Geospatial Deep Learning

ANNs

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

WV VIEW

AmericaView
Empowering Earth Observation Education
AmericaView.org - Est. 2003

The goal of this module is to provide a conceptual overview of how artificial neural networks (ANNs) work, the components of ANNs, and how deep learning extends this framework. I have tried to minimize the math and focus on key concepts.

We will specifically focus on fully connected neural networks (FCNNs) as opposed to convolutional neural networks (CNNs), which will be covered in later modules.

Module Topics

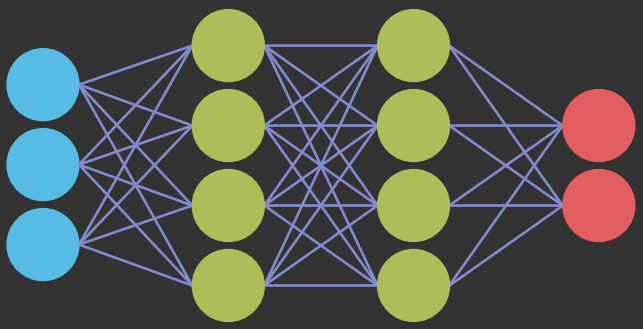
1. Artificial neural network structure
2. Weights and biases
3. Activation functions
4. Training process and terminology
5. Optimization algorithms
6. Combating overfitting

These are the topics covered in this module.

Conceptualizing ANNs

ANN Structure

- Consist of interconnected neurons or nodes
- Input layer represents predictor variables
- Output layer represents what is being predicted
- Model generated by learning a weight associated with each connection
- Referred to as fully connect architectures or densely connected architectures



4

ANNs are often described as being modeled after the human brain, as they consist of interconnect neurons or nodes. However, how these learning algorithms function is not necessarily comparable to the human brain, so I try to avoid using this analogy.

An ANN consists of neurons organized into layers as demonstrated on the slide. The Input layer represents input predictor variables, such as image bands, and has one neuron for each input, while the output layer represents the desired output, such as land cover categories, and has one neuron for each output class. For binary classification, one or two output neurons may be used. For regression, there will be one output neuron.

Between the input and output layers are one or more hidden layers containing multiple nodes. Within the network, all neurons in a layer are connected to the neurons or nodes in adjacent layers, and these connections have weights.

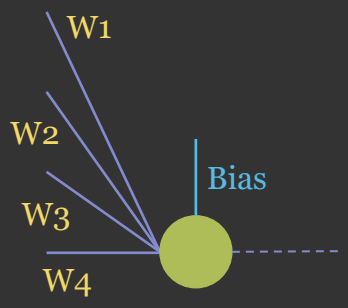
On the slide, the conceptualization contains three inputs (e.g., three image bands) and two hidden layers containing 4 nodes each. There are two classes being classified or predicted. Note that all nodes or neurons between adjacent layers are connected. These connections represent the weights.

Through the process of supervised learning, the weights in the network are

adjusted in an attempt to improve the prediction of the output.

Weights

- Weights are adjusted during training process
- Similar to slope or coefficient in regression
- Inputs from prior layer multiplied by weights
- Multiple Input X weight combinations are summed
- Weights relate to strength of activation or signal



$$\text{Activation} = b + \sum_{i=1}^{n} x_i w_i$$

5

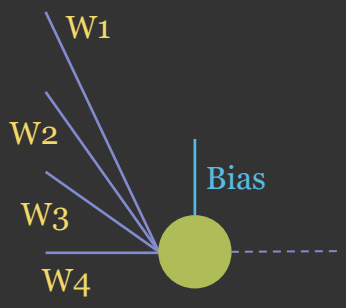
This slide further describes the concept of weights and how a signal or activation is produced at each node.

This is actually very similar to multiple linear regression. The weights are comparable to the coefficients for each predictor variable while the bias (b) is similar to the y-intercept.

So, at this point, the activation or output of the node is a linear combination of the inputs to the node, associated weights, and bias.

Bias

- A constant to shift the activation
- Similar to a y-intercept in linear regression

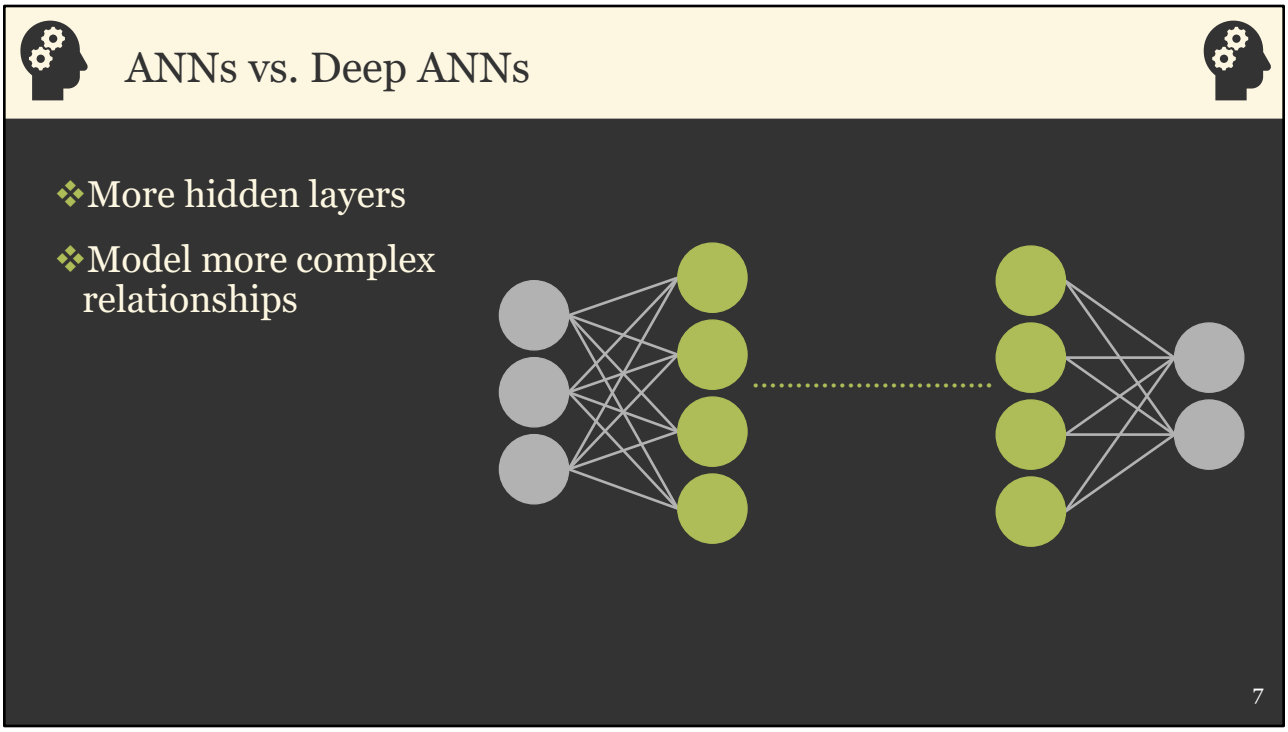


$$\text{Activation} = b + \sum_{i=1}^{n} x_i w_i$$

6

Again, the bias at each node is similar to the y-intercept in a linear regression equation. It is not multiplied by any input.

The bias allows the activation to be shifted to larger or smaller values, like an offset. The addition of a bias term further increases the flexibility of the model.



How is deep learning (DL) different from a traditional ANN? The key difference is the number of hidden layers, or layers between the input layer and output layer.

Deep learning algorithms generally have many hidden layers and associated nodes, as opposed to just one or two hidden layers.

So, you can think of "deep" as referring to the depth of the model or number of hidden layers.

Interestingly, it has generally been shown that incorporating more layers is more effective than having fewer layers with a large number of associated neurons. One reason why this might be the case it that a larger number of layers allows for more data abstraction than a small number of layers with a lot of nodes or neurons.

Strengths

- Model complex patterns/relationships
- Abstract data
- Less feature space engineering required
- Robust framework for a wide variety of tasks

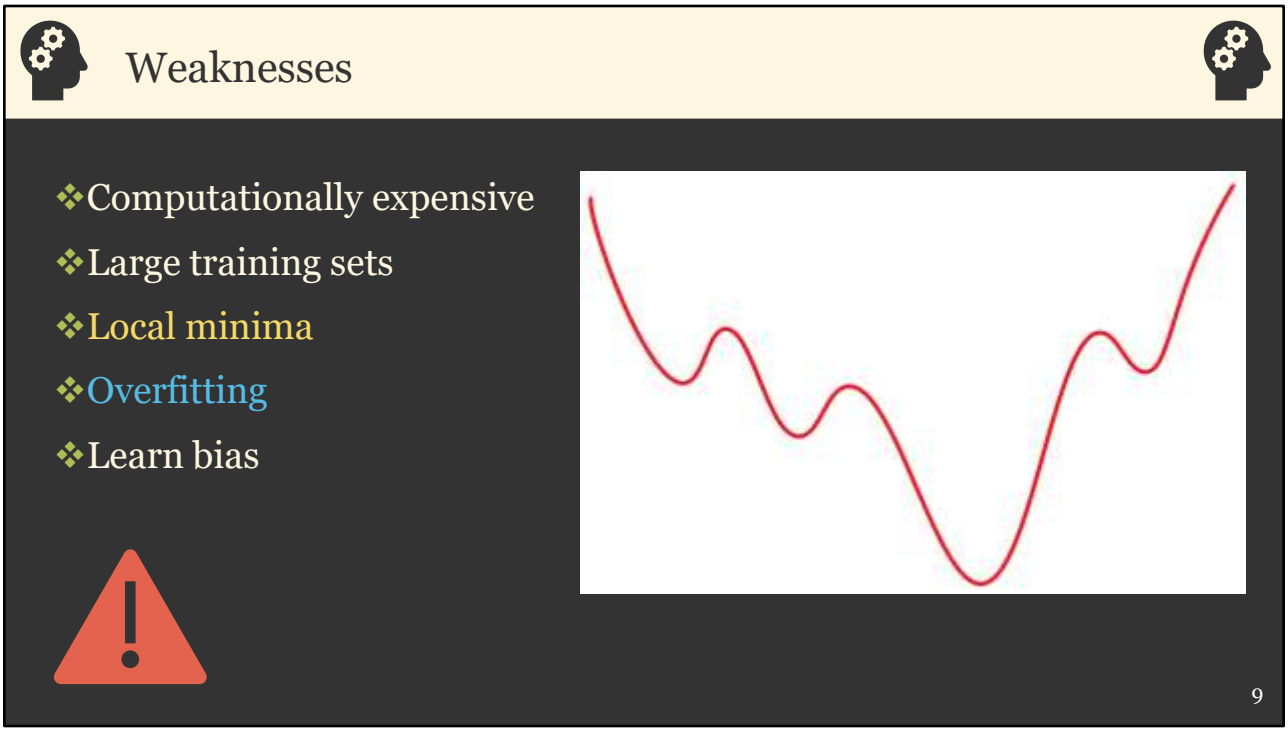
8

DL has taken off as it has been shown to perform well for a variety of predictive modeling tasks.

