TRAINING DEEP NEURAL NETWORKS

Training a deep DNN isn’t a walk in the park. Here are some of the problems you could run into:

- **Vanishing and Exploding Gradients:** These occur when gradients become too small or too large as they propagate back through the network, making it difficult to train lower layers effectively.
- **Insufficient Training Data:** Large networks require substantial amounts of data, which may not always be available or may be expensive to label.
- **Training Speed:** Training deep networks can be time-consuming due to the computational complexity.
- **Overfitting Risk:** With millions of parameters, DNNs are prone to overfitting, especially with limited or noisy training data.

Vanishing Gradient Problem:

Occurs in deep neural networks when gradients become very small, preventing the network's weights from updating effectively. This is often due to the use of certain activation functions like sigmoid or tanh, leading to slow or stagnant learning in early layers. It makes it hard for the network to learn complex patterns.

Exploding Gradient Problem:

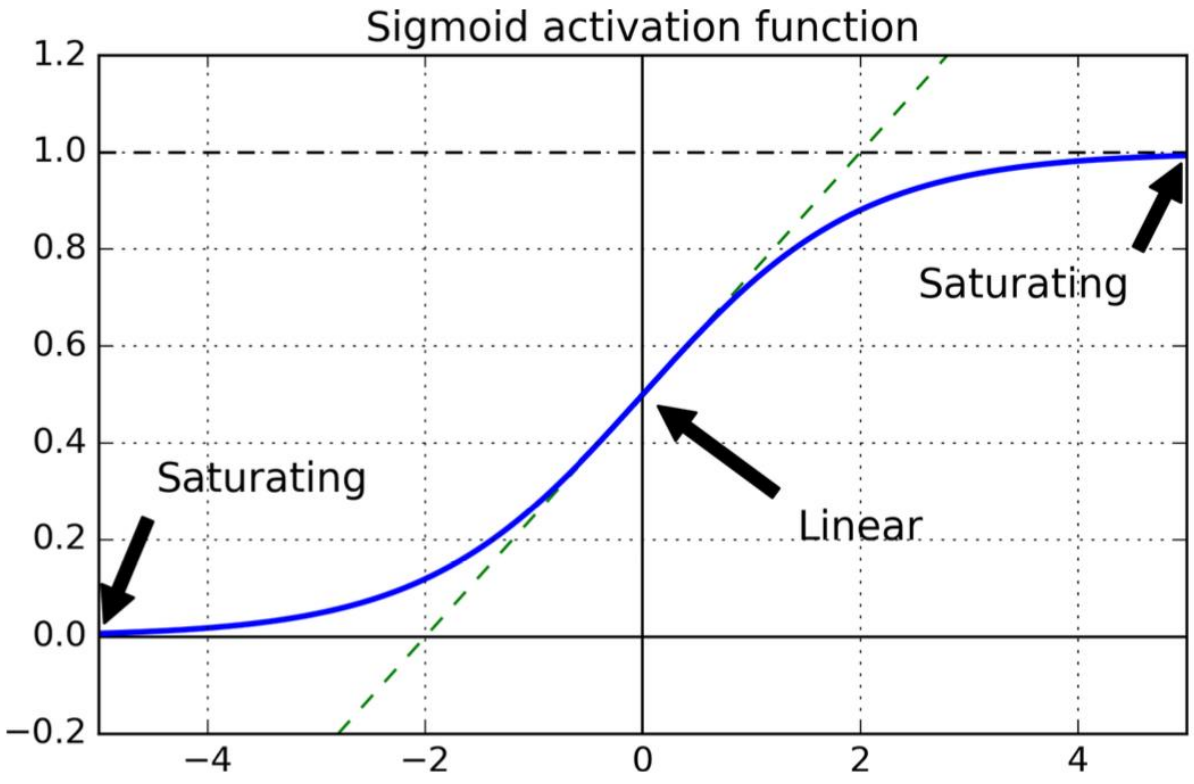
Happens when gradients grow exponentially large as we move backward through the network, causing overly large updates to the weights and leading to an unstable training process. It can result from deep network architectures, large input values, or improper weight initialization, leading to numerical instability.

Strategies for Improvement:

Activation Functions:

- **Hyperbolic Tangent (tanh):** Preferred in some contexts due to its output range (-1 to 1) and zero-centered nature, which addresses some issues associated with the sigmoid function.

- **Nonsaturating Functions:** ReLU and its variants (Leaky ReLU, ELU, SELU) are often better choices. ReLU is simple and effective but can lead to dead neurons. Leaky ReLU introduces a small, positive slope in the negative domain, preventing neurons from dying. ELU and SELU offer additional benefits like an average output closer to zero (reducing the vanishing gradient problem) and automatic normalization, respectively.



Weight Initialization:

- **Xavier/Glorot Initialization:** A strategy that aims for the variance of the outputs of each layer to match the variance of its inputs, promoting a healthy flow of gradients. It involves initializing weights randomly based on the layer's number of inputs and outputs.

Solutions to Gradient Problems:

- **For Vanishing Gradients:** Utilize ReLU or similar activation functions and include skip connections to facilitate gradient flow.
- **For Exploding Gradients:** Implement gradient clipping, proper weight initialization, and batch normalization to manage gradient magnitudes.

