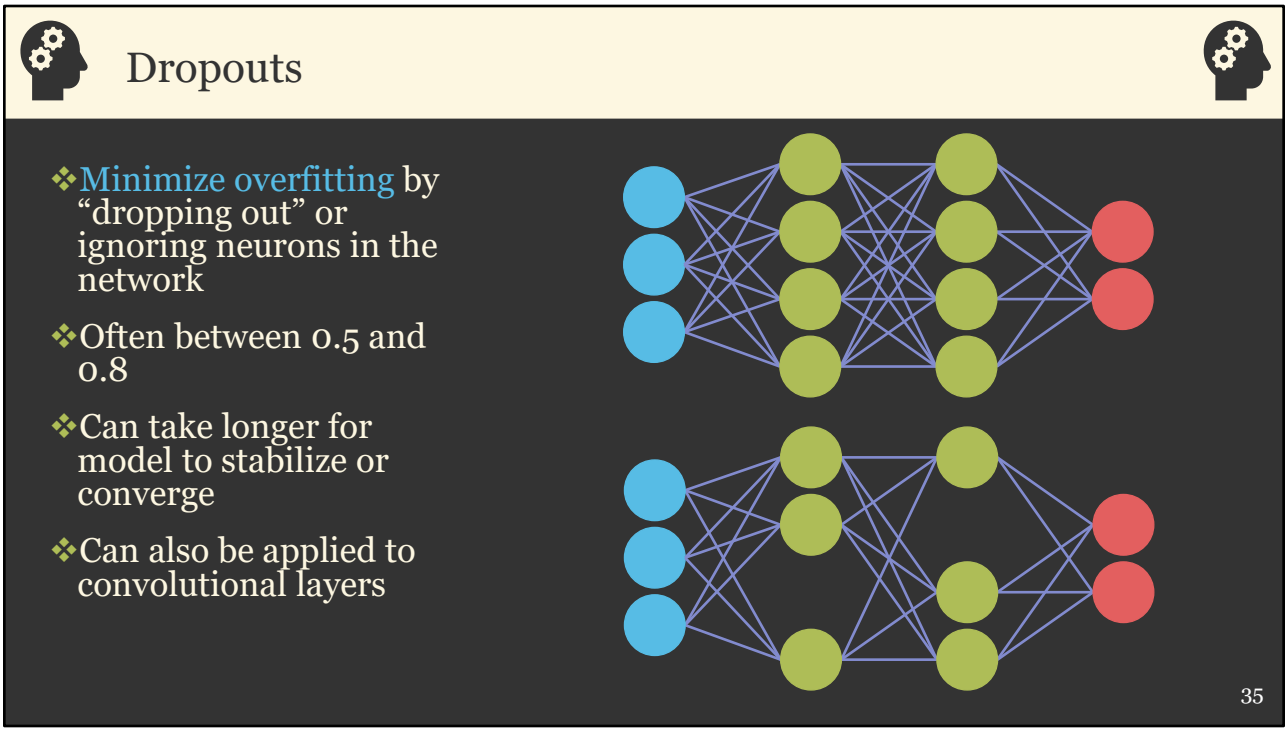
A more recently adopted activation function, which is used in some Transformer and ConvNeXT architectures, is the Gaussian Error Unit (GELU). It represents the cumulative distribution function of the standard Gaussian distribution and can be approximated using the equation provided at the bottom of the slide. This activation function can be considered as an alternative to ReLU.

Other Activation Functions

- Leaky ReLU
- Parameterized ReLU
- Swish
- Smish
- Mish
- Gaussian Error Linear Unit (GELU)

34

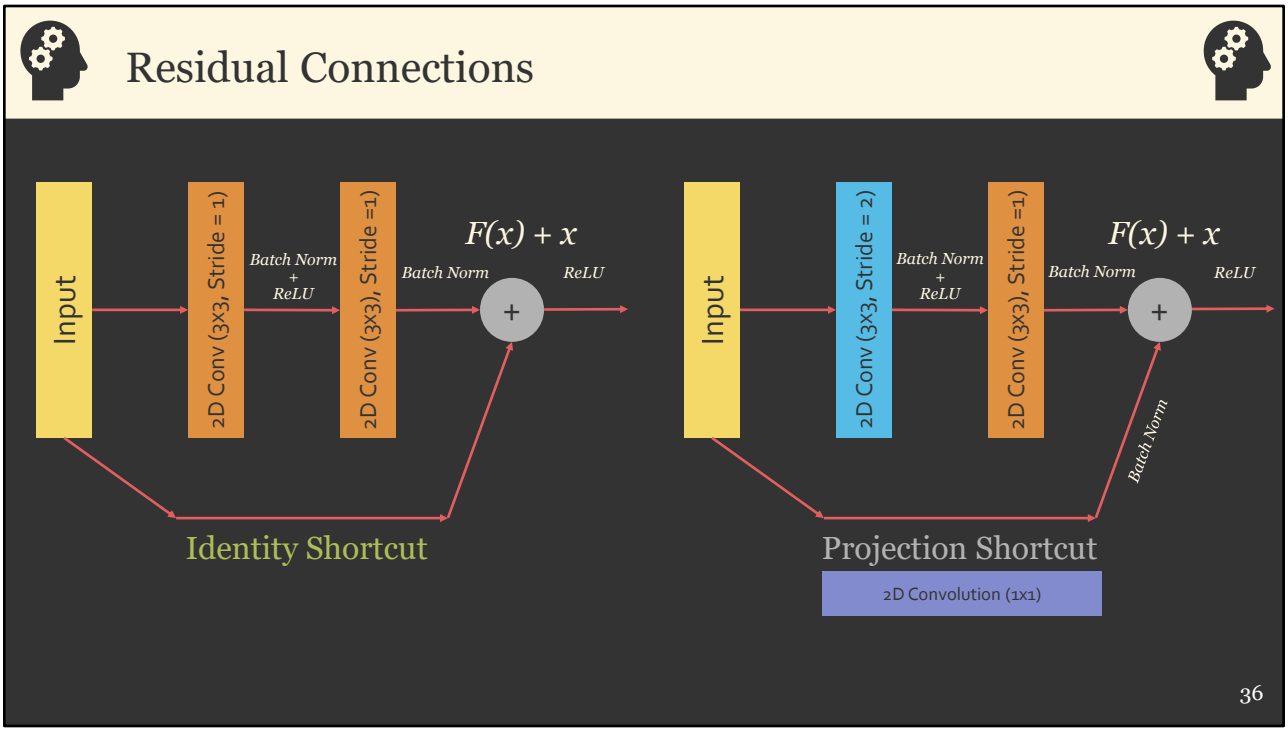
There are other activation functions that can be used in place of ReLU. A few options are listed here.



Dropouts consist of randomly dropping out or ignoring some neurons and their associated weights/parameters in the network during some weight/parameter updates.

Dropping this information can reduce the reliance and fit of the model to the training data to potentially improve generalization. This is a type of model regularization.

Dropouts can be applied to both fully connected and convolutional layers.



Residual connections, such as those implemented in ResNet architectures, have generally been shown to be of value.

It would make sense that increasing the size, complexity, and/or number of layers in a CNN would result in improved model performance due to the ability to model more complex relationships and abstract the data to a greater degree. However, it has been documented that this is not the case. Larger networks can lead to overfitting, especially when the training dataset is not large. Perhaps more problematic, when networks get very large, early layers are very separated from later layers, which can result in very small or infinitely small gradients. This is known as the vanishing gradient problem. This can result in the model being very difficult to train effectively. Thus, researchers hit a wall as continuing to increase the depth of the CNN architecture proved to be problematic.

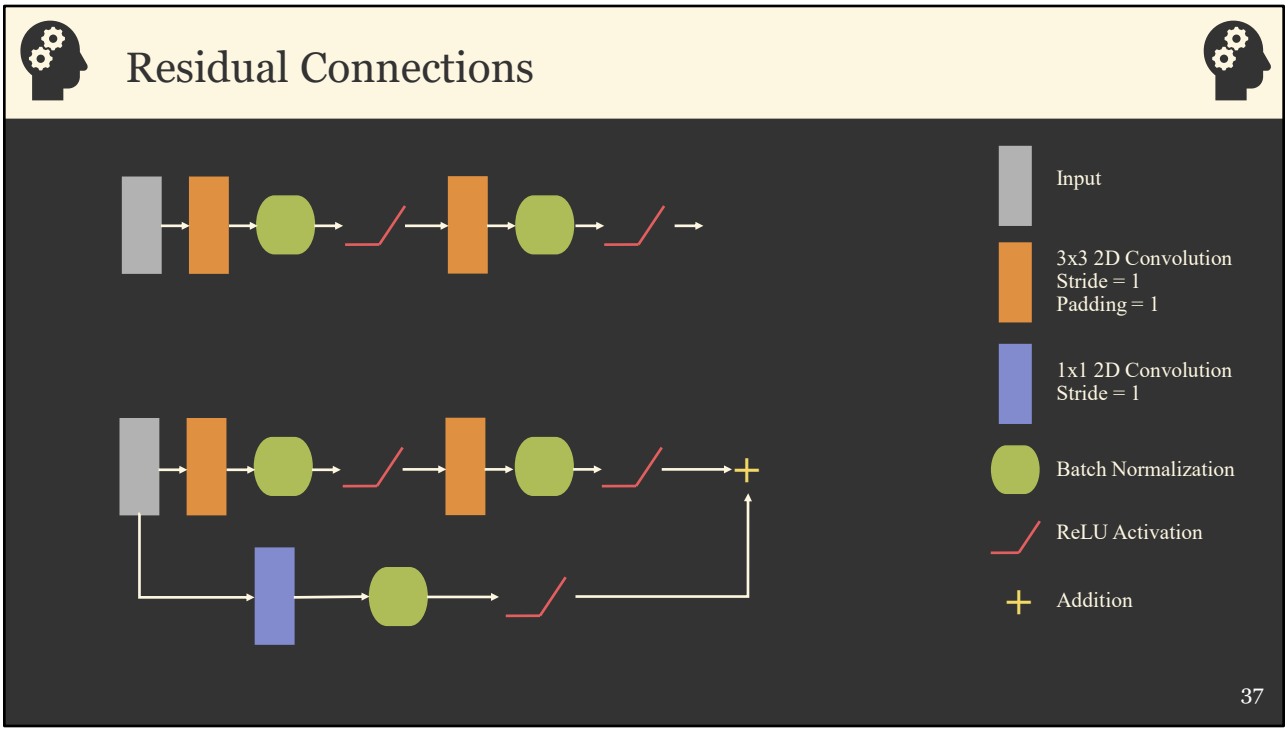
One solution to this problem was introduced by the ResNet architecture. The idea here is to use shortcut connections in which some layers skip past components of the model and are merged back in at a later point by simple addition. This allows for less separation between earlier layers and later layers and reduces the impact of the vanishing gradient problem. In other words, it provides a shorter path for the gradients to flow.