Although it is not always clear why DL algorithms can generate accurate predictions, it is generally suggested that this results from the algorithm's ability to model complex patterns and relationships in the input data due to the large number of hidden layers and learned weights/parameters. This allows for greater data abstraction to capture complex patterns and relationships.

Another strength of DL is that less feature space engineering is often required. Traditional machine learning methods often require the production of a variety of predictor variables to feed into the model. In contrast, DL models generally will accept a smaller number of input predictor variables and learn more complex features from the inputs using the hidden layers and weights. This reduces the amount of data pre-processing required.

Lastly, ANNs and DL is a general framework that can be applied to a wide variety of problems and tasks. It has been successfully applied to regression and classification problems. It has also revolutionized the field of computer vision, including fueling advancements in robotics and self-driving car technologies.



All modeling methods have weaknesses and complexities, and DL is no exception.

First, these models are complex and, as a result, computationally expensive. They tend to take a long time to train and require a lot of resources, such as memory. Some DL methods, such as those based on convolutional neural networks, cannot be ran efficiently on CPUs and require GPUs to perform the computations.

Generally, DL algorithms require a lot of examples or training data to learn complex patterns and not overfit. In fact, the availability of "big data" along with increased computational resources have been key in advancing DL over the past decade.

ANNs and DL suffer from the local minima problem. The image on the slide conceptualizes the problem. The lowest trough in the curve represents the optimal solution. However, there are other troughs in the curve, which represent local minima. During the process of training, the algorithm may get "stuck" in a local minima and fail to find the optimal solution.

DL algorithms can overfit to the training data. Overfitting means that the model does an excellent job predicting the training examples but does not

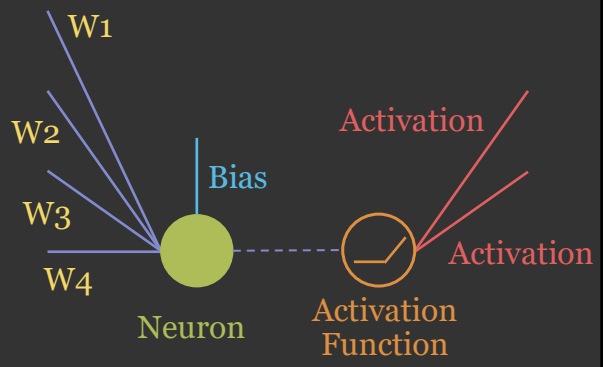
generalize well to new data. Training DL algorithms, and machine learning algorithms in general, is often a trade-off between reducing overfitting so that the model generalizes well to new data, but not generating a model that is too generalized and cannot model complex patterns in the data. This is why it is important to validate models using data that are not included in the training data set.

Lastly, DL algorithms, and machine learning methods in general, can generate a biased result or prediction if the training data are biased or not representative of the entire population. For example, if you are trying to predict where a tree species may occur on the landscape and you know that the species can be found in coves and in river bottoms, a model would likely be biased if only examples in coves were provided. Such issues can be very complex, especially when making predictions related to individuals, such as who is eligible for a mortgage to buy a home.

Activation Functions

Activation Functions

- Allow for modeling of nonlinear relationships
- A mathematical gate between inputs and outputs



$$\text{Activation} = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

11

I left out an important component above in the discussion associated with weights, bias, and activation at each neuron or node. As I mentioned, this is similar to a multiple regression equation where the bias acts as a y-intercept, the weights act as coefficients, and the inputs act as the predictor variable values. In reality, this is more than similar to a multiple regression equation: it is a multiple regression equation.

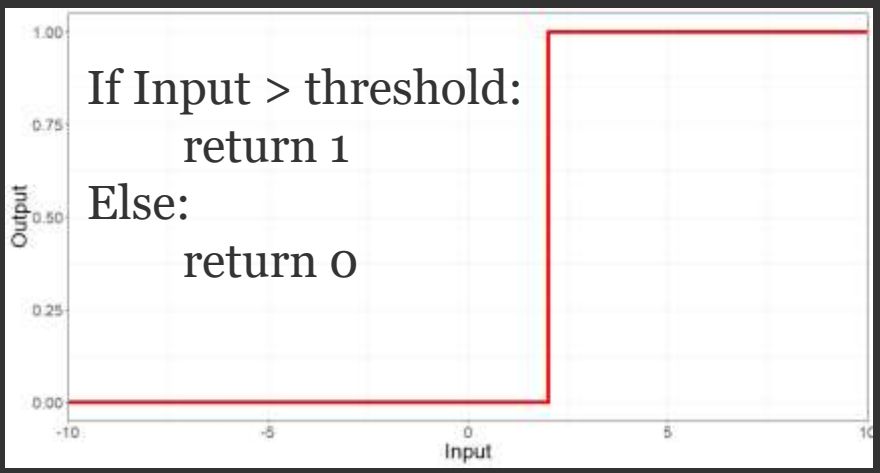
In order to be able to model complex patterns and relationships between variables and make predictions from a complex and noisy signal, it would be useful to be able to model or describe more complex patterns in the data.

This is accomplished by (1) having many interconnected nodes, associated weights, and hidden layers and (2) applying an activation function to transform the activation or signal (or, the output from the neuron). Over the next few slides, I will provide some examples of activation functions.

Note that just having multiple neurons and multiple hidden layers is not enough to model non-linear relationships. A series of linear relationships is still linear. Thus, activation functions are required to model non-linear patterns and relationships.

Binary Step

- Either neuron activates or does not
- Scaled from 0 to 1 ("on" and "off")
- Does not work with backpropagation (derivative 0, no gradient)
- Not commonly used



12

Perhaps the simplest activation function is a binary step. This activation function takes the values and transforms them into 0 and 1 or true and false. So, the neuron is either "on" or "off".

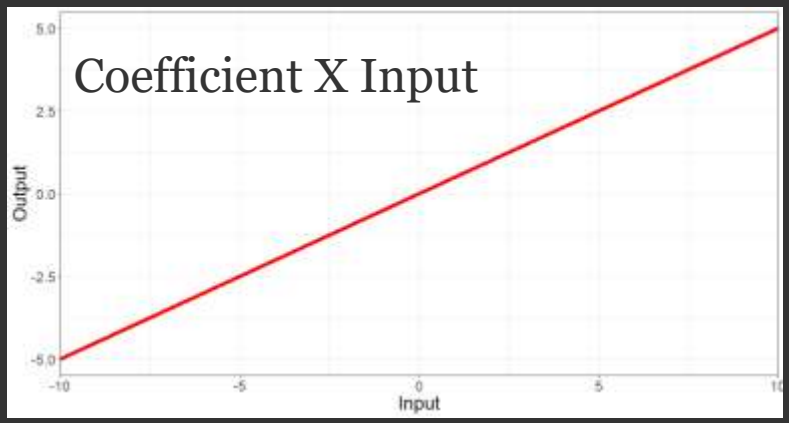
This can be thought of as a simple if…else statement. If the value meets a criteria, it is coded to 1, true, on, or full signal. If it does not, then it is coded to 0, false, off, or no signal. The only options are full activation signal or no activation signal. There are no gray levels.

As we will discuss later in this module, weights are updated using backpropagation, which relies on calculus and derivatives. The derivative of this equation would be zero, or no gradient. So, weights cannot be updated using backpropagation.

Due to its simplicity and incompatibility with backpropagation, this activation function is rarely used.

Linear

- Linear transformation
- Does not work well with backpropagation (derivative is a constant, no gradient)



13

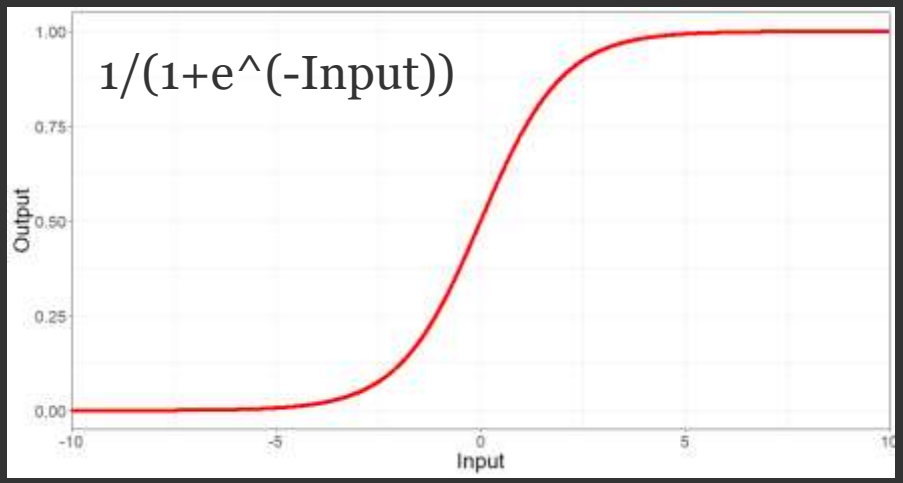
Values can also be linearly rescaled using a coefficient. Note that this does not fix the issue of modeling non-linear patterns, as the resulting activation function is still linear.

Also, the derivative of a straight line is a constant, so there is no gradient. Thus, backpropagation cannot be used to updated or learn the weights.

As a result, this activation function is also not commonly used.

Sigmoid or Logistic

- Nonlinear
- Scaled from 0 to 1
- Works with backpropagation
- Vanishing gradient problem
- Not zero-centered



14

A sigmoid or logistic function is used to rescale the data from 0 to 1. This is generally used when the goal is to differentiate between two groups or categories.

Weights can be learned when using a sigmoid function as there is a gradient. However, there are some issues. First, the slope approaches zero for high or low activations. This can result in a "vanishing gradient" problem, where the gradient approaches 0 and the model becomes difficult to train or update. It is also not zero-centered because the mean activation is not near zero; in the example, it is near 0.5 since the output is scaled from 0 to 1. This can be an issue, as it decreases strong positive and strong negative activations or signals.

In this class, you will primarily see sigmoid activation functions applied to positive case raw logits to rescale the values from 0 to 1.