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

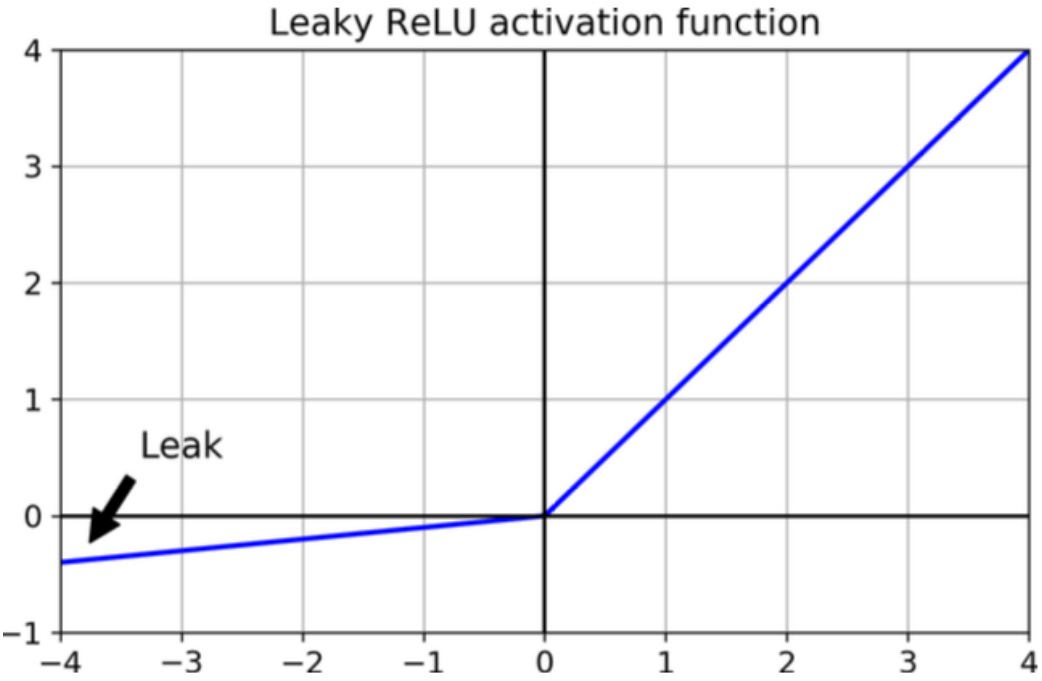
Nonsaturating Activation Functions

ReLU activation function, which does not saturate for positive values and is computationally efficient, performs better in deep networks. However, ReLU is not without its drawbacks, particularly the issue of "dying ReLUs" where neurons stop contributing to the learning process.

To address this, several ReLU variants have been introduced:

- **Leaky ReLU:** Introduces a small, positive slope for negative input values, ensuring neurons remain active.
- **Parametric ReLU (PReLU):** Allows the slope for negative inputs to be learned during training, showing significant improvement on large datasets but risking overfitting on smaller ones.

- **Exponential Linear Unit (ELU):** Outputs negative values for negative inputs, bringing the average output closer to zero and alleviating vanishing gradients. ELU has a continuous gradient which aids in faster convergence.
- **Scaled Exponential Linear Unit (SELU):** Builds on ELU by automatically scaling inputs to maintain a mean of 0 and standard deviation of 1 across layers, promoting self-normalization in networks.



These functions have their own benefits and use cases:

- **ELU** offers a balance between computational efficiency and performance by speeding up convergence without drastically increasing compute time.
- **SELU** stands out for networks where self-normalization can be achieved, often outperforming other functions in deep networks.

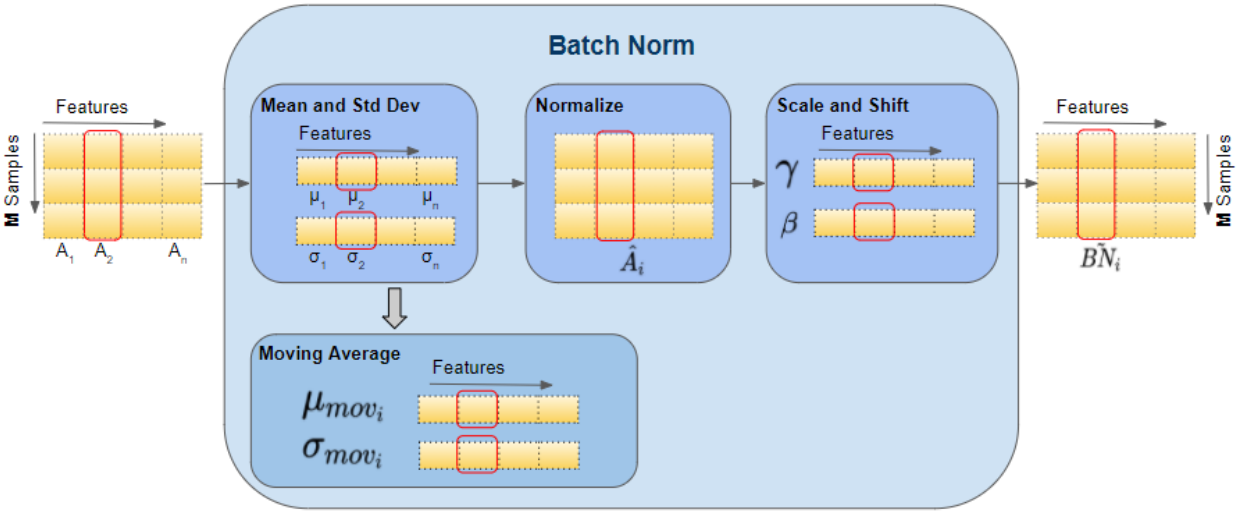
While SELU may offer the best performance for self-normalizing networks, ELU and its leaky variants provide robust alternatives for others. ReLU remains a solid default choice due to its simplicity and efficiency, particularly when computational speed is a priority.