Adding shortcuts in the architectures allows for shorter paths over which the gradient can be calculated since they bypass components of the model architecture. Practically, learning the original signal plus an added signal is easier than learning just the new signal generated by the block. Learning a residual, or the difference between the input and output of a residual block, is easier than learning the original feature maps.

Other than just within the ResNet family of architectures, residual connections have generally been found to be of value, especially as architectures grow in size.

This slide conceptualizes the two types of shortcut connections used within the ResNet architecture. When the size of the spatial dimensions and/or number of channels of the output data from the block and the data that are being fed through the shortcut are the same, there is no need to alter the shape of the array, and the data can simply be added together. This is known as an identity shortcut connection.

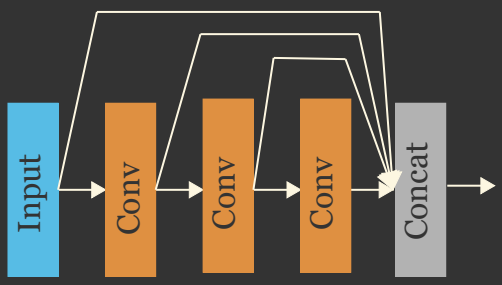
When the arrays that are to be merged have different sizes in the spatial dimensions and/or different numbers of channels, the data being fed along the shortcut path must be altered to match the output data from the block. This type of shortcut is called a projection shortcut connection. There are several ways to accomplish this resizing. One option is to use a 2D convolutional layer with a kernel size of 1x1 and a stride of 2 to decrease the array size by half. If the number of channels need altered to match, 1x1 2D convolution can also be used. A stride of 1 will not decrease the array size but can be used to alter the number of channels or feature maps.



This slide diagrams a simple residual connection. The top diagram represents a sequence of operations with no associated residual connection. In the bottom diagram, a residual connection is added. The residual connection bypasses the operations to add the input to the output of the series of operations. Along the skip connection, a 1×1 2D convolutional layer is used to augment the number of feature maps so that the number of input and output feature maps are the same and can be added together. Note also the use of batch normalization and activation functions throughout the architecture.

Dense Connections

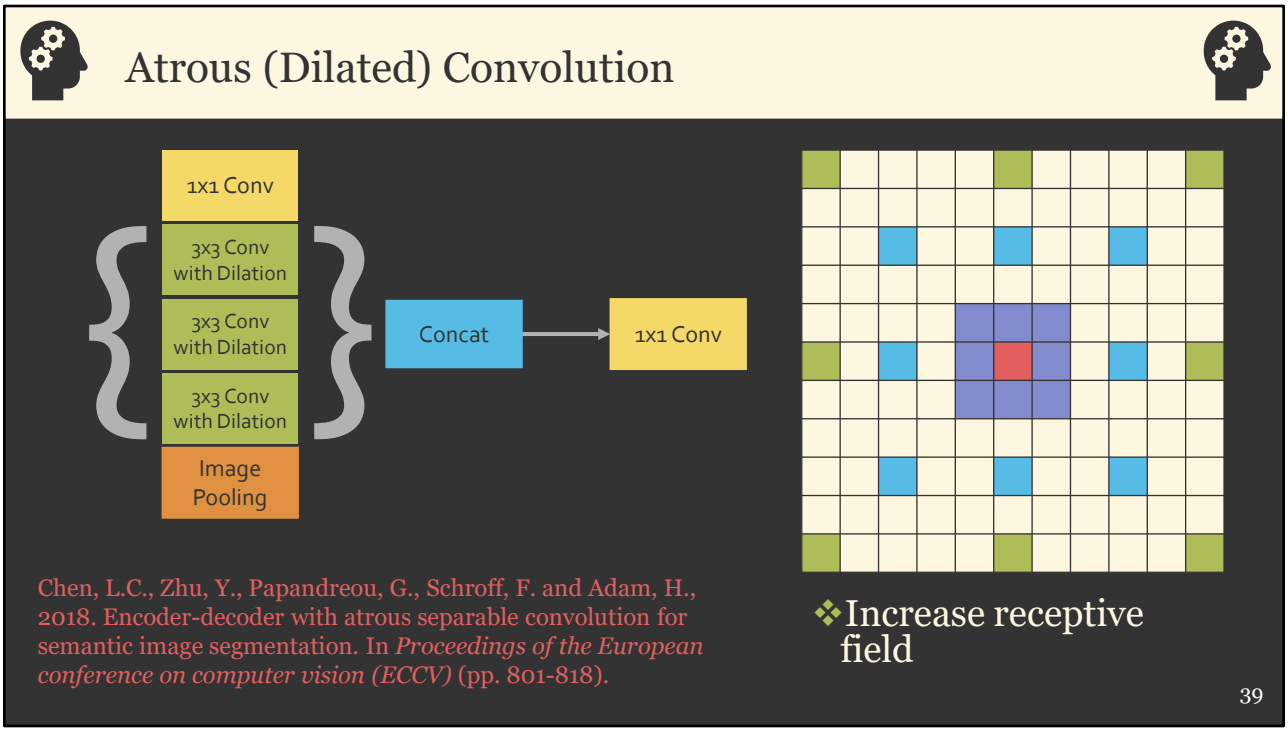
- More sharing of information between layers in the architecture
- Allows for deeper architectures



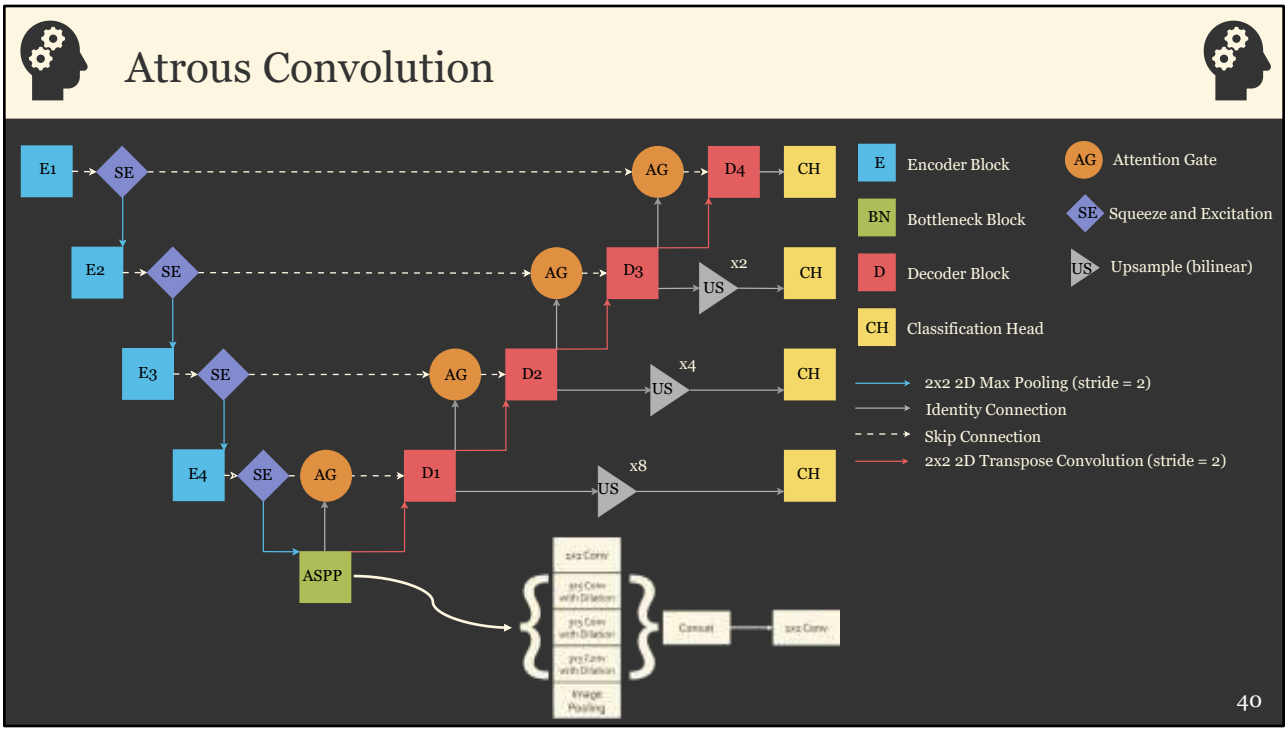
Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K.Q., 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700-4708).

38

Dense connections expand upon the concept of residual connections by adding connections between non-adjacent blocks. This can further reduce the issue of vanishing gradients and allow for develop larger, deeper architectures.