Logistic or Sigmoid

- Logit = 2.6
- $1/(1 + e^{-2.6}) = 0.931$

1/(1+e^(-Input))

15

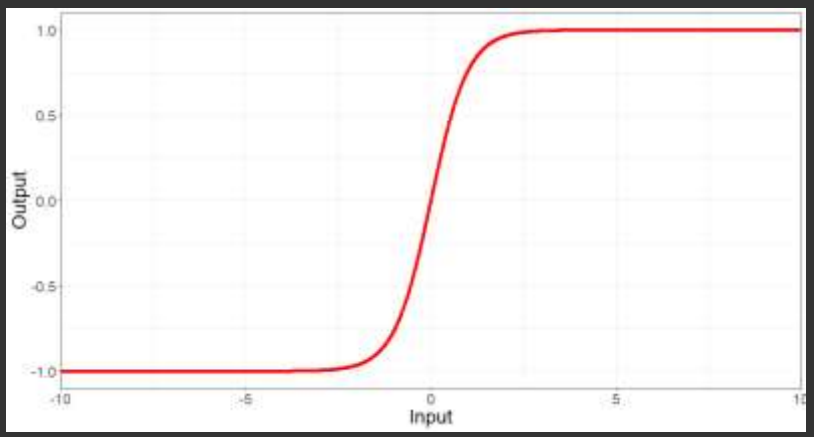
This slide provides an example of how a sigmoid activation is used to convert a raw activation or logit to an estimated probability value that is scaled between 0 and 1. In this case, the positive class probability is 0.931. Since probabilities must sum to 1, this means that the negative class probability is 0.069.

Note that there is some argument as to whether these values represent true probabilities. I prefer to use the term rescaled logit.

TanH or Hyperbolic Tangent

$$2*(1/(1+e^{(-(2*Input))}))-1$$

- Nonlinear
- Scaled from -1 to 1
- Works with backpropagation
- Vanishing gradient problem
- Zero-centered



16

TanH is an augmentation of the sigmoid activation function that is zero-centered. The output is scaled from -1 to 1 with a mean activation near 0.

However, this activation function still suffers from the vanishing gradient problem.

Softmax

- Nonlinear
- Similar to sigmoid, except allows for more than two classes
- Multiclass sigmoid
- Uses logit scores for multiple classes
- Used in final layer of multiclass classification models

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j))}$$

17

The sigmoid and TanH activation functions are commonly used in binary classification or when two categories are differentiated. The softmax activation extends this framework to allow for the differentiation of more than two class and is commonly used in multiclass classification problems.

In this class, you will commonly see this activation function applied to convert raw logits to rescaled logits or estimated class probabilities that sum to 1.

Softmax

- 3 Classes, B is correct class
- Logits = -1.5, 2.3, 1.8
- Denominator = $e^{-1.5} + e^{2.3} + e^{1.8} = 16.25$
- Est. Prob for A = $e^{-1.5}/16.25 = 0.014$
- Est. Prob for B = $e^{2.3}/16.25 = 0.614$
- Est. Prob for C = $e^{1.8}/16.25 = 0.372$
- 0.014 + 0.614 + 0.372 = 1.000

$$\frac{\exp(x_i)}{\sum_j \exp(x_j))}$$

18

This slide demonstrates how a softmax activation is used to convert raw logits for multiple classes to estimated probabilities or rescaled logits.

First, e is raised to the power of each of the raw logits then the results are added together. In order to then generate estimated probabilities or rescaled logits for each class, e is raised to the power of that logit then divided by the sum of e raised to a power of each logit. This results in a set of estimated probabilities or rescaled logits that sum to 1, with the predicted class being the one with the highest logit and associated estimated probability or rescaled logit.

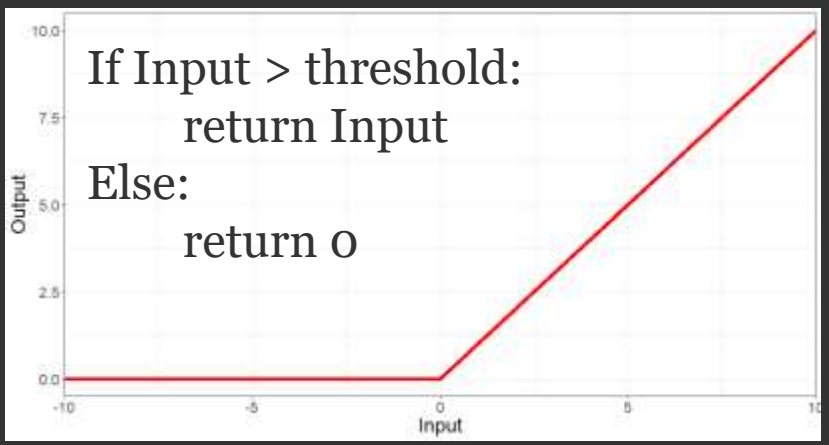
The example on the slide is for three classes. However, softmax will work with any number of classes other than 1. For a binary classification, the problem can be framed such that only one logit is predicted: the one associated with the positive class. In this case, you would use a sigmoid activation. If two estimated probabilities or rescaled logits are returned, one for the positive and one for negative class, then you would use a softmax activation. A sigmoid activation function cannot be used when more than two classes are differentiated. Also, both sigmoid and softmax are not applicable to regression problems.

Again, there is some debate as to whether or not applying a softmax or sigmoid function to raw logits generates true class probabilities. Although I will use the

term probability to describe these rescale values, the term rescaled logits may be more appropriate. However, it is common for these values to be referred to as probabilities.

Rectified Linear Unit (ReLU)

- Nonlinear
- More computationally efficient than sigmoid or tanh
- Widely used
- Not zero-centered
- Not all neurons updated during backpropagation due to 0 gradient (dying ReLU)



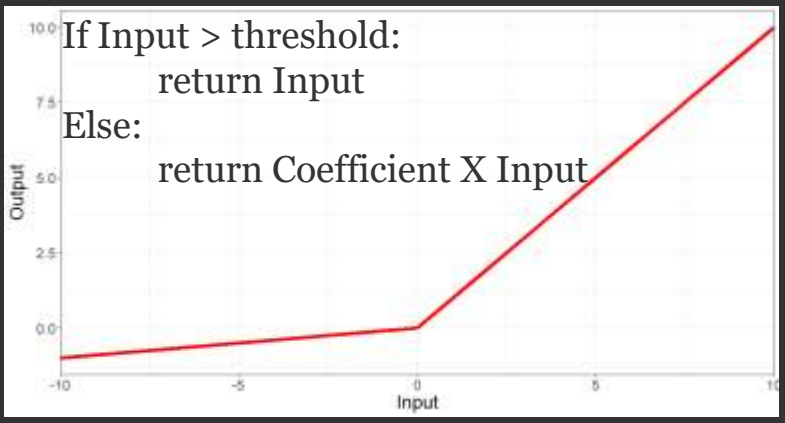
19

The rectified linear unit (ReLU) activation function is currently commonly used. It allows for an activation of 0 below a certain threshold and a linear activation above a certain threshold. It is also more computationally efficient than sigmoid and TanH. Generally, negative values are converted to 0 while positive values are maintained.

This activation function does have some issues. First, it is not zero-centered and, second, since part of the function has a slope or gradient of 0, this can cause issues with backpropagation known as the “dying ReLU” problem. Despite some issues, it is still commonly used.

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



20

In order to combat the "dying ReLU" issue, Leaky ReLU maintains a gradient by applying a coefficient to the part of the function that has a slope of 0 when using ReLU. Thus, the gradient does not become zero, but changes gradually. However, the magnitude of negative activations is reduced.

Parameterized Rectified Linear Unit (Parameterized ReLU)

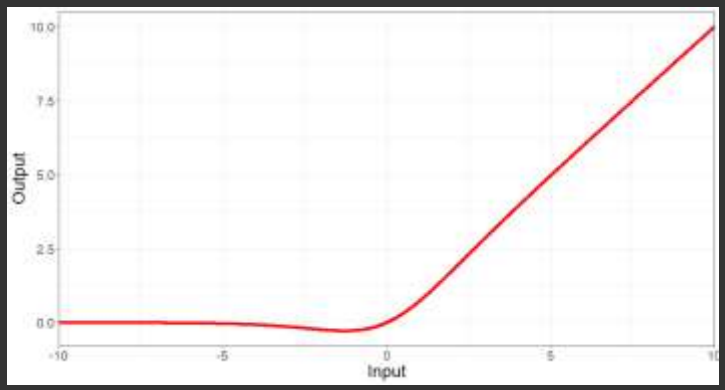
- Like Leaky Relu, accept coefficient of negative component is trainable
- Fix dying ReLU

21

The parameterized rectified linear unit activation is similar to leaky ReLU except that the coefficient of the negative component is trainable by the model.

Swish

- Input X Sigmoid(Input)
- Nonlinear
- Easier to converge while training compared to sigmoid
- Computationally expensive

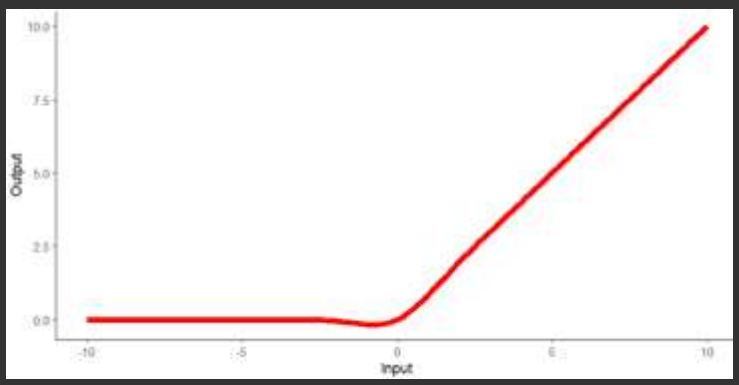


22

The swish activation function is similar to sigmoid accept that it does not suffer from the vanishing gradient problem. It has similar computational efficiency to ReLU.

GELU

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



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

23

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.