Batch Normalization

What It Does:

- **Standardizes Inputs:** BN standardizes and normalizes the inputs of each layer within a network, making training more stable and efficient. It adjusts the inputs by zero-centering and scaling them based on the batch's statistics (mean and standard deviation).

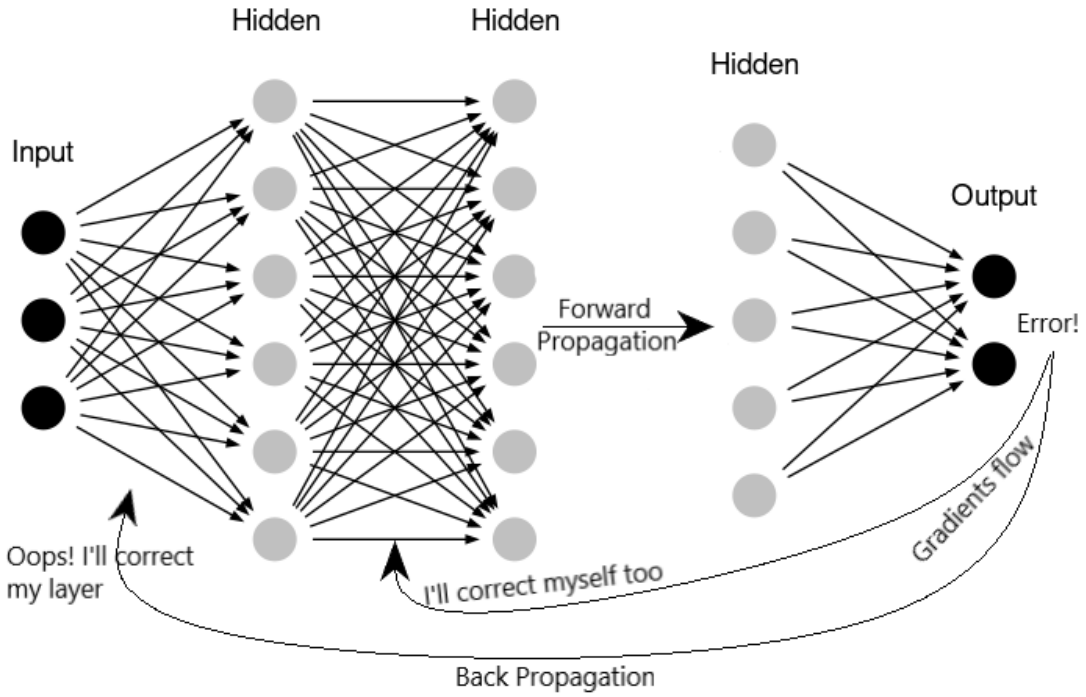
- **Two-Step Process:** First, it normalizes the inputs by subtracting the batch mean and dividing by the batch standard deviation. Then, it applies a scale and shift transformation, allowing the network to undo the normalization if needed.



How It Works:

- **During Training:** BN calculates the mean and standard deviation for each mini-batch and uses these values to normalize the inputs. It then scales and shifts the normalized inputs using parameters that the network learns through backpropagation.

- **During Testing:** BN uses the entire dataset's mean and standard deviation (estimated during training) instead of the mini-batch's to maintain consistency in predictions.



Benefits:

- **Improves Training Speed and Stability:** By normalizing the inputs, BN allows the use of higher learning rates, accelerates the learning process, and makes the network less sensitive to weight initialization.
- **Acts as Regularizer:** BN can reduce the need for other regularization techniques, potentially simplifying the model.
- **Flexibility in Activation Functions:** It enables the use of activation functions that are otherwise prone to causing vanishing or exploding gradients, like tanh or logistic sigmoid.

Implementation:

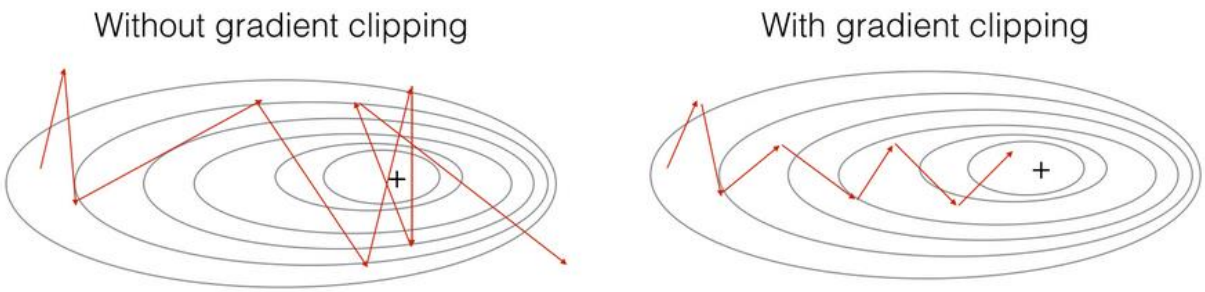
- **With Keras:** Implementing BN in Keras is straightforward. Add a **BatchNormalization** layer before or after each hidden layer's activation function. Optionally, it can also be the very first layer of the model.
- **Adjustable Parameters:** Includes scale and shift parameters learned during training, and estimated means and standard deviations for use during testing.