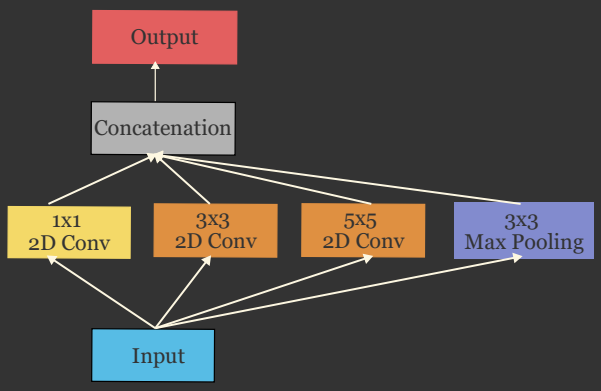
Atrous convolution, such as is implemented within the atrous spatial pyramid pooling (ASPP) module used in the DeepLabv3+ architecture, can be used to capture spatial connect information over varying and larger fields of view. One of the drawbacks of convolutional layers is that they have a limited receptive field since they rely on small kernels (generally, 3×3 or 5×5). Adding zeros into the kernels allows for increasing the receptive field without increasing the number of trainable parameters.

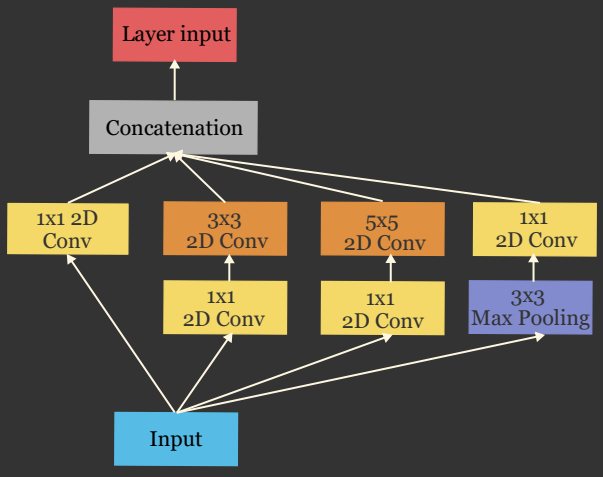


As noted in the Semantic Segmentation module, the ASPP module is implemented within the DeepLabv3+ architecture. It is also possible to incorporate this module into other architectures. For example, the bottleneck block of a UNet architecture can be replaced with an ASPP module. Again, the goal here is to capture spatial context information over a larger receptive field.

Branching

- Key innovation = branching architectures





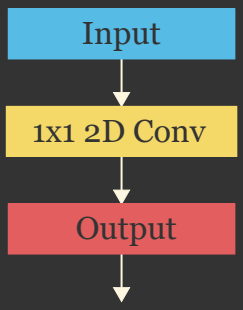
Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818-2826).

41

Branching architectures, such as those implemented within the InceptionNet- and GoogleNet-family of architectures, offer another means to capture spatial context information over varying scales.

Feature Reduction

- 1x1 2D convolution can be used to reduce the number of feature maps in the architecture
- Can be used to make architectures lighter weight and to reduce the number of trainable parameters
- Not really convolution: a smaller set of values at each cell are generated from the input values; adjacent cell values are not considered

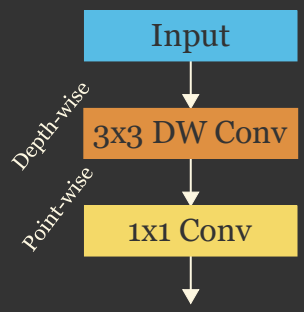


42

1×1 2D convolutional layers can be integrated to reduce the number of feature maps and the associated number of trainable parameters. This is not strictly a convolutional operations, since spatial patterns are not being captured using kernels. Instead, a set of cell values at a pixel location are reduced to a smaller set of pixel values.

Depth-Wise Separable Convolution

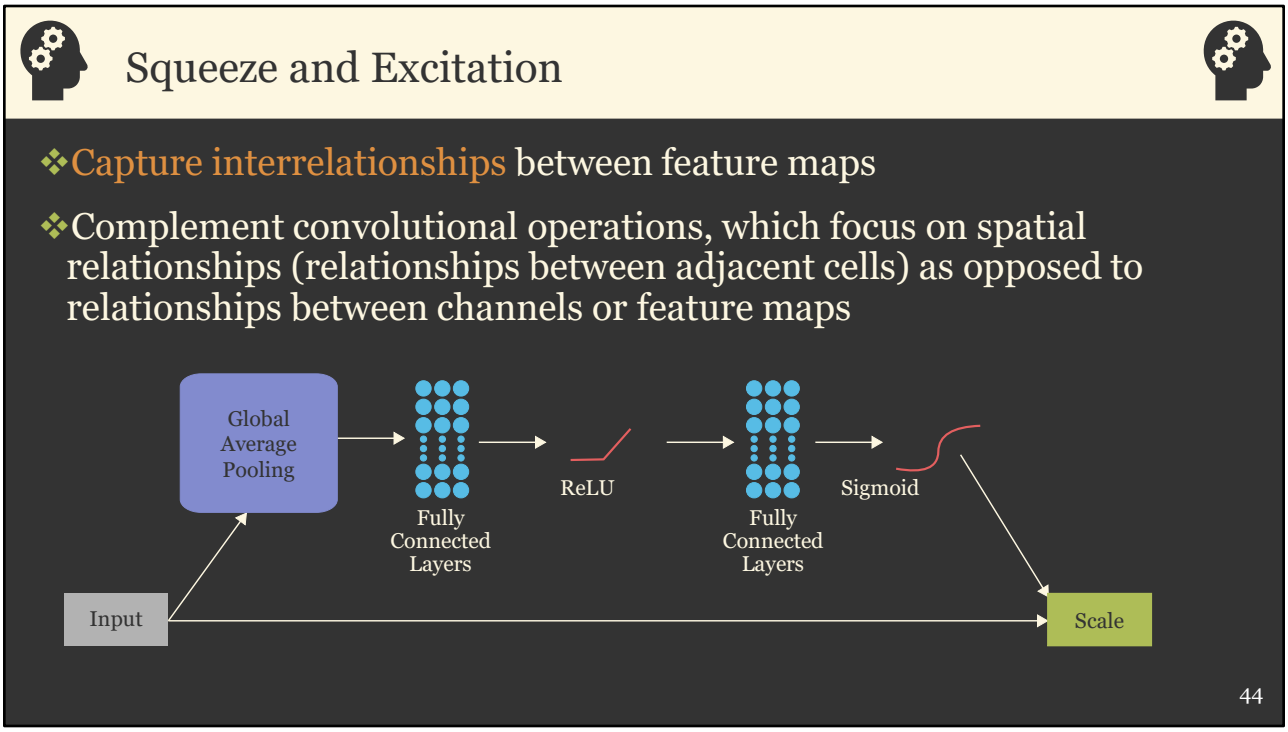
- Depth-wise = perform separate convolution operations on each input channel or feature map to generated the same number of output feature maps (focus on spatial information)
- Point-wise = 1x1 2D convolution to change the number of feature maps and share information between the learned representations
- Lighter weight than traditional convolution (less matrix multiplication required)
- Key component of efficient architectures (e.g., MobileNet-v2)



43

Other than improving model accuracy, it is also sometime of interest to improve model efficiency. This is especially important when models must process data quickly or on mobile or edge devices.

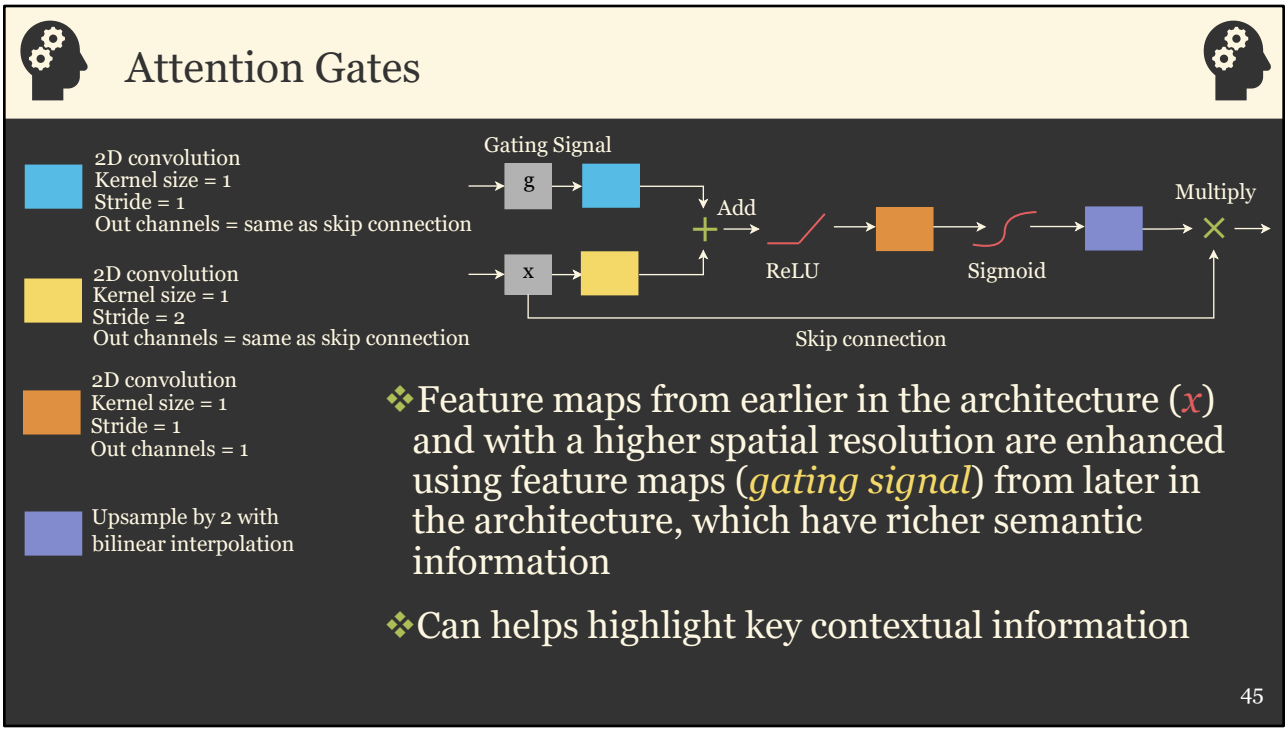
Depth-wise separable convolution offers a lighter means to perform convolution operations in comparison to traditional convolution methods. This is accomplished by breaking the operations into two components. Depth-wise convolution, which generally uses a 3×3 window, learns separate kernels for each input channel or feature map while point-wise convolution accepts the representations learned by the depth-wise convolution to change the number of feature maps and share information between the learned representations. The depth-wise component focuses on learning spatial information while point-wise convolution focuses on learning relationships between representations learned by the depth-wise operations.