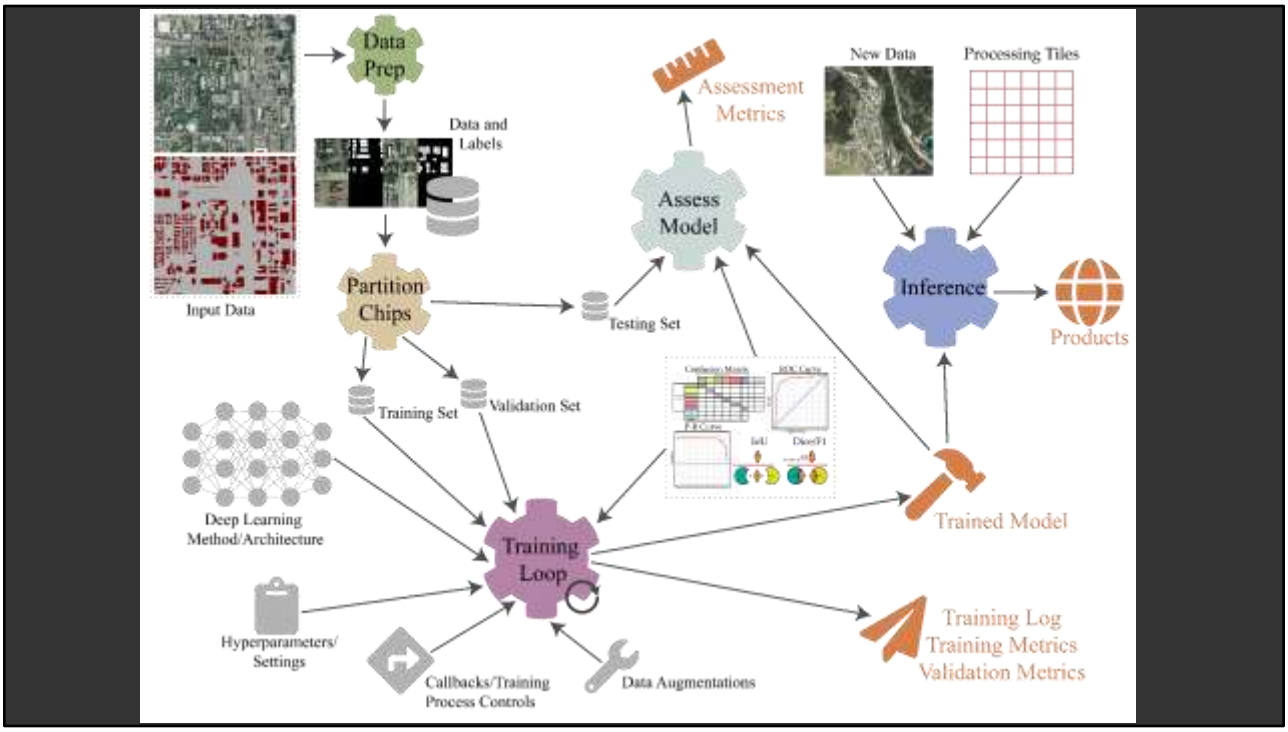
What activation function to use?

- Sigmoid or Tanh should be avoided due to vanishing gradient problem
- However, sigmoid is commonly used as the activation for the last layer in the network when performing binary classification
- Softmax is used for last layer of multiclass classification
- ReLU is a good default
- Use Leaky or Parameterized ReLU to avoid dying ReLU problem
- Modern architectures commonly use GELU

24

This slide provides some suggestions for selecting an activation function. Note that ReLU is currently one of the most commonly applied activation functions. GELU is used in some modern architectures.

The Process



The image on this slide conceptualizes the learning process.

First, input data, in this example image chips and associated labels, must be generated then partitioned into separate training, validation, and testing sets. The training data are used to train the model while the validation data are used to assess the model at the end of each training epoch, or iteration over the training data. Once a final model is obtained, it can then be assessed on the withheld testing data.

Other than preparing data, the analyst will also have to select an appropriate architecture for the problem or use case. The base architecture may need to be augmented for specific tasks. The user must also define hyperparameters and other settings or components (for example, the optimizer, learning rate, loss metric, and accuracy metrics). It is also possible to define schedulers or callbacks, such as manipulating the input data or hyperparameters between training epochs or logging the loss and accuracy metrics.

Once a final model has been obtained, it can be assessed using the withheld testing data and, if deemed accurate or useful enough, be used to infer to new data. For example, all images covering a desired spatial extent could be predicted to produce a land cover map.

Terminology

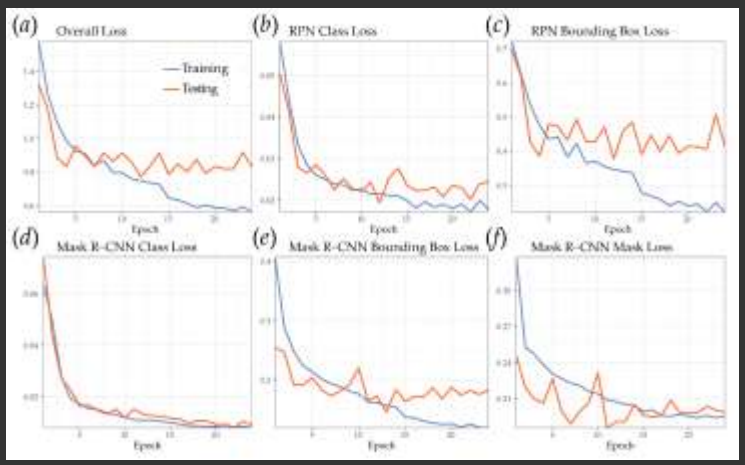
- Training = process of learning from the provided examples/training data
- Assessment = use withheld data to assess model performance
- Inference = use trained model to predict to new data
- Training Set = samples used to train the model
- Validation Set = samples used to assess the model at the end of each training epoch
- Testing Set = withheld data used to assess the final model
- Optimizer = algorithm used to update weights/parameters
- Loss Metric = metric or metrics used to guide optimizer in the learning process (e.g., binary cross-entropy, cross-entropy, or MSE)
- Assessment Metrics = metrics monitored to assess model performance but not used by the optimizer to update parameters (e.g., overall accuracy, precision, recall, F1 Score)
- Epoch = one iteration over the training set
- Batch or Mini-Batch = subset of data; weight/parameter updates performed after processing each training mini-batch
- Hyperparameters = settings provided by the user and not learned in the training process (e.g., learning rate)
- Data Augmentations = manipulation of input data to reduce overfitting (e.g., randomly flip images)
- Callback = makes change to training process at end of a training mini-batch or epoch, often based on some criteria (e.g., save model if there was an improvement over prior model or change learning rate after 25 epochs)
- Overfitting = model does not generalize well to new data because it has learned to memorize the training samples

27

This slide provides some common terminology associated with the learning process.

Loss Functions

- Measure of error
- Used to assess learning during training
- Goal is to update weights/parameters to minimize the measure
- Use different measures for different situations
- Can create custom measures or combine multiple measures



28

The loss function is a metric that is monitored to determine if predictions are improving or if the network is learning. The goal is to minimize or decrease the loss over multiple iterations, or epochs, over the training data. Since loss is to be minimized, it is commonly a measure of error as opposed to accuracy.

Loss metrics are very important, as they guide the learning process and the weight/parameter updates. We will discuss different loss functions in the next module.

Confusion Matrix or Error Matrix

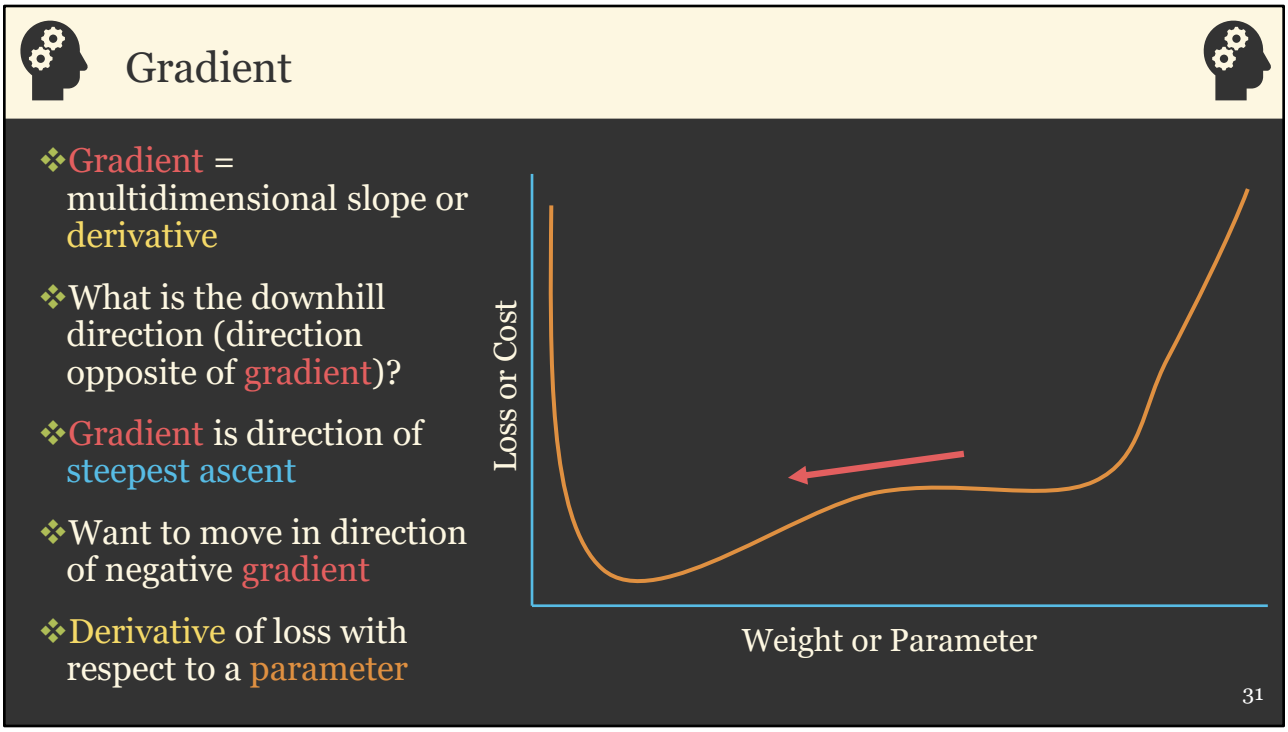
		Reference Data							
		Forested	*Pasture/ Grass*	*Barren*	*Cropland*	*Developed*	*Water*	Total	User's Accuracy
Classified Data	*Forested*	91	8	1	11	0	2	113	81%
	Pasture/ Grass	4	81	0	15	0	0	100	81%
	Barren	0	0	87	2	10	2	101	86%
	Cropland	3	11	0	69	0	0	83	83%
	Developed	0	0	10	0	73	0	83	88%
	Water	2	0	2	3	17	96	120	80%
	Total	100	100	100	100	100	100		
	Producer's Accuracy	91%	81%	87%	69%	73%	96%		

29

Other than just monitoring or reporting the loss, it is also generally useful to assess the model using accuracy assessment metrics. For example, assessment metrics can be calculated for withheld validation data at the end of each training epoch, or iteration over the training samples. The final model can be assessed relative to withheld testing samples.

We will discuss accuracy assessment metrics for different use cases in the next module.

Optimizers



When a neural network architecture is first defined, the weights or trainable model parameters will be randomly initialized. Since these parameters are purely random, it is expected that the model will not predict the data well. The goal of the learning process is to update the weights/parameters to improve the predictive power of the model. This is accomplished by calculating the loss then updating the weights/parameters to reduce the loss metric.