Considerations:

- **Adds Complexity:** While BN improves model performance and stability, it introduces additional complexity and computational overhead during training.
- **Runtime Impact:** Models with BN layers may have slightly slower prediction times due to the extra computations, though this can often be mitigated by merging the BN calculations with the preceding layer after training.

Gradient Clipping

Gradient Clipping is a technique to prevent the exploding gradients problem, particularly useful in recurrent neural networks (RNNs) where Batch Normalization might not be effective. It involves limiting the size of the gradients to a specific threshold during backpropagation, ensuring they do not become too large. There are two main methods of gradient clipping:



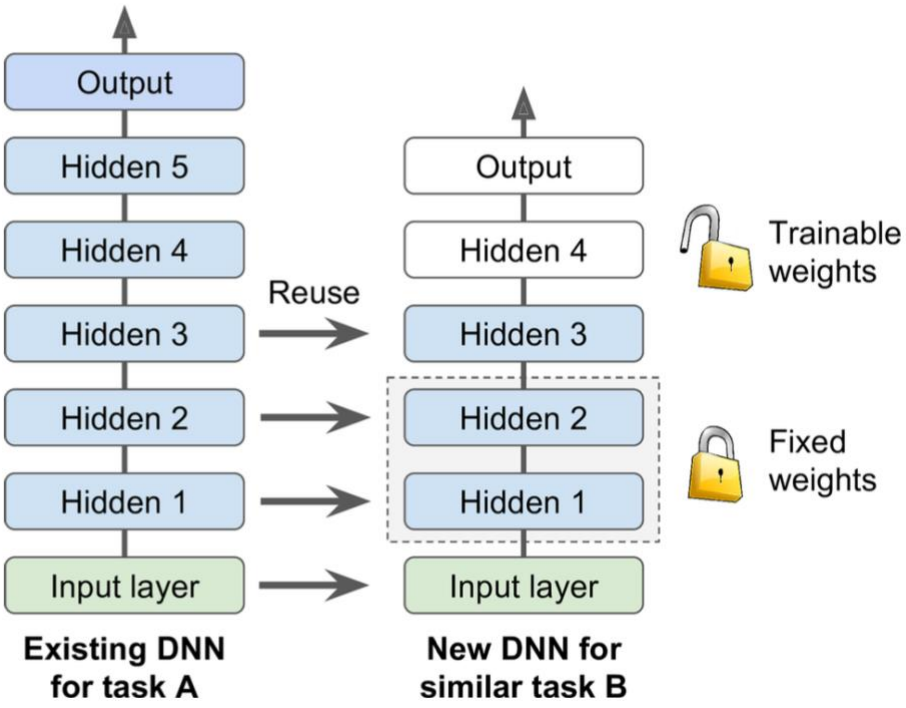
1. **Clipping by Value:** Limits each component of the gradient vector to lie within a specified range. While effective in controlling the magnitude of gradients, this approach can alter the direction of the gradient vector.
2. **Clipping by Norm:** Scales down the entire gradient vector if its norm exceeds a specified threshold, preserving the direction of the gradient vector but reducing its magnitude.

The choice between clipping by value and by norm, along with the threshold setting, is a matter of experimentation and may vary depending on the model and training data. Monitoring gradient magnitudes during training can help determine the necessity and effectiveness of Gradient Clipping.

Reusing Pretrained Layers

Reusing pretrained layers, a form of transfer learning, is an efficient approach when tackling new neural network tasks similar to ones that have already been solved. This method, recommended over training large deep neural networks (DNNs) from scratch, offers several advantages:

1. **Speeds Up Training:** Leveraging existing networks can significantly reduce training time.
2. **Requires Less Data:** Transfer learning is particularly useful when training data is limited.



The process involves repurposing the lower layers of an existing network that was trained on a related task. Since these layers have learned to recognize low-level features that are likely similar across both tasks, they can serve as a solid foundation for the new task. However, adjustments are often necessary:

- **Input Size:** If the new task's input images differ in size from those in the original model, preprocessing steps to resize them may be needed.
- **Output Layer:** This layer should typically be replaced to fit the new task's specific output requirements.

- **Hidden Layers:** While lower layers can often be reused directly, the higher layers, which detect more task-specific features, may need adjustments. The strategy here includes:

    - **Layer Reuse:** The similarity of the tasks dictates how many layers to reuse. For closely related tasks, it might be beneficial to keep all hidden layers, replacing only the output layer.
    - **Freezing Layers:** Initially, freezing the reused layers (making their weights non-trainable) can prevent them from losing their pre-trained features during the initial phase of training.
    - **Unfreezing and Fine-tuning:** Depending on performance and the availability of training data, gradually unfreezing and fine-tuning some of the top layers may improve results.
    - **Adjusting Layers:** If performance is still not satisfactory, especially with limited data, removing the top hidden layers or adjusting the architecture further might help. Conversely, with ample data, it could be beneficial to add new layers.