The general goal of convolution operations is to learn local spatial relationships. There is less of a focus on learning interrelationships between input channels or associated feature maps. Capturing these interrelationships may improve model performance. This is the purpose of a squeeze and excitation module.

The input data are collapsed to a single value for each input channel. Specifically, the average or mean of each input is returned. These means are then passed through a fully connected layer to further augment the values and learn interrelationships. The number of output features from this fully connected layer will be less than the number of input features. A ReLU activation is then used to incorporate non-linearity. A second fully connected layer is used to further augment the data and return a set of values equal in length to the number of channels or feature maps provided as input to the squeeze and excitation block. These outputs are then rescaled to a range of 0 to 1 using a sigmoid activation. The input data are then multiplied by these 0 to 1 values to rescale them.

Again, the goal of this process is to model interrelationships between the input data. Note that there are different implementations of squeeze and excitation modules. This is just one example.

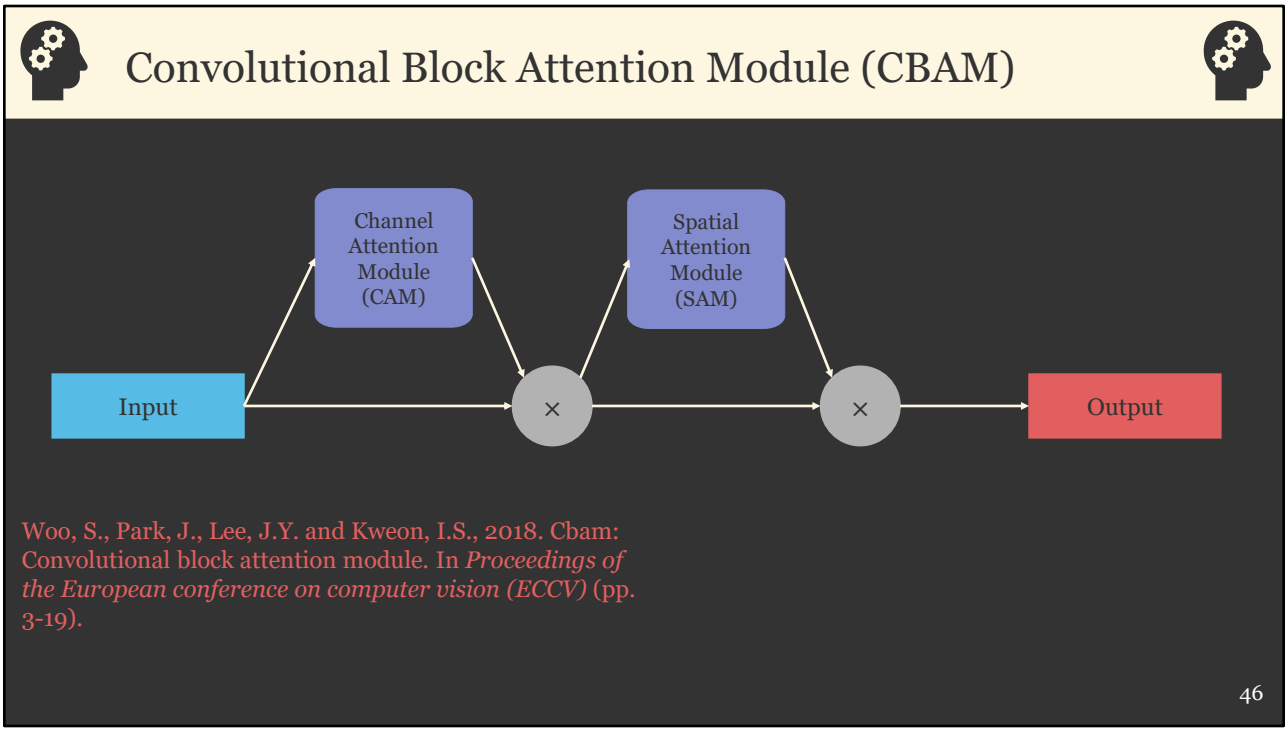


The goal of an attention gate is to augment prior feature maps using the richer semantic information learner later in the architecture. For example, in a UNet architecture, the feature maps from Encoder Block 2 could be augmented using information from the output feature maps from Decoder Block 3. The idea here is that the richer semantic information from deeper in the architecture can be used to modify or add focus to improve earlier feature maps. This can allow for improving intermediate feature maps or data representations.

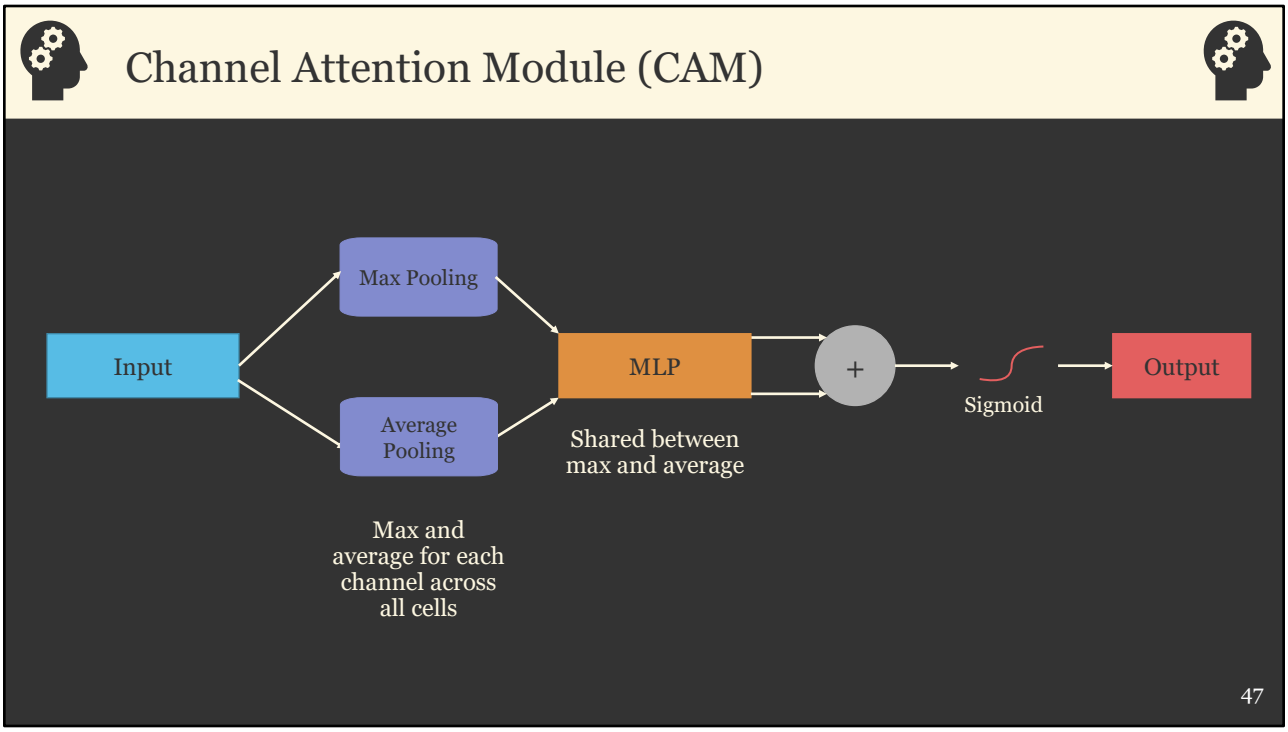
Attention gates can be implemented using different methods. The example on this slide is simply one implementation.

The x variable represents the input feature maps that will be augmented. The gating signal will be features from deeper in the architecture that will guide the augmentation of the input feature maps. The x feature maps are first augmented using a 3×3 2D convolutional layer with a stride of 2. This reduces the size of the feature maps in the spatial dimensions such that they match that of the gating signal feature maps. The feature maps from the gating signal are augmented with a 3×3 2D convolutional layer with a stride of 1 that maintains the array size in the spatial dimensions but changes the number of feature maps to match that of the x feature maps. These two sets of feature maps are then added together and a ReLU activation is applied to add non-linearity. The