In the example graph, the x-axis represents a value or parameter being learned while the y-axis represents the loss. The orange line conceptualizes how the model performs, as measured by the loss, using different values of the learned parameter. The best parameter is the one that provides the lowest loss, which corresponds to the lowest point on the curve. Note that this conceptualization can be extended into more dimensions to represent more trainable parameters. For example, this surface could be modeled in two dimensions if there are two trainable parameters. Although we cannot visualize this surface when there are a large number of trainable parameters, it can be mathematically described.

Unfortunately, the value for the parameter that provides the lowest loss is not known beforehand. So, training data must be used to estimated the optimal parameter value during the learning process. Updates to this parameter are made using the derivative of the loss with respect to the parameter. The goal is

to update the weights or parameters in the opposite direction of the gradient, or opposite to the maximum ascending slope, to further minimize the loss.

The concept of a gradient is similar to a derivative except that multiple parameters must be considered. In other words, you can think of gradients as multidimensional derivatives. Before making weight or parameter updates, the derivative of the loss with respect to each trainable parameter must be calculated. The parameters are iteratively updated using an optimization algorithm in an attempt to minimize the loss. This is accomplished using derivatives and the chain rule.

Optimizers

- Algorithm used to adjust/update weights during training process
- Goal is to minimize the loss
- Update parameters in the negative gradient direction to minimize loss
- Based on derivatives and the chain rule

32

The gradient decent algorithm is used to update the network parameters and weights in an attempt to minimize the loss function during the process of supervised learning.

Derivatives and the chain rule are used to determine the gradient, or whether the model is moving toward the minimum or away from it. This knowledge can then be used to update the weights/parameters so that the model moves towards the minimum to decrease the loss. Since there are many weights/parameters that are calculated, the slopes, derivatives, or gradients are interrelated. So, this problem must be solved using the chain rule.

Although the mathematics of the optimization process are outside of the scope of this course, the important point here is that algorithms can be used to improve the loss over multiple learning iterations over the training set to gradually reach or approach the optimal solution or minimum loss (i.e., global minimum). This requires that all functions or operations in the large number of functions or operations that are performed be differentiable.

Backpropagation

- Use derivatives and chain rule to calculate contribution of each trainable parameter to the total loss

33

This process relies on backpropagation where errors are propagated backward through the network to mathematically understand their source. In other words, the gradient of the loss with respect to each trainable parameter is determined. This allows for weights/parameters to be adjusted at the correct locations to improve the model.

Optimization Algorithms

- Gradient Descent
- Stochastic Gradient Descent
- Mini-Batch Gradient Descent
- Mini-Batch Gradient Descent with Momentum
- Nesterov Accelerated Gradient (NAG)
- Adaptive Gradient Algorithm (Adagrad)
- Adadelta
- Root Mean Square Propagation (RMSProp)
- Adaptive Moment Estimation (Adam)
- Adaptive Moment Estimation with Decoupled Weight Decay Regularization (AdamW)
- Nesterov Accelerated Adaptive Moment Estimation (Nadam)

https://keras.rstudio.com/reference/index.html

34

A variety of optimization algorithms are available that solve this problem using different methods.

Currently, RMSProp, Adam, and AdamW are commonly used. However, this is still an active area of research, so augmented or new optimization methods will likely be made available in the future.

Gradient Descent

- Gradient Descent = use entire training set to guide weight updates
- Stochastic Gradient Descent = only use one sample to guide each weight update
- Mini-Batch Gradient Descent = use a subset or mini-batch of the data to guide each weight update

35

The most commonly used optimization algorithms build on gradient descent. Gradient descent uses all of the training samples to guide the weight updates. In contrast, stochastic gradient descent uses only a single training sample to guide each weight update. Mini-batch gradient descent breaks the data into a set of partitions, or mini-batches, which are then used to guide each weight update.

It is most common to use mini-batch gradient descent. For example, if you have 10,000 training samples available, you may split them into 100 mini-batches each containing 100 samples. Each of the mini-batches will then be fed through the model, losses will be calculated using the predictions and provided labels, backpropogation will be performed, and the optimization algorithm will be used to update the trainable parameters.

A single iteration over all of the training mini-batches sequentially is called a training epoch. Again, when using mini-batch gradient descent, the weights/parameters are not just updated at the end of the entire training epoch but after processing each mini-batch. This is generally more computationally efficient than gradient descent and less noisy than stochastic gradient descent. Also, it can help minimize overfitting in comparison to gradient descent.

Generally, when users indicate that they implemented stochastic gradient

descent, what they really mean is that they used mini-batch gradient descent.

Learning Rate

- Learning Rate = hyperparameter that specifies how large a step is taken to reduce the loss
  - High values = may jump over minima or fail to model complex patterns
  - Low values = take too long to converge or get caught in local minima

New Value = Old Value – (Learning Rate * Gradient w.r.t. parameter)

36

Optimizers rely on user-defined settings, and the learning rate is generally considered to be one of or the most important setting.

The learning rate specifies how large a step is taken to update model parameters at each weight/parameter update. If this value is high, this may allow the model to jump over or not get stuck in local minima; however, the model may fail to model complex patterns in the data. If it is too low, it may take a long time for the model to converge or stabilize. Further, there is a greater chance of the model getting stuck in a local minima. The best setting is case specific.

The learning rate controls how much a parameter can change with each update. To calculate the new parameter, you multiply the learning rate by the gradient and subtract this from the old parameter value. So, a larger gradient and a larger learning rate can both yield a larger change in the parameter value during an update.

A local minima can be thought of as a low point or trough in the loss surface that is not the lowest point, or global minima. Thus, if the optimizer gets stuck in a local minima, it is not able to return or find the optimal model parameters.

Learning Rate Schedulers

- Use schedulers to change the learning rate
- Change learning rate after a defined number of epochs
- Change learning rate if loss is not changing
- Change learning rate based on a mathematical function
- Early Stopping = stop training early if no improvement in loss

37

It is also possible to manually change the learning rate during the learning process using call backs or schedulers. For example, you can change the learning rate after a defined number of training epochs or if the loss has not improved after a given number of weight/parameter updates. You can also define learning rate schedulers that change the learning rate using a mathematical function, such as oscillating using cosine annealing or linearly increasing or decreasing using a slope term.

You may choose to halt the training process early if the model has not improved after a given number of training epochs based on either the training loss, validation loss, or an assessment metric calculated from the training or validation data. Using the validation data to gauge when to stop the training process can be useful for avoiding overfitting to the training data.

We will discuss learning rate schedulers in more detail in a later module.

Mini-Batch Gradient Descent

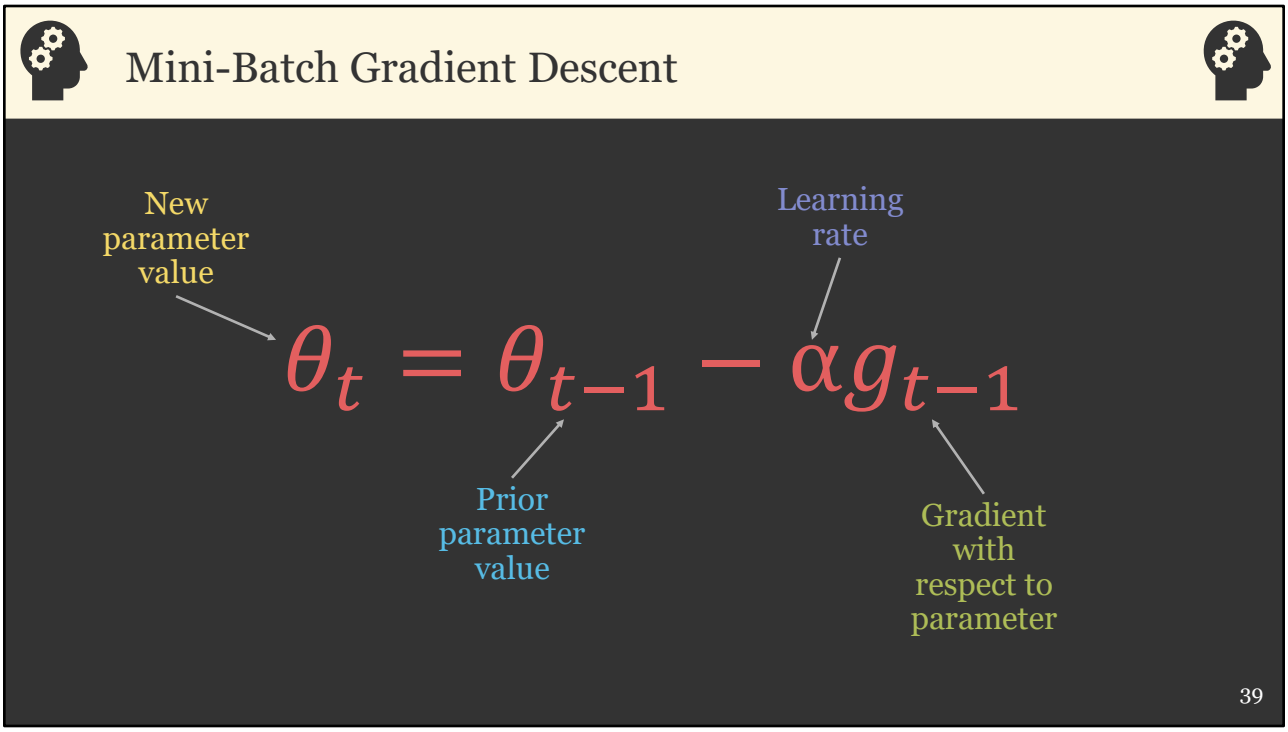
- Start at random weights/parameters
- Weights/parameters updated after each training mini-batch
- Iteratively adjust the weights/parameters to descend to lowest value of loss, cost, or objective function
- Calculate the step sizes for each weight/parameter as (Learning Rate * Gradient w.r.t. parameter)
- New Value = Old Value – (Learning Rate * Gradient w.r.t. parameter)
- Repeat process until reach stopping criteria

38

This slide provides a general overview of how mini-batch gradient descent is implemented.

The neural network architecture is initialized with random weights or parameters. The training data are fed through the model as mini-batches. The resulting predictions and provided labels are then used to calculate the loss. Backpropagation is performed to estimated the gradient of the loss with respect to each trainable parameter. The weights/parameters are then updated using the optimization algorithm. The magnitude of the update depends on the learning rate and the gradient. One iteration over all training mini-batches is called a training epoch.

The learning process will iterate over the entire training set multiple times, or over multiple epochs. The learning process will stop once a stopping criteria has been met. Common stopping criteria include (1) reaching a specified loss, (2) reaching a specified number of training epochs, (3) reaching a specified loss or assessment metric value for the withheld validation data, or (4) seeing no improvements in the loss or model performance for a given number of epochs.