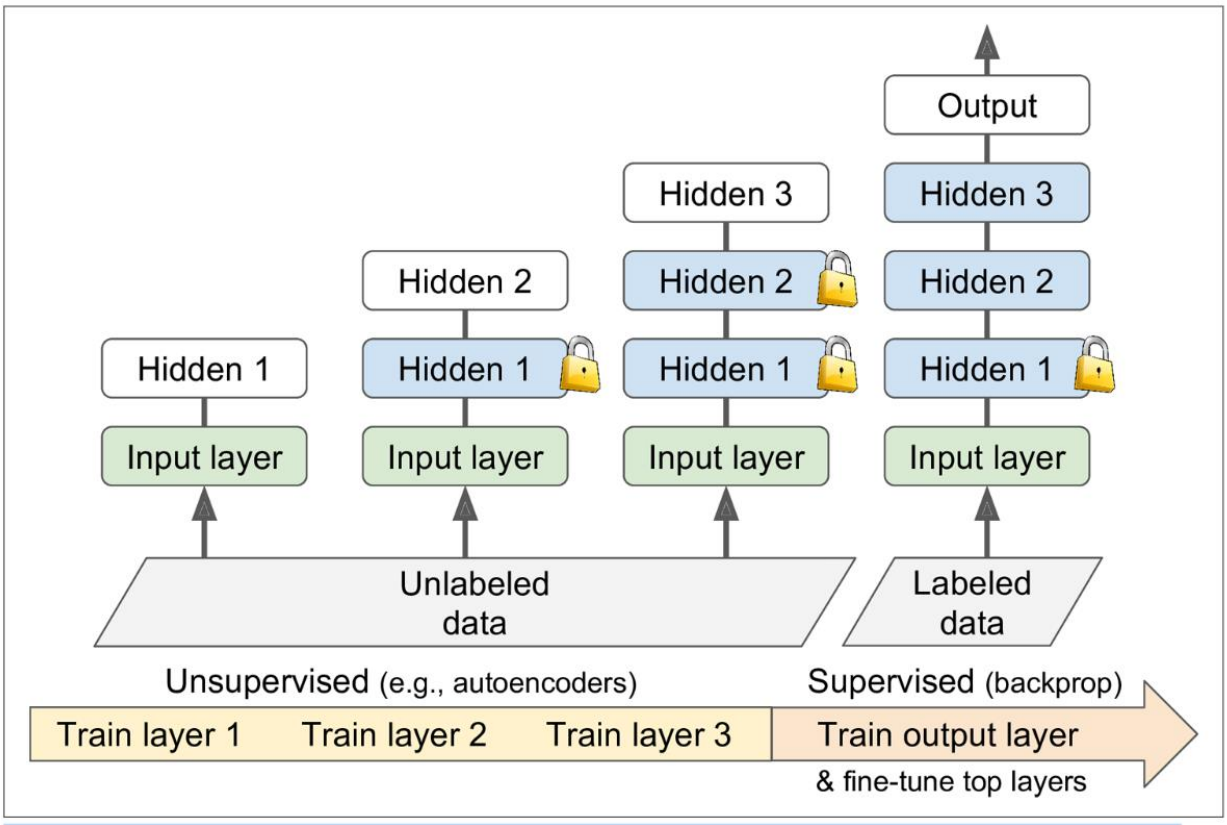
Creating a neural network for dog breed classification, transfer learning is applied by using a pretrained model originally designed for general image classification:

1. **Preprocessing:** The input images, initially 500x500 pixels, are resized to 224x224 pixels to match the pretrained model's specifications.
2. **Output Layer:** The model's original output layer, designed to classify 1,000 categories, is replaced with a new layer capable of classifying 50 dog breeds.
3. **Hidden Layers:**
    - Most lower layers are reused as they contain general features applicable to the task.
    - These layers are initially frozen to keep their learned features intact.
    - To refine the model, top hidden layers are later unfrozen for fine-tuning, allowing the model to better distinguish between similar dog breeds.
    - Further adjustments include adding new layers to capture more complex patterns specific to dog breeds, based on the available data.

Unsupervised Pretraining

Unsupervised pretraining is a technique useful for tackling complex tasks where there's a shortage of labeled training data and no pre-existing model for a similar task is available. The key idea is to leverage a large amount of easily obtainable unlabeled data to train a model in an unsupervised manner—meaning the model tries to learn patterns from the data without any guidance on what it's looking for.

This approach typically uses autoencoders or generative adversarial networks (GANs), which learn to encode or generate data, respectively.
After training on the unlabeled data, the next step involves reusing the learned features (specifically, the lower layers) from this unsupervised model by adding a new output layer tailored to the specific task at hand. This modified model is then fine-tuned with the available labeled data through supervised learning, where the model is now given specific targets to aim for.



Some simplifications:

Autoencoders are like skilled artists who capture the essence of a portrait by focusing on crucial features for reconstruction, aiming to compress data into a simpler form and then accurately rebuild it, shedding light on important data characteristics.

GANs (Generative Adversarial Networks) involve a creative battle between a data-generating forger and a critical detective. The forger strives to create data so convincingly real that the detective can't distinguish it from actual data, leading to the generation of highly realistic outputs.