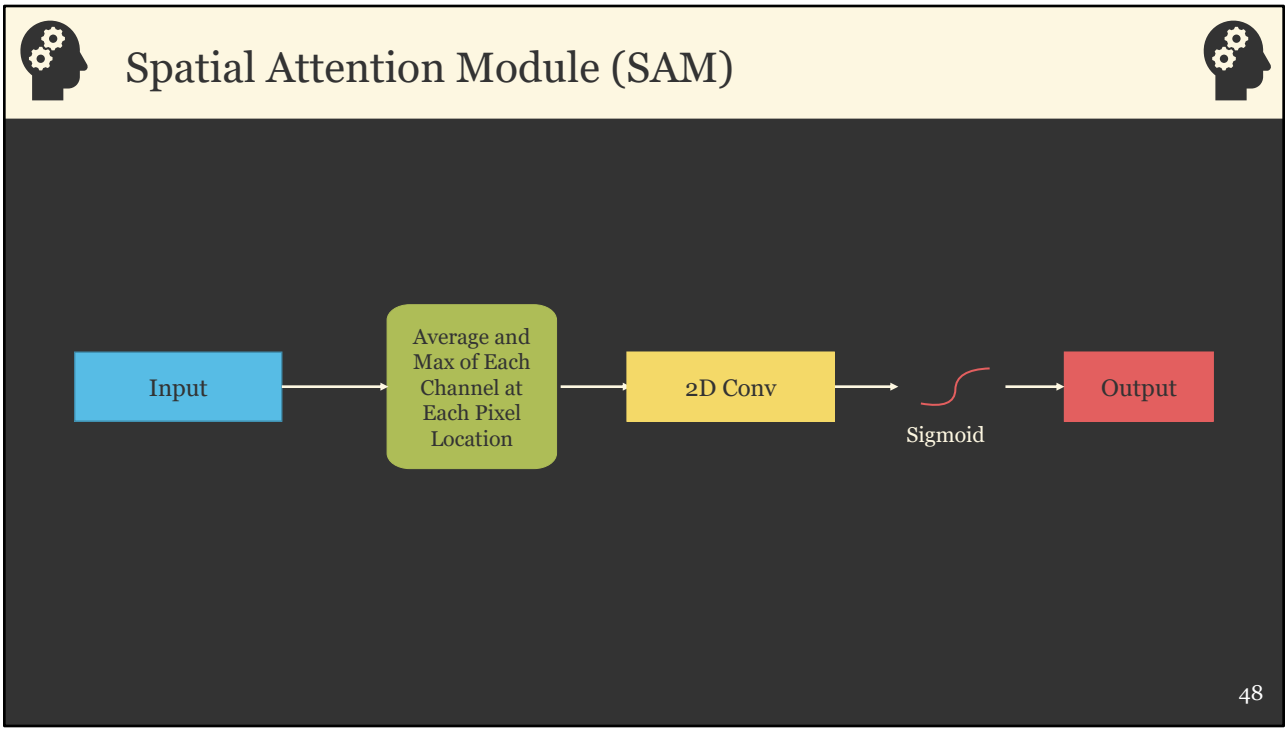
results are then pass through another 3×3 2D convolutional layer with a stride of 1. This layer will output a single feature map that is then rescaled to a range of 0 to 1 using a sigmoid function. In order to return an array with the original spatial resolution of the *x* feature maps, bilinear interpolation upsampling is then applied. The results are then multiplied by the input *x* feature maps. Larger values in the 0 to 1 grid are used to add focus to important information in the input feature maps.



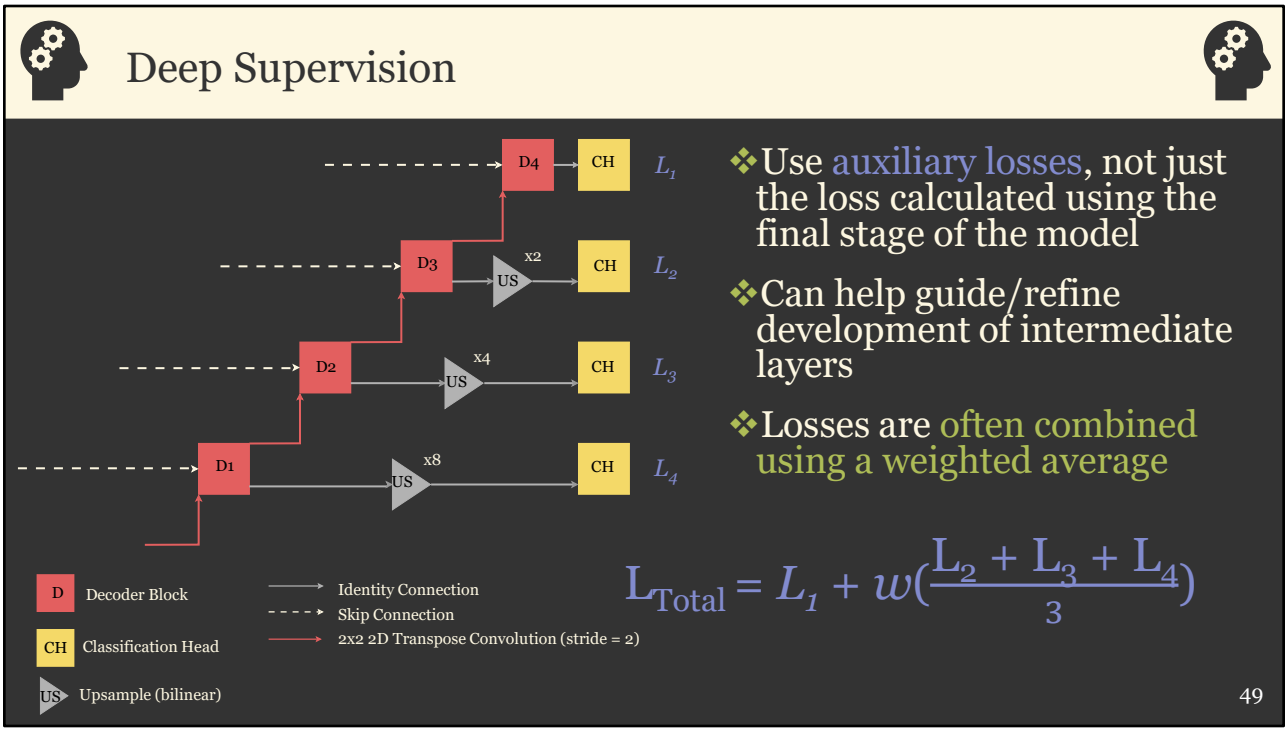
The Convolutional Block Attention Module (CBAM) is a lightweight, general-purpose attention mechanism designed for convolutional neural networks that sequentially refines feature maps along two distinct dimensions: channel and spatial. Rather than treating all feature channels and spatial locations as equally important, CBAM learns to emphasize the most informative features and suppress less useful ones, effectively asking what to attend to and *where* to attend. It is inserted between convolutional layers and operates in a forward-pass-compatible way, meaning it can be plugged into most standard CNN architectures (such as ResNets or VGG) with minimal overhead. The module produces an attention-weighted feature map by first applying channel attention and then spatial attention in sequence, with each stage generating a multiplicative mask that rescales the input feature map before passing it to the next stage.



The Channel Attention Module (CAM) focuses on what is meaningful by exploiting the inter-channel relationships of a feature map. Given an input feature map, it compresses the spatial dimensions using both global average pooling and global max pooling in parallel, producing two separate channel descriptors that capture complementary statistics: average-pooled features encode soft distributional information, while max-pooled features capture the most salient activations. Both descriptors are then passed through a shared multi-layer perceptron (MLP) with a bottleneck hidden layer (controlled by a reduction ratio to keep the parameter count low), and their outputs are summed element-wise before being passed through a sigmoid activation. The resulting 1D channel attention vector is then multiplied back across the spatial dimensions of the original feature map, effectively rescaling each channel according to its relative importance.



The Spatial Attention Module (SAM) complements the CAM by focusing on where informative features are located within the feature map. Taking the channel-refined output from the CAM as input, it aggregates information across the channel dimension by applying both average pooling and max pooling along the channel axis, yielding two 2D feature maps that capture spatial statistics. These two maps are concatenated along the channel dimension and passed through a single convolutional layer with a large kernel (typically 7×7) to capture broad spatial context, followed by a sigmoid activation. The resulting 2D spatial attention map highlights the most task-relevant spatial regions and is multiplied element-wise with the incoming feature map, allowing the network to focus its representational capacity on the most informative locations in the image.



There are different means to implement deep supervision; however, the underlying idea is to calculate additional, auxiliary losses from other stages in the architecture to supplement the final loss and further guide the learning process, especially for improving intermediate layers.

The slide provides an example implementation of deep supervision within the decoder component of a UNet architecture. In order to implement deep supervision, separate predictions are made using the outputs from Decoder Blocks 1 through 4. Since Decoder Blocks 1 through 3 will have a smaller spatial resolution than the reference or target masks, they must be upsampled prior to being passed to a classification head. Classification heads are implemented with 1×1 2D convolution with a stride of 1.

Losses can then be calculated by comparing each prediction to the target or reference masks. The final loss is calculated as a weighted average of all the losses. The loss for the final layer, Decoder Block 4 in the example, is generally weighted higher than the other losses.

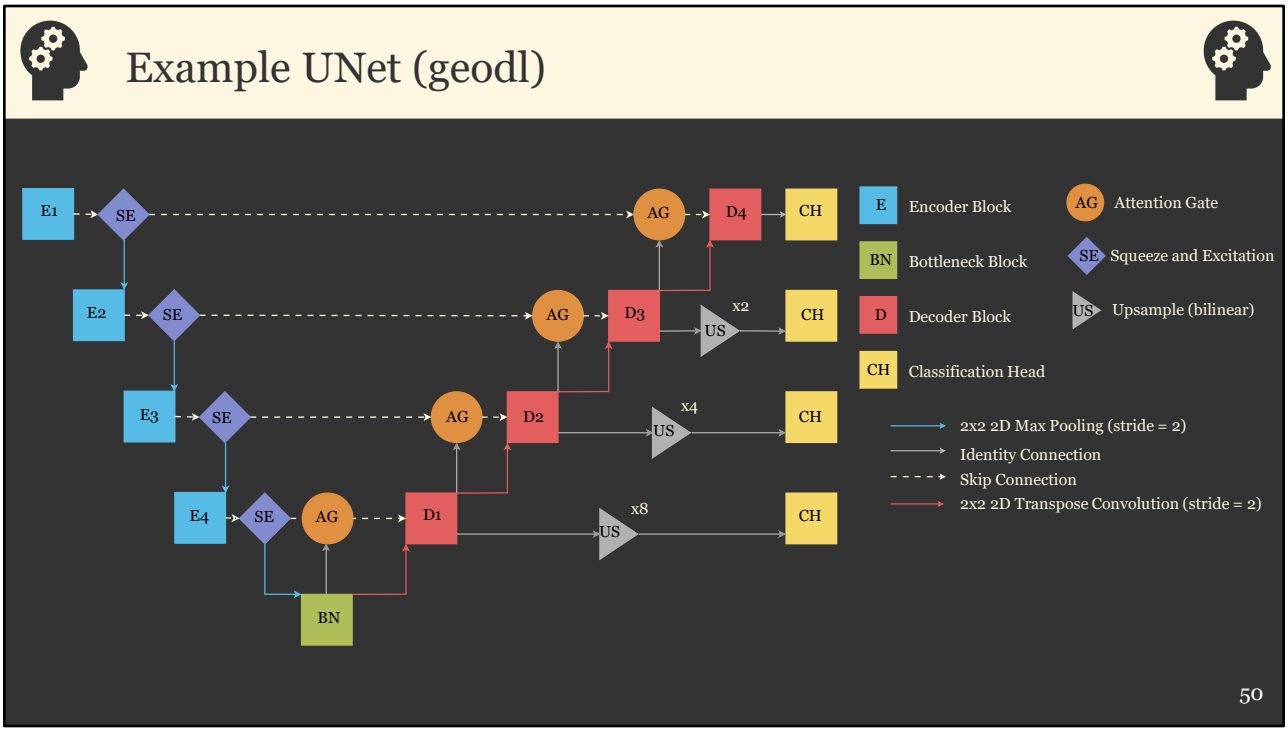
Again, the goal here is to improve the feature representations generated by intermediate layers using a loss that is not as far removed from them as the loss calculated using the results from the final stage of the architecture.