This slide provides the equation for mini-batch gradient descent. Theta represents a trainable parameter in the model. The prior value of theta is augmented by subtracting the gradient with respect to that parameter multiplied by the learning rate.

Again, a larger gradient with respect to that parameter and/or a larger learning rate will result in a larger update magnitude.

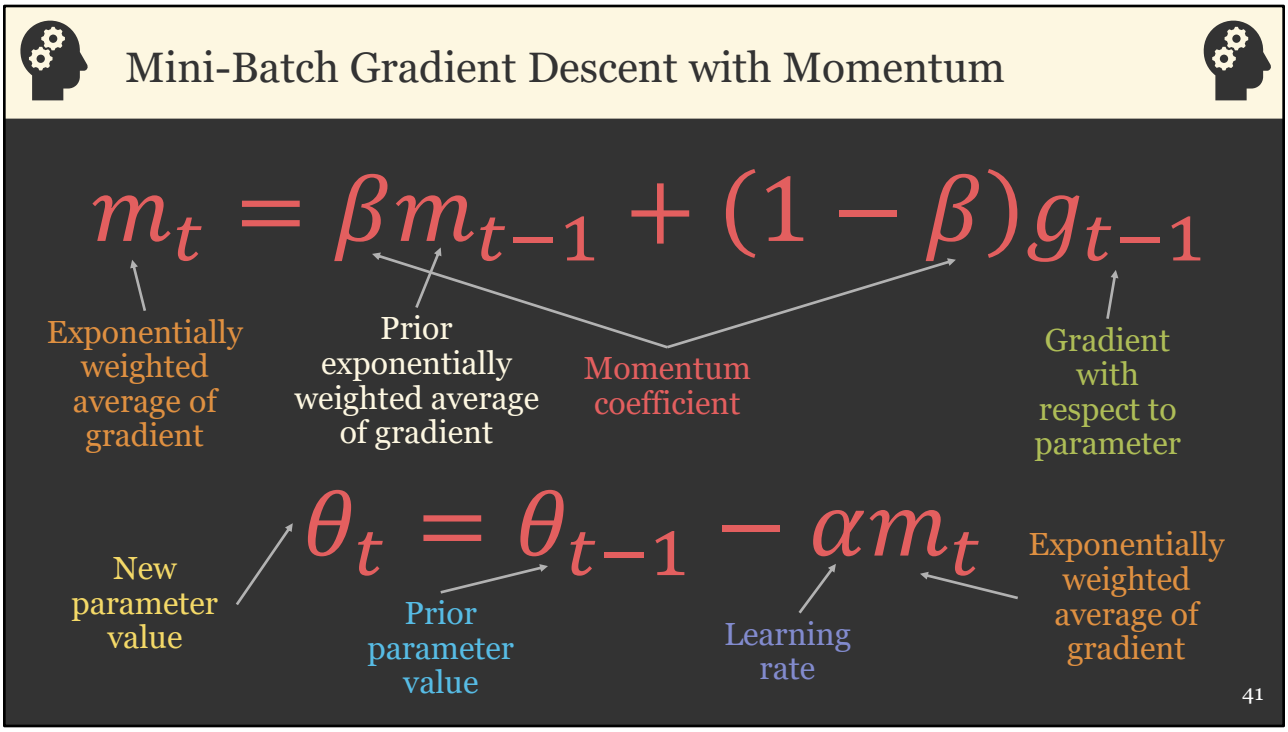
Mini-Batch Gradient Descent with Momentum

- Accumulate momentum
- Momentum term increases for parameters whose gradients point in the same direction
- Reduces updates for parameters whose gradients change directions
- β
  - β = 0 then equivalent to mini-batch SGD
  - >β yield higher momentum

40

Most optimizers are simply advancements or augmentations of mini-batch gradient descent. One issue with gradient descent is that it only considers the current gradient. In order to allow for prior gradients to inform the current parameter updates, a momentum term can be added. Momentum is basically just an exponentially weighted average over the past gradients. If the gradients are large on average, this will allow for larger parameter updates. When gradients are smaller and/or have varying directions, weight updates will be smaller.

In physics, the concept of momentum relates to a moving objects ability to change speed and/or direction. Larger or more massive objects traveling at a higher velocity or speed have more momentum. As a result, these objects are harder to stop or change their direction of motion. This same concept applies here. With larger weighted average gradients, the parameters will be easier to update and less likely to not get stuck or not update. This is one means to not get stuck in a local minima or at a “saddle” point.



This slide describes the equations underlying momentum.

Instead of using only the current gradient, an exponentially weighted average of all prior gradients is used. This is accomplished by multiplying the prior exponentially weighted average of the gradients by a beta term and adding 1 minus beta multiplied by the current gradient with respect to the parameter. Larger values of beta result in putting more weight on the prior gradients. If beta is 0, the first part of the equation drops out, and only the current gradient is considered. In this circumstance, the result is equivalent to traditional mini-batch gradient descent.

In the second equation, which is used to calculate the new parameter value, the gradient is now replaced with the exponentially weighted average of the prior gradients. Other than that, the equation is the same as that for mini-batch gradient descent.

Note that the first parameter update will be equivalent to the update made with mini-batch gradient descent since there is no prior gradient to consider.

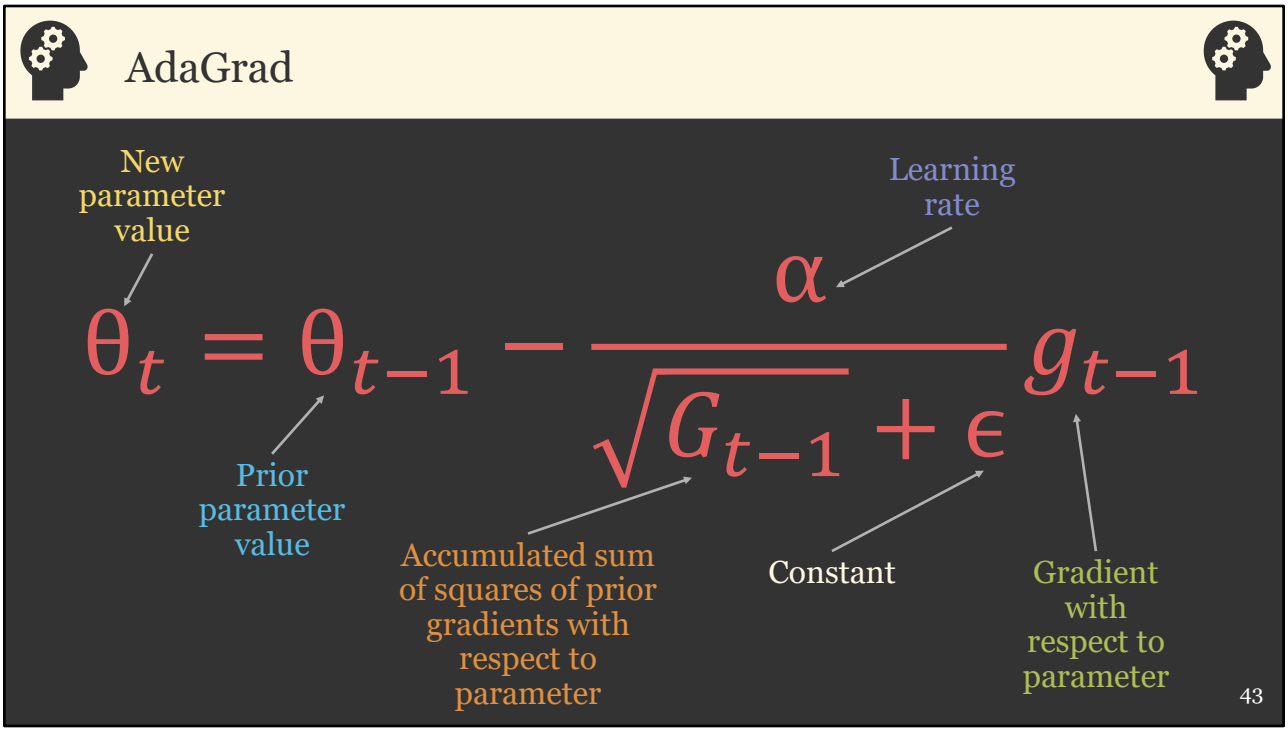
AdaGrad

- Adaptive Gradient
- Considers the sum of squares of the prior gradients with respect to the parameter of interest for all prior parameter update iterations
- Continually accumulating the gradient over an increasing number of iterations will continue to lower the effective learning rate, which results in a slowing of the learning process and a risk of getting caught in a local minima

42

AdaGrad, or Adaptive Gradient, is on means to implement an adaptive learning rate. Using this method, the learning rate is augmented by dividing by the square root of the accumulate sum of squares of all prior gradients with respect to the parameter of interest. Practically, this results in the effective learning rate decreasing as the learning process progresses.

One issues with AdaGrad is that continually accumulating the gradients over an increasing number of iterations will continue to lower the effective learning rate, which can result in the learning processing slowing as the magnitude of parameter updates gets smaller. This can result in more likelihood of getting stuck in a local minima.