RBMs (Restricted Boltzmann Machines) are early tools that facilitate understanding of data by representing visible data points and their hidden, significant features through a network where only cross-group interactions are allowed. They paved the way for more advanced methods like autoencoders and GANs, which are now preferred for their effectiveness in a variety of applications.

Pretraining on an Auxiliary Task

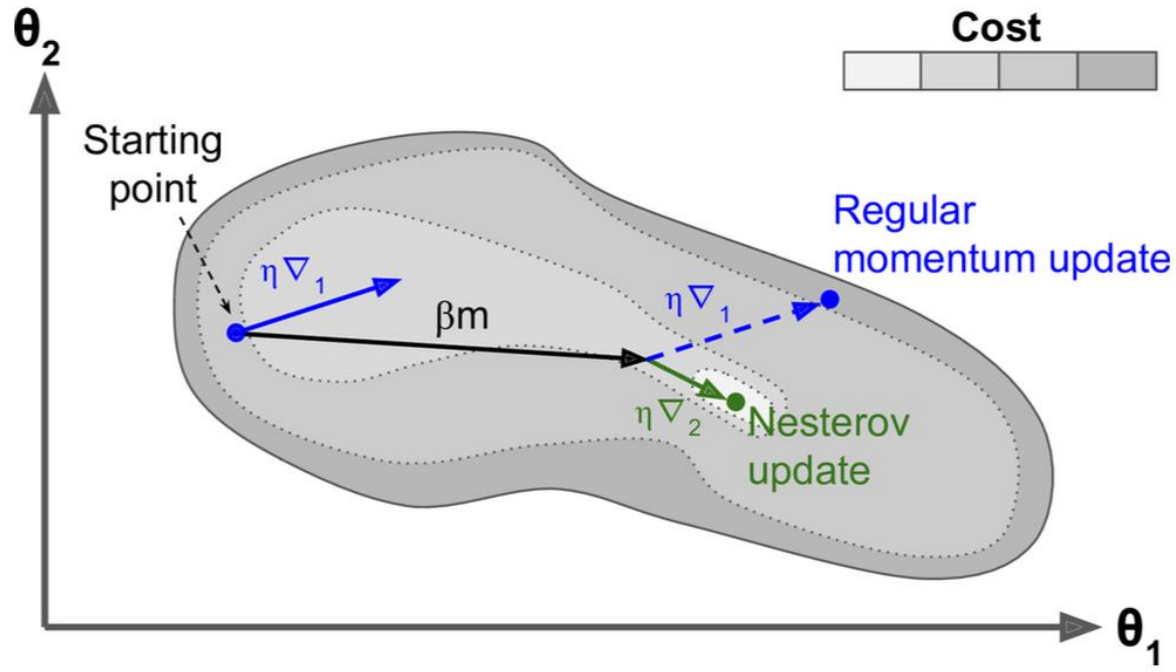
Pretraining on an auxiliary task helps when you don't have enough labeled data for your main project. Essentially, you first train a neural network on a related task with plenty of data. This network learns useful features that can be reused for your actual task, improving its performance despite limited data.

For example, if you're working on facial recognition with few images per person, you could start by training a network to identify if two photos are of the same person using many web images. This teaches the network to recognize key facial features, which can then help in your specific facial recognition task.

In language tasks, you could use a large text collection to teach a model to fill in missing words, thus learning about language structure. This knowledge can then be applied to more specific language processing tasks with fewer data requirements.
This process, called self-supervised learning, creates its own labels from the data, making it a cost-effective method that falls under unsupervised learning due to its lack of need for manually labeled data.

To train large networks faster, beyond methods like better initial setup or using pretrained parts, using advanced optimizers like momentum optimization or Adam can make the training quicker and more effective.