There are different ways to implement deep supervision. For example, deep supervision is implemented within UNet++ using its nested architecture.

In the early stages, the deeper (lower-resolution) supervision signals from the intermediate decoder stages are arguably more important. The network has not yet learned meaningful high-level representations, and the auxiliary losses at lower resolutions guide the encoder and early decoder layers to learn semantically relevant features faster than they would with only the final loss. Keeping the auxiliary weights relatively substantial early on accelerates this coarse-level learning and helps combat vanishing gradients through the deeper parts of the network.

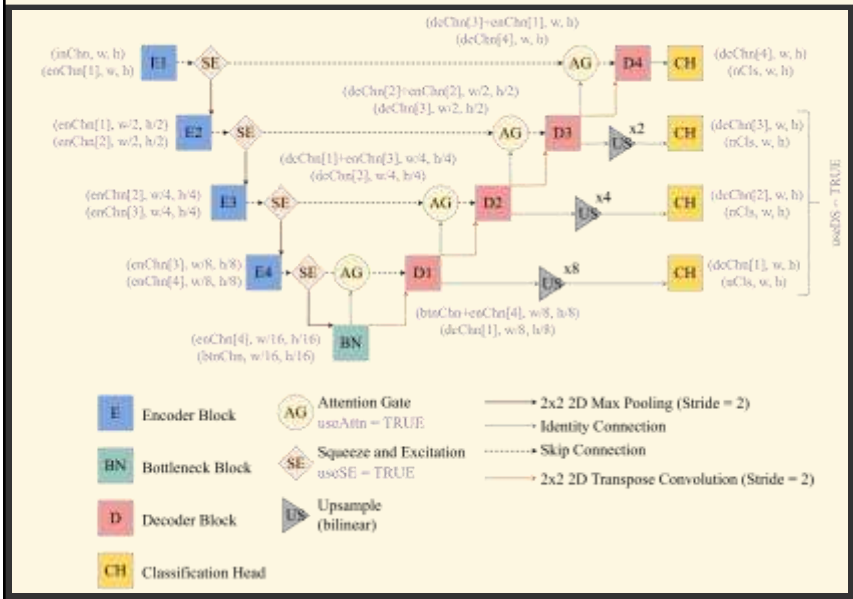
As training progresses and the network begins converging, the balance should gradually shift toward the final full-resolution output loss. The intermediate feature maps at lower resolutions are inherently limited in their ability to capture fine-grained spatial detail: they simply lack the resolution to supervise precise boundary delineation or small structure segmentation. Continuing to weight them too heavily at this stage can actually act as a regularization bottleneck, preventing the final decoder layers from specializing toward high-resolution precision.

By the final stages, the primary loss at the full output resolution should dominate, with auxiliary losses reduced to a small fractional weight (or even annealed toward zero). The network's capacity should be fully directed at optimizing the final prediction, and strong auxiliary signals at this point can introduce conflicting gradient directions that destabilize fine convergence.



This slide shows a conceptualization of a UNet-like architecture that incorporates squeeze and excitation modules into the encoder, attention gates along the skip connections, and deep supervision. One of the benefits of the modules that were discussed in this section is that they can be implemented into a variety of existing architectures. For example, a variety of UNet-like architectures have been proposed to potentially enhance base UNet.

Example with UNet (geodl)

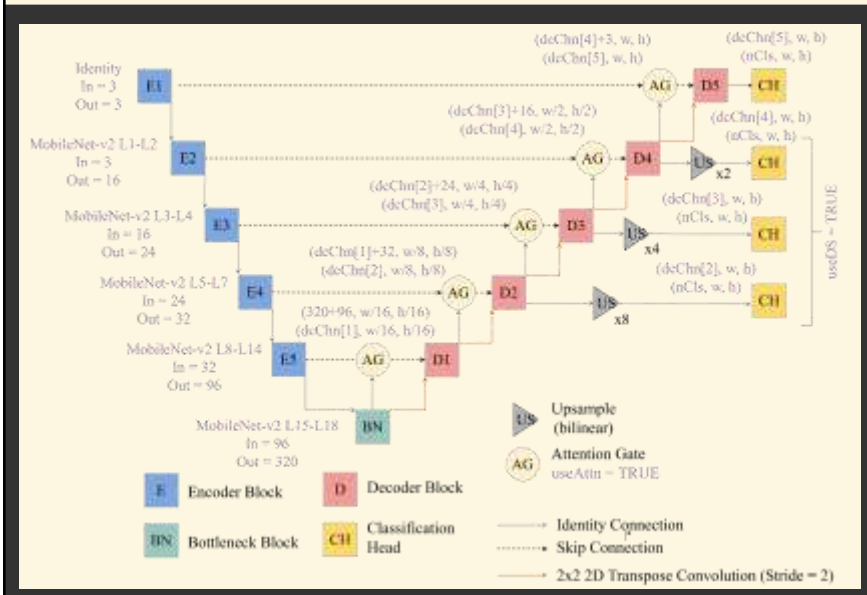


- Specify number of output feature maps in each encoder, bottleneck, and decoder block
- Incorporating residual connections throughout the architecture
- Use swish or leaky ReLU activation function as opposed to ReLU
- Squeeze and excitation in the encoder
- Attention gates between the encoder and decoder/along skip connections
- Replacing the bottleneck with a ASPP module
- Deep supervision

51

The UNet implementation within the geodl R package developed by the author of this course allows the user to implement a variety of augmentations to a base UNet architecture including changing the number of output feature maps from the encoder, bottleneck, and decoder blocks; incorporating residual connections throughout the architecture; replacing ReLU with a different activation function (leaky ReLU or swish); incorporating squeeze and excitation modules into the encoder; incorporating attention gates along the skip connections; replacing the bottleneck block with a ASPP module; and implementing deep supervision.

Pre-Trained Encoders



- Use MobileNet-v2 architecture in encoder
- Can use pre-trained encoder either frozen or unfrozen
- Specify number of output feature maps in the decoder blocks
- Use swish or leaky ReLU activation function as opposed to ReLU
- Attention gates between the encoder and decoder along skip connections
- Deep supervision

52

A second UNet implementation provided by the geodl package uses an encoder backbone based on MobileNet-v2. It also allows for using pre-trained parameters in the encoder, which can be trained during the learning process or left frozen; specifying the number output feature maps from each decoder block; replacing ReLU activation functions with leaky ReLU or swish; incorporating attention gates along the skip connections; and using deep supervision.

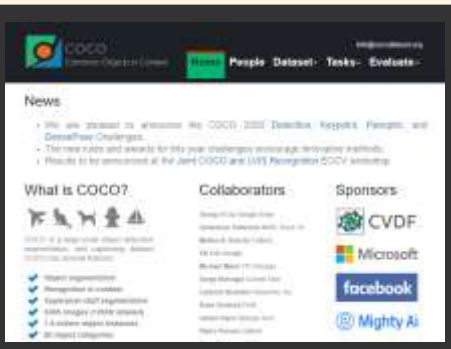
Training Process Considerations

53

Other than augmenting the model architecture, it is also possible to augment the learning process or training loop. This is the focus of this final section of the module.

Transfer Learning

http://www.image-net.org/



https://cocodataset.org/#home

```
import segmentation_models_pytorch as smp

model = smp.Unet(
    encoder_name="resnet34",        # choose encoder, e.g. mobilenet_v2 or efficientnet-b7
    encoder_weights="imagenet",     # use `imagenet` pre-trained weights for encoder initialization
    in_channels=1,                  # model input channels (1 for gray-scale images, 3 for RGB, etc.)
    classes=3,                      # model output channels (number of classes in your dataset)
)
```


Transfer learning can be applied to make use of weights/parameters learned from larger datasets, such as ImageNet and COCO.

Generally, transfer learning has been shown to improve model performance, even if the input data and problem are different than the original use case. For example, ImageNet has been used to initiate models for labeling aerial images, even though the set includes no aerial images and different classes are differentiated. The idea here is that all images have some characteristic features, such as edges, tones, and textures, so initiating from pre-trained weights/parameters is better than using a random initiation.