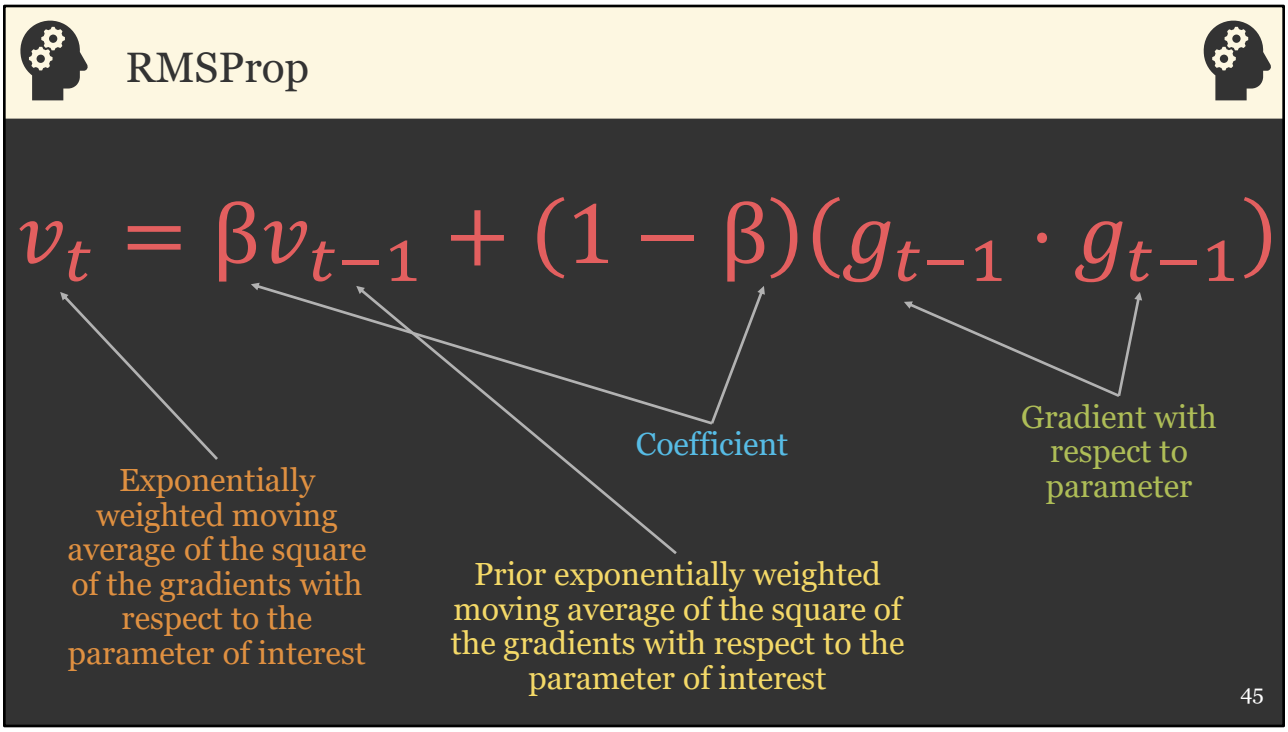
This slide describes the math of AdaGrad. The learning rate is divided by the accumulated sum of squares of the prior gradients with respect to the parameter. Also, a small constant is added to enhance numeric stability and avoid divide-by-zero errors. This augmented learning rate is then multiplied by the current gradient with respect to the parameter of interest, which is then subtracted from the current parameter value to obtain the new parameter.

RMSProp

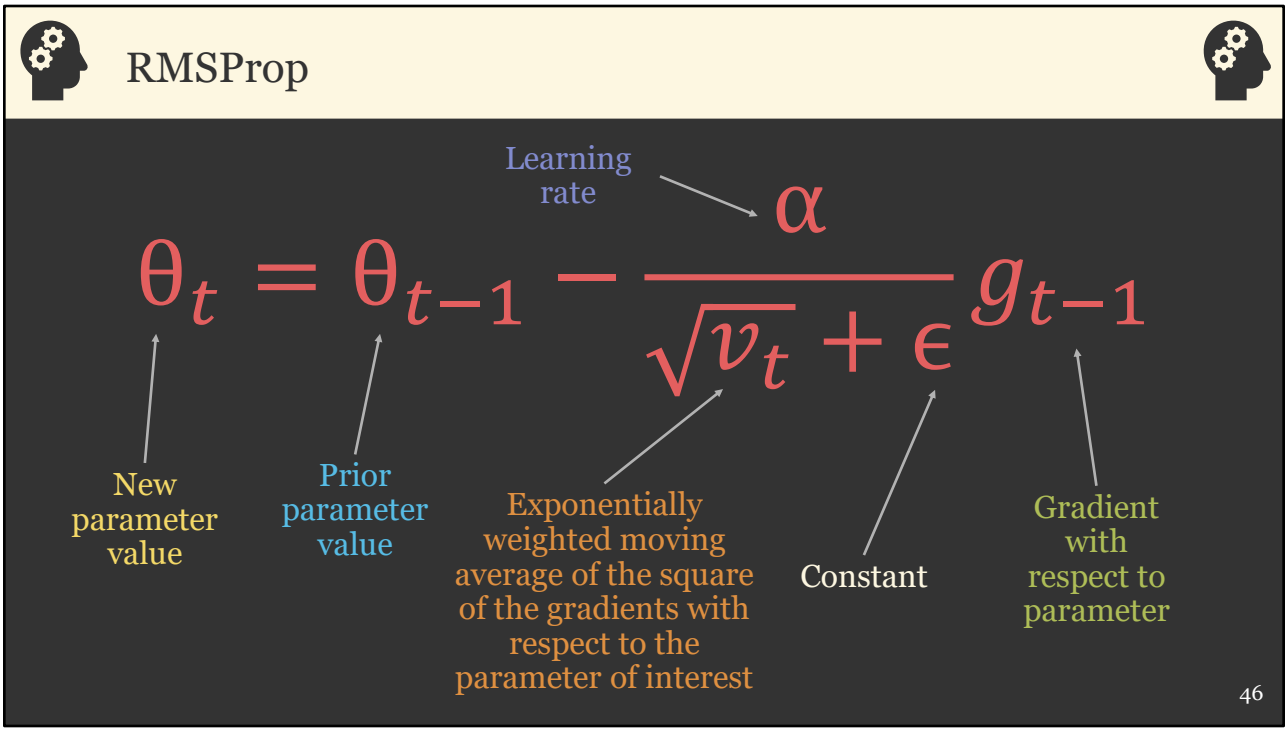
- Root Mean Square Propagation
- Incorporate a weighted average of the square of the prior gradients for each parameter such that more recent gradients have a larger impact on the augmentation of the learning rate.
- β controls the relative weight of prior gradients
  - β = 0 then equivalent to mini-batch SGD
  - >β yield higher impact/consideration of prior gradients

44

RMSProp, or root mean square error propagation, offers another means to adapt the effective learning rate. It incorporates a weighted average of the square of the prior gradients with respect to the parameter as opposed to the square root of the accumulate sum of squares of all prior gradients with respect to the parameter of interest. It also includes a beta term that controls the relative weighting applied to the prior gradients.



The equation on this slide explains how the exponentially weighted moving average is calculated within the RMSProp method. The prior exponentially weighted moving average of the gradient with respect to the parameter is multiplied by beta. If beta is zero, the prior exponentially weighed moving average is not considered. One minus beta is multiplied by the square of the current gradient with respect to the parameter of interest. This is then added to the prior exponentially weighted moving average of the square of the gradients.



In order to augment the effective learning rate, it is divided by the square root of the exponentially weighted moving average of the square of the gradients. A constant term is used to avoid divide-by-zero errors and to maintain numeric stability.

The augmented learning rate is then multiplied by the current gradient with respect to the parameter and subtracted from the parameter value to obtain the new parameter value.

In short, RMSProp implements a different means to incorporate the prior gradients in comparison to AdaGrad. Note that RMSProp avoids the continuously accumulating gradient issue inherent to AdaGrad.

Adam

- Adaptive learning rates for each parameter separately
- Uses exponentially decaying average of past squared gradients
- Keeps track of multiple past gradients and incorporates momentum
- RMSProp + Momentum
- $\beta 1$ = momentum component
- $\beta 2$ = relative weight of prior gradients

Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

47

Adam is currently one of the most commonly used optimization algorithms. It is essentially a combination of RMSProp and momentum.

It has two user-defined terms. Beta 1 is used within the momentum component while beta 2 is used within the relative weight of prior gradients, or adaptive learning rate, component.

Adam

Exponentially weighted moving average of the gradient w.r.t. parameter calculation for momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Moving average of the squared gradients w.r.t. parameter calculation for adaptive learning rate

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias corrections

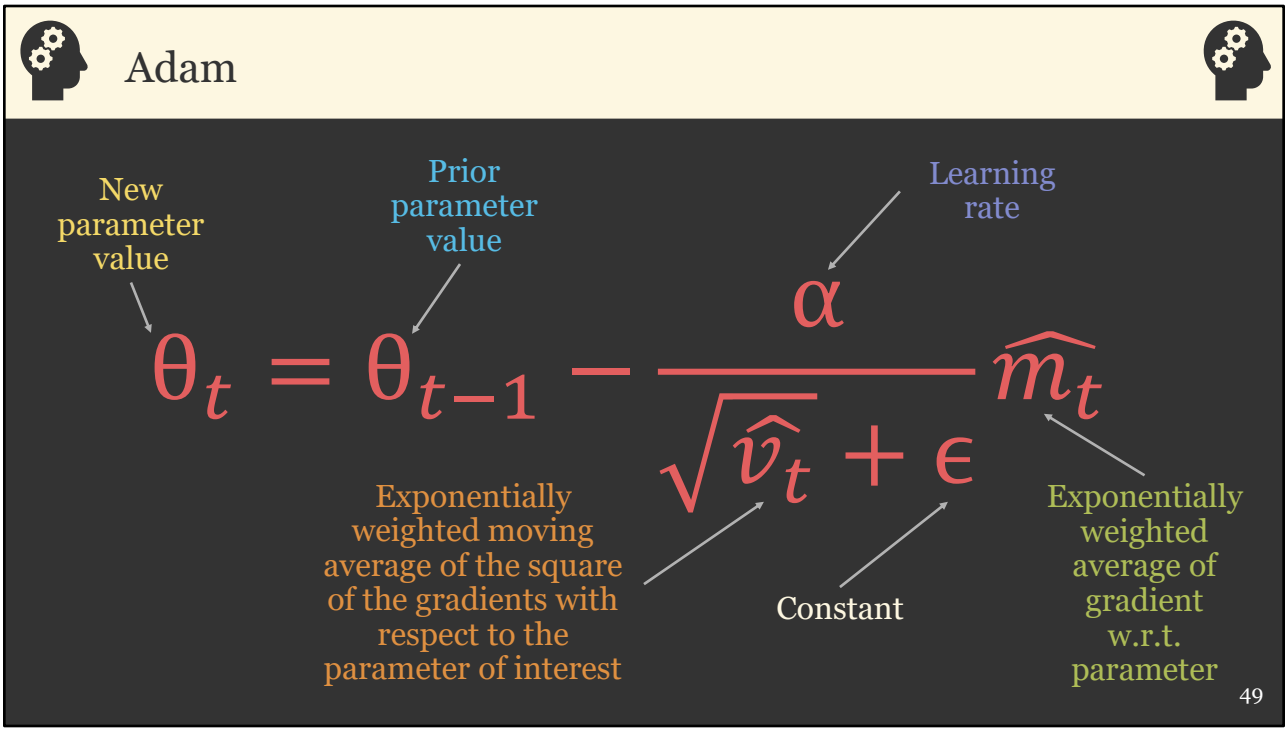
$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t} \qquad \widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

48

The first equation summarizes the momentum component. Note that this is the same implementation of momentum discussed above. If beta is zero, the prior momentum term is not considered, and the result is equivalent to mini-batch gradient descent.

The second equation is the same as the exponentially weighted moving average calculation used for RMSProp.

Adam also implements a bias correction for the two components relative to the defined beta terms and the current update iteration.



Once the momentum and exponentially weighted moving average, or adaptive learning rate, components are calculated, they can be used in the calculation of the new parameter value.