Nesterov Accelerated Gradient

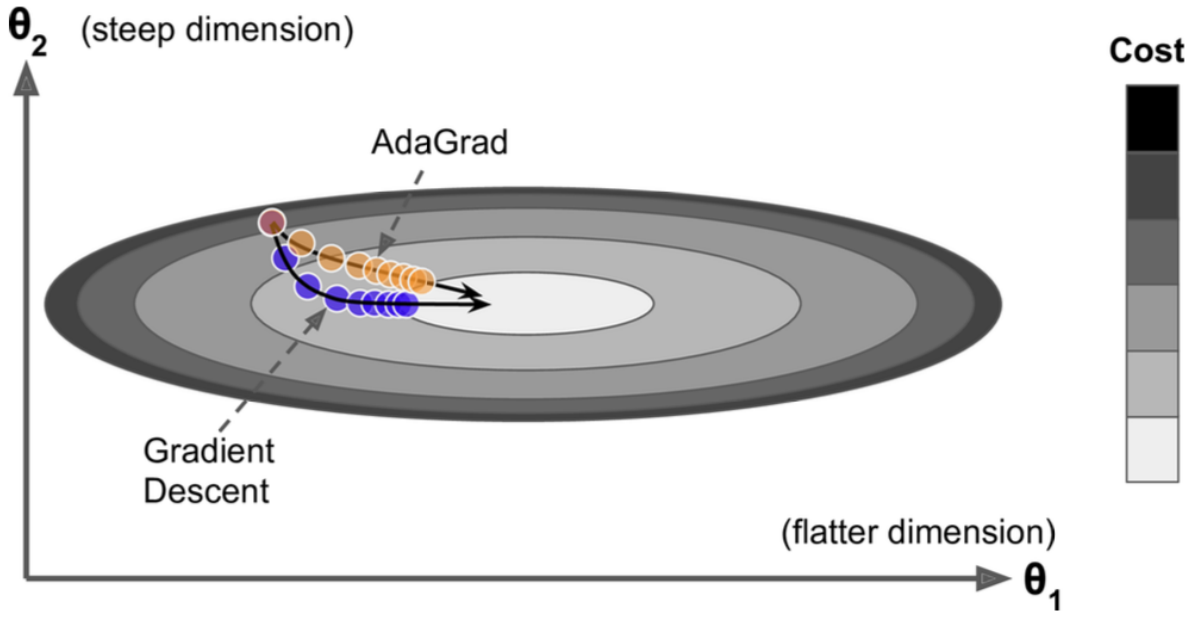


The Nesterov Accelerated Gradient (NAG), a tweak on momentum optimization proposed by Yurii Nesterov in 1983, speeds up the learning process by looking ahead in the momentum's direction when calculating gradients. Instead of calculating the gradient at the current position, NAG estimates it slightly ahead, where the algorithm is likely to be after applying momentum. This foresight allows for more accurate updates, steering the optimization more directly towards the optimum and reducing oscillations, making NAG generally faster and more efficient than traditional momentum optimization. To implement NAG in Keras, simply enable the `nesterov` parameter in the SGD optimizer.

AdaGrad

AdaGrad helps direct the learning process more accurately towards the optimum by adjusting the learning rate differently across all dimensions of the data. It does this by scaling down gradients more in steeper dimensions, effectively giving an adaptive learning rate that makes

it easier to navigate through complex cost functions. While AdaGrad simplifies the need for hyperparameter tuning and shows promise in simple quadratic problems, its aggressive downscaling of the learning rate can lead to premature stopping in deep neural network training, making it less suitable for these applications.



RMSProp

RMSProp addresses AdaGrad's tendency to slow down too much by only considering recent gradients for the learning rate adjustment, using a technique called exponential decay. This approach prevents the learning rate from diminishing too rapidly, maintaining a balance that encourages consistent progress towards the global optimum. With a default decay rate that often requires no tuning, RMSProp has shown to be effective across a wide range of problems, surpassing AdaGrad, especially in neural network training, until the introduction of the Adam optimizer, which combines the benefits of RMSProp and other optimizations for even better performance.

Adam and Nadam Optimization

Adam, short for adaptive moment estimation, merges momentum optimization's idea of tracking past gradients and RMSProp's concept of adjusting gradients based on their recent magnitude. It computes decaying averages of both past gradients and their squares, adjusting updates to be more efficient. This optimizer is especially user-friendly as it requires less tuning of the learning rate, often working well with the default setting. Adam adjusts learning rates adaptively, improving upon both AdaGrad and RMSProp by offering a balanced approach that generally leads to fast and stable convergence.

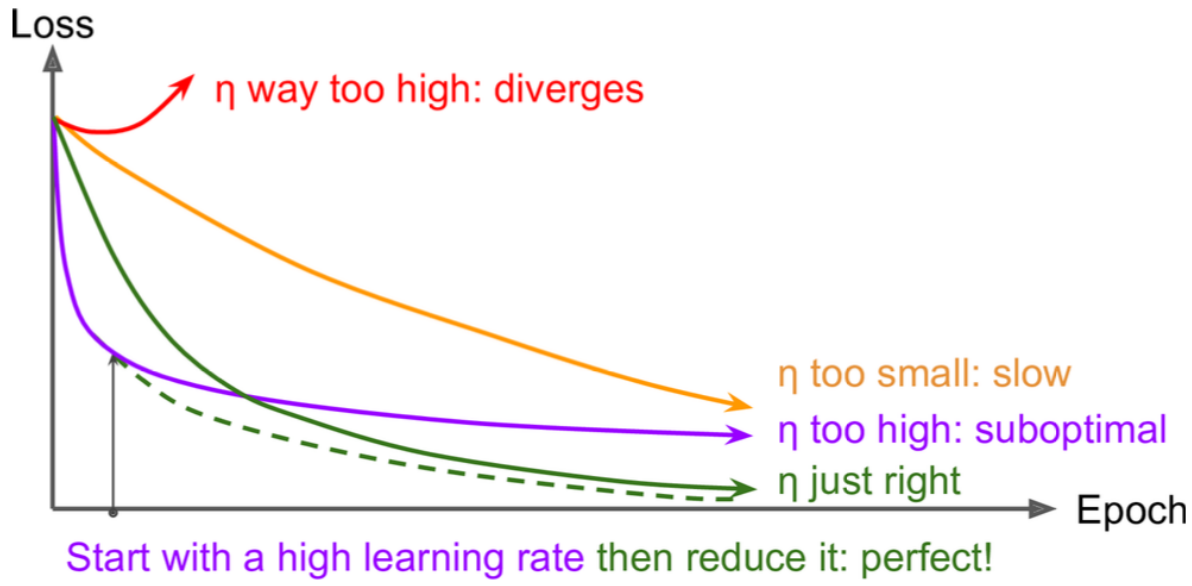
Adam has two notable variants: AdaMax, which uses the max of gradients instead of their square roots for updates, potentially offering more stability but generally underperforming compared to Adam; and Nadam, which incorporates the Nesterov trick into Adam, usually resulting in slightly faster convergence.