Transfer Learning

Less is More when Applying Transfer Learning to Multi-Spectral Data

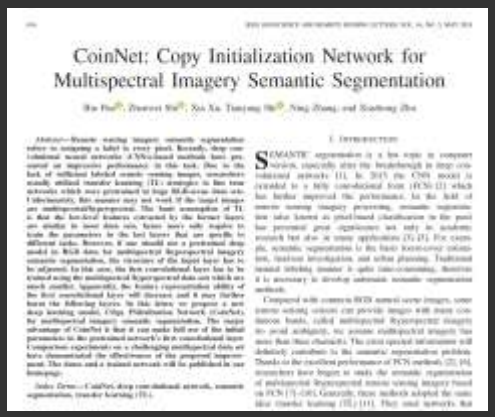
Yuvraj Sharma[1] and Robert Ross[2]

[1] Adapt Centre, Technological University Dublin, Ireland
[2] Adapt Centre, Technological University Dublin, Ireland

Abstract. Transfer Learning is widely recognized as providing notable benefits in many image processing tasks. But not all problems in the computer vision field are driven by traditional Red, Green, and Blue (RGB) imagery as used to be assumed by most large pre-trained models for Transfer Learning. Satellite based remote sensing applications for example typically use multispectral bands of light. While transferring RGB features to this non-RGB domain has been shown to generally give higher accuracy than training from scratch, the question remains whether a more suitable fine tuned method can be found. Given this challenge, this paper presents a study in multispectral image analysis using multiple methods to achieve feature transfer. Specifically, we train and compare two pre-trained models based on the ResNet50 architecture and apply them to a multi-spectral image processing task. The key difference between the two models is that one was pre-trained in a conventional way against RGB data, while the other was trained against a single band greyscale variant of the same data. Our results demonstrate an improved performance on the greyscale pre-trained model relative to the more traditional RGB model.

Keywords: Deep learning · Transfer learning · Image Analysis · Resnet · CNN · Multispectral images · ImageNet · Satellite imagery · EuroSat.

Sharma, Y. and Ross, R., 2020. Less is More when Applying Transfer Learning to Multi-Spectral Data. In *AICS* (pp. 301-312).



CoinNet: Copy Initialization Network for Multispectral Imagery Semantic Segmentation

B. Pan, Z. Shi, X. Xu, T. Shi, N. Zhang and X. Zhu, "CoinNet: Copy Initialization Network for Multispectral Imagery Semantic Segmentation," in *IEEE Geoscience and Remote Sensing Letters*, vol. 16, no. 5, pp. 816-820, May 2019, doi: 10.1109/LGRS.2018.2880756.

55

One complexity in implementing transfer learning is that most large datasets on which pre-trained weights are based, such as ImageNet and COCO, are RGB or three-band data. However, in remote sensing and geospatial science, we are often interested in extracting information from other data types, such as digital terrain data or scanned thematic maps. Also, our data may have a larger number of bands, as in the case of multispectral imagery. Since these additional bands likely contain information that may improve the differentiation of the classes, we generally do not want to throw this information out. Other disciplines have similar problems, such as the analysis of medical imagery.

One option is to initiate the model using the pre-trained weights and provide three input bands, even if they are not RGB data. In many cases, this has been shown to provide more accurate results than initiating the model using random weights/parameters, especially if all weights are unfrozen during training.

If more than three bands are used, methods have been developed to augment the available weights/parameters. Two example papers that explore this topic are referenced on this slide. Note that this is still an active area of research.

Another option is to augment the model architecture to accept more bands.

You may also need to augment the model architecture to match the number of differentiated classes.

Transfer Learning and Variable Learning Rates

- If using pre-trained weights, may want to use different learning rates in different stages of the encoder and the decoder.
- Generally lower learning rates deeper into the encoder
- Can augment learning rates during the training process

56

It has also generally been suggested that using different learning rates for different stages or components of the architecture can be useful when transfer learning is applied but the pre-trained components will still be trainable or unfrozen.

Generally, it has been documented to be useful and allow for a more stable learning process to use lower learning rates earlier in the architecture and larger learning rates later in the architecture. For example, in a UNet architecture with a pre-trained backbone, the learning rate may be lowest for Encoder Blocks 1 and 2, moderate for Encoder Blocks 3 and 4, and largest for the bottleneck and decoder components.

It is also possible to augment learning rates during the learning process differently for different components of the architecture.

In a PyTorch module, I will demonstrate how to apply different learning rates to different stages of the architecture.

Schedulers

https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.StepLR.html

- ❖Augment hyperparameters
- ❖Change learning rate
- ❖Augment training set
- ❖Shuffle data
- ❖Freeze/Unfreeze weights
- ❖Early stopping

57

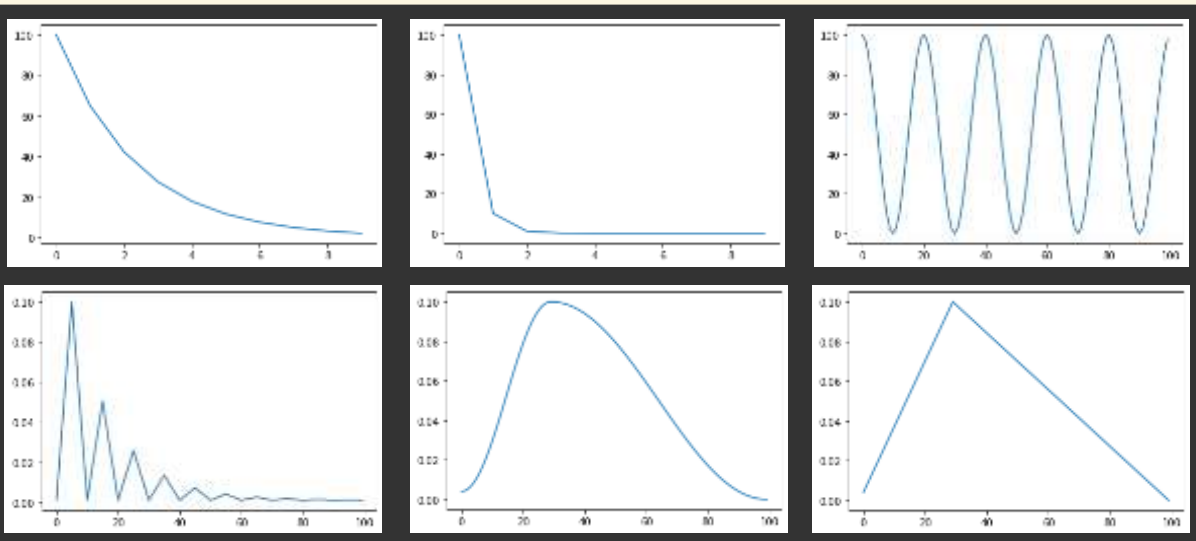
Schedulers allow for altering the learning process between mini-batches or epochs, or iterations over the training data.

For example, schedulers can be used to augment hyperparameters, change the learning rate, augment the training dataset, shuffle the training data, freeze or unfreeze weights, or apply early stopping. Early stopping is used to stop the training process before the final epoch is reached if the model is no longer improving.

As an example, let's say you have a binary classification problem in which the background class is far more abundant than the class of interest. In other words, your dataset is imbalanced. However, there is a lot of variability in the background class, so you would like to provide a wide variety of background examples. One option would be to schedule the dataset and data loader to update after each epoch to select different subsets of the background set. This would allow for the model to see a different random subset of the background training samples in each epoch.

Note that PyTorch has methods available for applying schedulers. These are generally defined then applied in the training loop. The example on the slide allows for decreasing the learning rate.

Learning Rate Schedulers



https://www.kaggle.com/code/isbhargav/guide-to-pytorch-learning-rate-scheduling 58

Augmenting the learning rate during training has generally been documented to improve model performance. It can also allow for faster training. I am specifically a fan of a one-cycle learning rates policy, where the learning rate oscillates from a low learning rate to a high learning rate then back to a low learning rate during the entire training process.

The link on this slide provides example implementations of different learning rate schedulers.

Lower Learning Rate on Plateau

- If the loss does not improve for a given number of epochs, lower the learning rate
- Why?
  - The current learning rate is too large relative to the local optimization landscape
  - Weight updates are too coarse = overshooting good solutions
  - The optimizer can't fine-tune

59

If the model is not improving for several epochs, it can be useful to reduce the learning rate. The key idea here is that the current learning rate may be too large for the current optimization landscape, which results in the updates overshooting optimal solutions or model parameters. Lowering the learning rate can reduce this issue and allow for more fine tuning.

Early Stopping

- Stop learning process early
- Training loss still decreasing but validation loss increasing
- Reduce overfitting

60

It can sometimes be helpful to stop the training process early. If the training loss continues to decrease while the validation loss starts increasing, this indicates that the model may be overfitting to the training data.

It is also possible to save models after each training epoch or save models for epochs at which the validation loss or a validation assessment metric decreases. You can then select one of the save points as the final model as opposed to using the model state after the final epoch.

Super-Convergence

- Faster convergence
- One learning rate cycle
- Large maximum learning rate
- Large learning rate regularizes training

Smith, L.N. and Topin, N., 2019, May. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial intelligence and machine learning for multi-domain operations applications* (Vol. 11006, pp. 369-386). SPIE.

Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates

Leslie N. Smith
U.S. Naval Research Laboratory, Code 5514
4555 Overlook Ave., SW., Washington, D.C. 20375
leslie.smith@nrl.navy.mil

Nicholay Topin
University of Maryland, Baltimore County
Baltimore, MD 21250
ntopin2@umbc.edu

Abstract

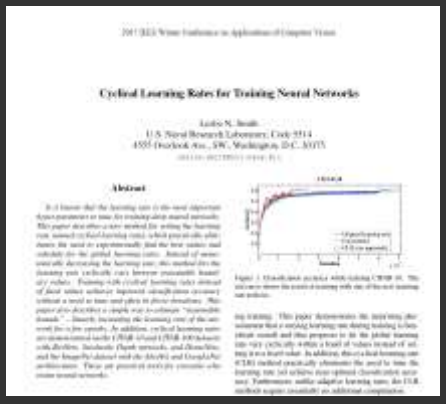
In this paper, we describe a phenomenon, which we named "super-convergence", where neural networks can be trained an order of magnitude faster than with standard training methods. The existence of super-convergence is relevant to understanding why deep networks generalize well. One of the key elements of super-convergence is training with one learning rate cycle and a large maximum learning rate. A primary insight that allows super-convergence training is that large learning rates regularize the training, hence requiring a reduction of all other forms of regularization in order to preserve an optimal regularization balance. We also derive a simplification of the Hessian Free optimization method to compute an estimate of the optimal learning rate. Experiments demonstrate super-convergence for Cifar-10/100, MNIST and Imagenet datasets, and resnet, wide-resnet, densenet, and inception architectures. In addition, we show that super-convergence provides a greater boost in performance relative to standard training when the amount of labeled training data is limited. The architectures to replicate this work will be made available upon publication.