As with RMSProp, the learning rate is divided by the square root of the exponentially weighted moving average of the prior squared gradients with a constant term added for numeric stability. As opposed to multiplying the result by the current gradient with respect to the parameter, it is multiplied by the exponentially weighted average of the prior gradients. The result is then subtracted from the current parameter value to obtain the new parameter value.

Weight Decay

- Aid in generalization/reduce overfitting
- Regularization (L_2)
- Add prior parameter value to the current gradient
- In order to decrease the impact of adding the prior parameter value, it is multiplied by λ
  - $\lambda = 0$ yields no regularization
  - $> \lambda$ yields more regularization

Gradient with respect to parameter

Lambda hyperparameter

Prior parameter value

$$g_{t-1} = g_{t-1} + \lambda\theta_{t-1}$$

50

Another common augmentation is to implement weight decay as a means to combat overfitting. This is a form of L_2 regularization, similar to ridge regression.

When implementing weight decay, the prior parameter value, multiplied by a user-defined lambda term, is added to the current gradient with respect to the parameter to augment the current gradient.

If lambda is zero, then no regularization is applied. Larger values of lambda yield more regularization.

AdamW

- Loshchilov and Hutter (2017) suggest that weight decay is equivalent to L2 normalization when used along with optimizers that do not incorporate adaptive learning rates, such as mini-batch gradient descent
- This is not the case for optimizers, such as Adam
- Suggest an alternative means of implementing weight decay
- Generally preferable to use AdamW if weight decay is being implemented ($\lambda \neq 0$)

$$g_{t-1} = g_{t-1} + \alpha\lambda\theta_{t-1}$$

Loshchilov, I., Hutter, F., 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101.

51

Loshchilov and Hutter (2017) argues that the original implementation of Adam is biased when weight decay is applied. As a result, they suggest an altered form that corrects for this bias when Adam is used with weight decay. This augmentation is generally referred to as AdamW. If weight decay is being used along with Adam, we suggest using AdamW as opposed to Adam. AdamW without any weight decay applied is equivalent to Adam. So, using AdamW is not necessary if there is no weight decay applied.

Overfitting

Overfitting

- Learns patterns that are too specific to training data
- Does not generalize well to new data
- Balance between overfitting and overgeneralization
- Caused by:
  - Complex models
  - Small training datasets
  - Training for too many epochs
  - ANN architecture and design issues

53

Again, training deep learning models is often a tradeoff between overfitting, where the model predicts the training data well but does not generalize to new data, and overgeneralization, where the model is too general and fails to model the complexity of the data.

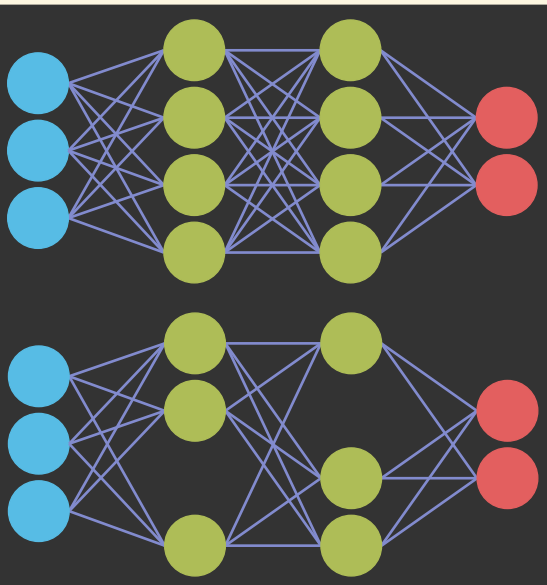
In deep learning, we generally tend to worry more about overfitting, as this tends to be an issue if models are complex, the number of training samples is small, and/or the architecture and design of the network is not optimal.

Fortunately, methods have been developed to combat overfitting.

We will mention a few commonly used methods here and explore more methods in later modules.

Dropouts

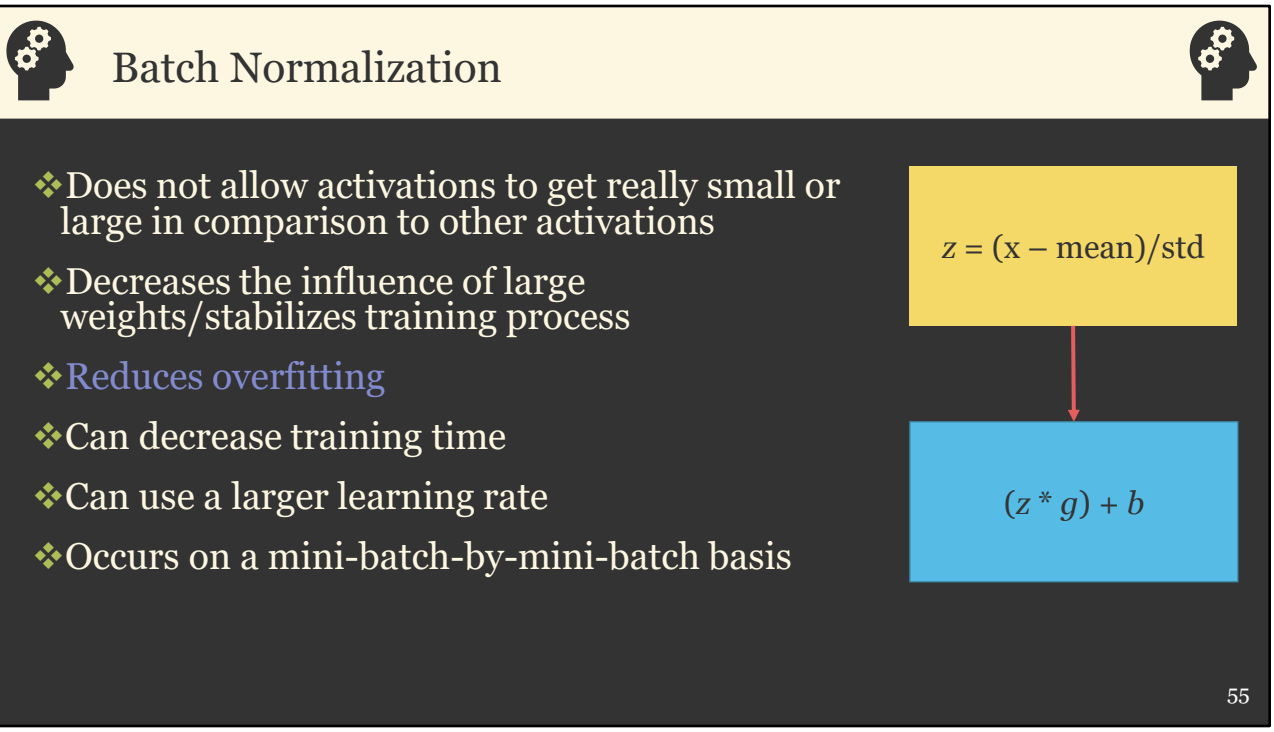
- Minimize overfitting by "dropping out" or ignoring neurons and associated parameters in the network
- Often between 0.5 and 0.8
- Can take longer for model to stabilize or converge



54

One option is to incorporate dropouts. This consists of randomly dropping out or ignoring some neurons and their associated weights/parameters in the network during some weight 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.



Batch normalization moderates the changes in activation so that they do not get too large or too small. Normalization is accomplished by subtracting the mini-batch mean and dividing by the standard deviation for the learned parameters.

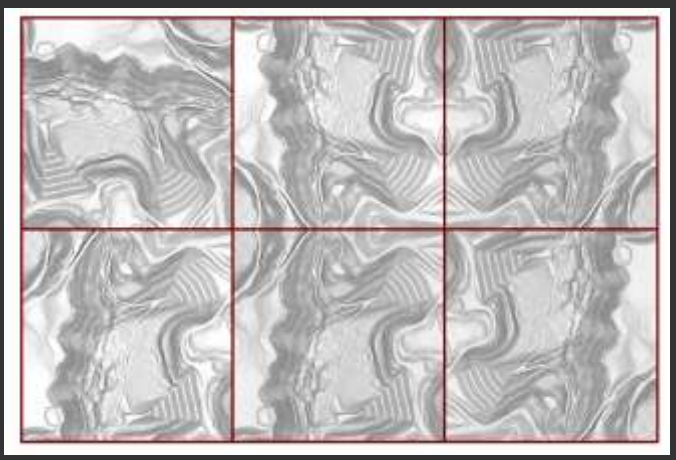
You can then apply an adjustment of the parameter values using a coefficient (g) and a bias (b). Both of these parameters are trainable.

Batch normalization has been shown to reduce the need for dropouts and alleviates the associated information loss. This normalization can reduce overfitting, decrease training time, and allow for the use of larger learning rates.

I have found batch normalization to be really useful and tend to include it in my models. We will discuss batch normalization and how to implement it within different architectures throughout the course.

Data Augmentation

- Add random noise to data
- Generate more variety and more samples
- Common when working with images



56

Another option is to augment the input data.

This is common when working with image data. The data can be augmented using flips, rotations, rescaling, cropping, blurring, sharpening, or changing the hue, saturation, and/or brightness.

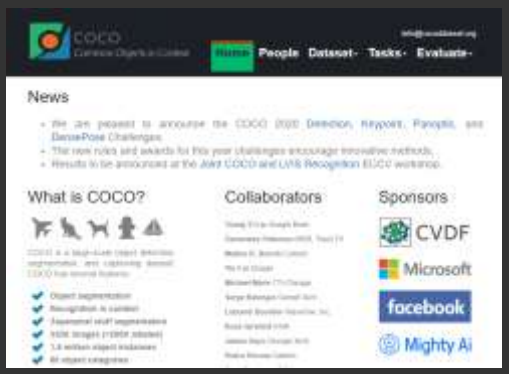
These transformations are generally applied randomly and within certain pre-defined constraints. For example, saturation may only be allowed to change by 20%.

The idea is to (1) increase the effective number of training samples and (2) increase the variety or variability within the training samples so that more generalizable patterns are learned. We will discuss and demonstrate data augmentations in later modules and code examples.

Transfer Learning

- Start with weights learned from a larger training data set
- Refine weights using new data

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



https://cocodataset.org/#home

57

If you have a small dataset, it is possible to initialize your model using pre-trained weights/parameters as opposed to random weights/parameters.

For example, large image datasets, including ImageNet and COCO, have been used to train convolutional neural networks. The learned weights/parameters can then be used to initiate a model that is then refined using new samples. This process is known as transfer learning.

The idea here is that images share common features and patterns. So, beginning with patterns learned from other data sets, even if the problem and data are different, is better than starting from random.

Transfer learning can improve model performance and decrease the training time or number of iterations or epochs required to stabilize the result. However, this may not be true in all cases.

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.