While adaptive optimization methods like Adam, RMSProp, and Nadam are often effective, they may not always lead to the best generalization on some datasets. In such cases, simpler methods like Nesterov Accelerated Gradient might be more effective. It's also worth noting that all these techniques primarily rely on first-order derivatives due to the impracticality of calculating second-order derivatives in deep neural networks, which would require extensive memory and computational resources.

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`Nadam`	***	** or ***
`AdaMax`	***	** or ***

Learning Rate Scheduling

Learning Rate Scheduling involves adjusting the learning rate as training progresses to improve convergence speed and model performance. Choosing the right initial learning rate is crucial: too high can lead to divergence, too low can slow down convergence, and slightly high rates may cause the model to oscillate around the optimum without settling.



Several strategies for learning rate scheduling include:

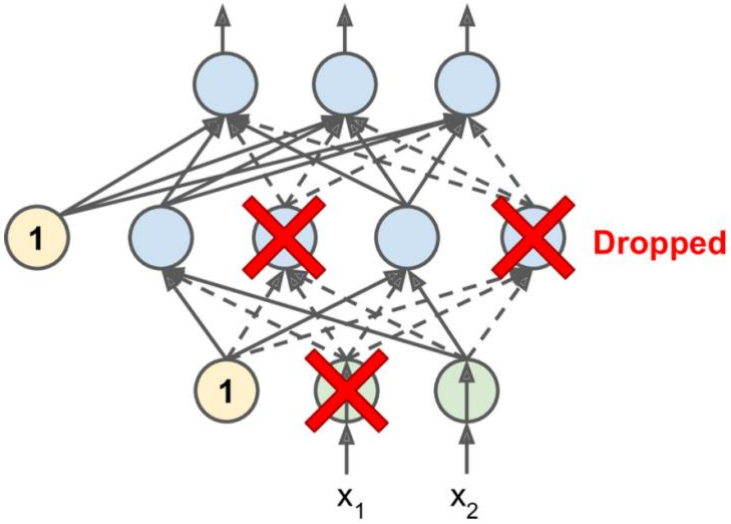
- **Power scheduling:** Gradually decreases the learning rate following a pre-defined mathematical formula, slowing down the rate reduction over time.
- **Exponential scheduling:** Reduces the learning rate by a constant factor every few steps, maintaining a steady pace of reduction.
- **Piecewise constant scheduling:** Uses fixed learning rates for set numbers of epochs, requiring manual adjustments to find the optimal rates and durations.
- **Performance scheduling:** Lowers the learning rate in response to the stagnation of model improvement, similar to early stopping but for learning rate adjustment.
- **1cycle scheduling:** Starts with increasing the learning rate for the first half of training, then decreases it for the second half, potentially speeding up training and improving performance.

Regularization techniques are essential for preventing overfitting in neural networks, which can happen due to their large number of parameters. Key strategies include:

- **Early stopping:** Halts training when the model's performance on a validation set stops improving.

- **L1 and L2 regularization:** Adds penalties on layer weights to encourage smaller weights, helping to simplify the model. L1 regularization leads to sparse models with more weights close to zero.

- **Dropout:** Randomly "drops out" a subset of neurons in each layer during training, forcing the network to learn more robust features. This can be viewed as training a large ensemble of networks with shared weights, improving generalization.



- **Max-norm regularization:** Restricts the size of the weight vector for each neuron to a maximum value, combating the unstable gradients problem and further reducing overfitting.

Dropout is notably effective, offering a simple yet powerful method to improve model robustness and performance. It involves randomly ignoring a portion of neurons during each training step, making the network less reliant on any specific set of neurons.

Post-training, dropout isn't applied, but the weights are adjusted to compensate for the dropout during training.

Monte Carlo (MC) Dropout is an extension that provides better performance and uncertainty estimates by averaging predictions over multiple forward passes with dropout enabled, reflecting a more robust consensus of the network's predictions.

Max-norm regularization controls the maximum norm of the incoming weights to each neuron, offering another lever to prevent overfitting by constraining the model's complexity. Implementing these regularization techniques in Keras varies in complexity, from simple attribute adjustments to custom layer and callback definitions. Each technique can be tailored to the specific needs of the model and the task, balancing model complexity with the capacity to generalize well to unseen data.

Summary and Practical Guidelines

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Additional recommendations include:

- Normalize input features.
- Reuse parts of pretrained networks if applicable.
- Consider unsupervised pretraining or pretraining on an auxiliary task with abundant unlabeled or similar labeled data, respectively.

Exceptions and further adjustments might be needed for:

- **Sparse models:** Use L1 regularization and possibly the TensorFlow Model Optimization Toolkit for even sparser models.
- **Low-latency models:** Optimize for speed with fewer layers, integrate Batch Normalization, opt for faster activation functions like leaky ReLU or ReLU, and consider reducing float precision.
- **Risk-sensitive applications or when latency is less critical:** Utilize MC Dropout to improve performance and provide reliable probability estimates along with uncertainty estimates.