61

The paper referenced here introduced the concept of super-convergence. The idea hear is to oscillate the learning rate and use a fairly large learning rate as the upper bound. This large learning rate can have the effect of regularizing the model and improving model generalization. This process can also allow for achieving better performance with a lower number of total training epochs.

I highly recommend reading through this paper. Note that super-convergence may not be possible for all models or problems.

Learning Rate Finder



Cyclical Learning Rates for Training Neural Networks

Leslie N. Smith

Abstract

Smith, L.N., 2017, March. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)* (pp. 464-472). IEEE.

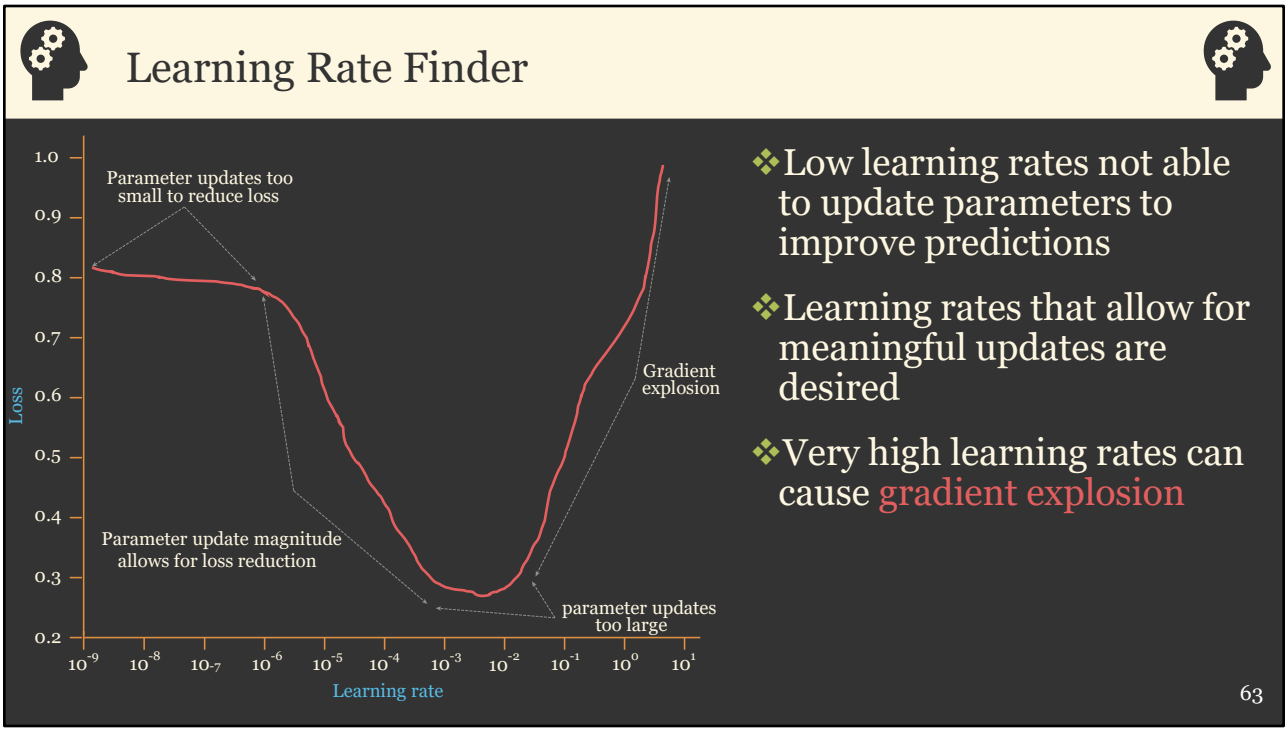
https://github.com/davidtvs/pytorch-lr-finder

62

It is generally suggested that the learning rate is the most important hyperparameter in terms of model performance and the number of training epochs required to obtain suitable results. Thus, selecting a learning rate is key.

Smith (2017) introduced a method for selecting an optimal learning rate or range of learning rates. This methods has been implemented in PyTorch and the arcgis.learn module. It is generally very fast to implement. This slide provides a link to the GitHub repo for a PyTorch implementation of this method.

If you are interested in the specifics of this method, please read through the paper reference on the slide. This method will be generally described on the next slide.



The learning rate finder works by augmenting the learning rate as each mini-batch is processed and monitoring the resulting loss. The process starts with a very small learning rate. Since a small learning rate only allows for small parameter updates, the loss generally does not improve. As the learning rate increases, the loss will begin to improve. Eventually, the learning rate will become too large, resulting in gradient explosion and an unstable learning process.

Generally, the best learning rate or range of learning rates are associated with the portion of the graph with the fastest decreasing slope. Again, very small learning rates are not effective since they do not allow for meaningful parameter update magnitudes while very large learning rates result in instability and gradient explosion.

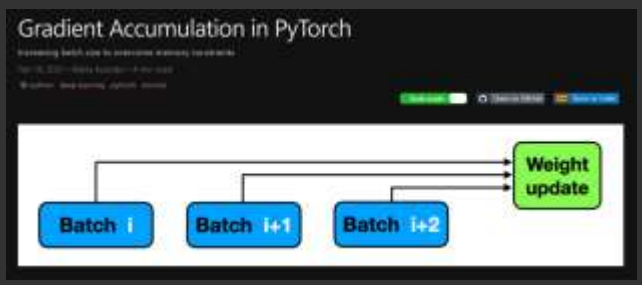
The best learning rates are associated with the descending limb as opposed to the trough since the trough represents learning rates that border the larger learning rates that result in gradient explosion. As a result, these learning rates may also result in unstable learning.

In the example on the slide, learning rates around 1e-5 or 1e-4 would be optimal. This is just an example, and results will vary for different problems. The graphs can also be noisy. Sometimes smoothing is applied to generalize

the trend.

Gradient Accumulation

- Useful with small mini-batch sizes
- Do not update gradients after every mini-batch
- Caution when using large mini-batch sizes



https://kozodoi.me/python/deep%20learning/pytorch/tutorial/2021/02/19/gradient-accumulation.html

```
if ((batch_idx + 1) % accum_iter == 0) or (batch_idx + 1 == len(trainDL)):
            optimizer.step()
            optimizer.zero_grad()
```

64

When training complex models, it is generally necessary to use a small mini-batch size so as not to run out of memory. However, small mini-batch sizes can yield noisy weight/parameter updates. In order to combat this issue, the gradients can be accumulated over multiple mini-batches before the optimizer is applied and the weights/parameters are updated. This is known as gradient accumulation.

For example, you could keep track of the gradients and only apply weight/parameter updates and optimization after every 10 mini-batches. If the mini-batch size is 10 also, this means that 100 samples would be predicted before weight updates are applied.

This can be set up in PyTorch by augmenting the training loop. See the linked blog for a more in-depth explanation and examples.

As discussed earlier in the module, large mini-batch sizes can be problematic, so gradient accumulation should be used with caution.

Gradient Clipping

- Large gradients can cause:
  - Training instability
  - Loss spikes
  - Poor convergence
- Gradient clipping = rescales gradients if they exceed a threshold (like a governor on a gas pedal)
  - Compute L2 norm of all gradient tensors stacked together
  - If norm > 1.0 (or your threshold), multiply all gradients by 1.0 / grad_norm
  - This rescales the gradient vector to have magnitude exactly 1.0, preserving direction but capping magnitude
  - L2 Norm = square root of the sum of the squared gradients

65

Gradient clipping combats exploding gradients, training instability, loss spikes (or the loss being calculated as NaN, which breaks the training process), and poor convergence by rescaling them when their L2 norm exceeds a threshold.

The most common approach is L2-based normalization: if the L2 norm (the square root of the sum of squared gradients) exceeds a threshold (typically 1.0) all gradients are multiplied by threshold / L2_norm. This rescales the gradient vector to have magnitude equal to the threshold while preserving its direction.

Exponential Moving Average (EMA)

- Exponential moving average of model weights
- EMA Weight = [(1 - Decay) × Current Weight]+ [Decay × EMA Old Weight]
- Decay is typically 0.999 or 0.9999
- Why?
  - Temporal smoothing
  - Improve generalization
  - Training stability

66

Exponential moving average (EMA) maintains a shadow copy of the model weights that updates slower than the primary training model, providing a temporally smoothed version of the learned parameters.

During training, the EMA weights are updated after each gradient step according to the formula:

ema_weight = [(1 – decay) × current_weight] + [decay × ema_weight]

where decay is typically 0.99 or 0.999.

This design causes the EMA model to respond to consistent learning signals accumulated over many steps while damping out noise from individual batches, unstable gradient steps, or high learning rates.

At validation time, metrics are computed using the EMA model rather than the training model, which often yields better generalization because the smoothed weights have not overfit to recent training batches.

Mixed Precision Training

- Mixed precision training = lower-precision arithmetic (float16) for the forward pass and weight updates and higher precision (float32) for loss computation and gradient accumulation
- Why?
  - Reduce computational demand
  - Allow for larger mini-batch sizes

67

Mixed precision training leverages the speed and memory efficiency of lower-precision arithmetic (float16) for the forward pass and weight updates, while retaining higher precision (float32) for loss computation and gradient accumulation to maintain numerical stability.

Modern GPUs have specialized hardware that executes float16 operations significantly faster than float32 operations, and float16 tensors require half the memory, enabling larger mini-batch sizes for the same GPU memory budget.

This is less about improving model performance and more about reducing computational cost.

This is the end of this lecture module.

Please return to the West Virginia View Webpage for additional content.

Thanks! Hope you found this useful.