Training Deep NeuralNetworks

Problems when Training large Deep Nets

1. **Vanishing or Exploding Gradients**: Gradients can either become too small, making training slow, or too large, causing instability during updates. This is common in deep networks with many layers.
2. **Insufficient or Unlabeled Data**: Deep nets require large datasets for effective learning; without enough data, models struggle to generalize. Lack of labeled data further complicates supervised training.
3. **Slow Training**: Training deep networks requires significant computational resources and time, especially with large datasets and complex models. Optimizing this process is a major challenge.
4. **Overfitting with Large Models**: Large models with millions of parameters risk overfitting, especially if the data is noisy or insufficient. Regularization techniques are necessary to prevent this.

Vanishing Gradient Problem

The vanishing gradient problem occurs during the training of deep neural networks when gradients—used to update model parameters via backpropagation—become very small as they propagate backward through the layers. This causes the weights of the earlier layers to update very little, effectively preventing the model from learning efficiently. More generally deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

Few suspects including the combination of the popular sigmoid function, and the weight initialization technique.(mean 0 and standard deviation 1). It shows that with the sigmoid function and initialization scheme the variance of the outputs of each layer is much greater than the variance of the inputs. Going forward in the network the variance keeps increasing after each layer until the activation function saturates at the top layers.

when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0 (i.e., the curve is flat at both extremes). Thus, when backpropagation kicks in it has virtually no gradient to propagate back through the network.

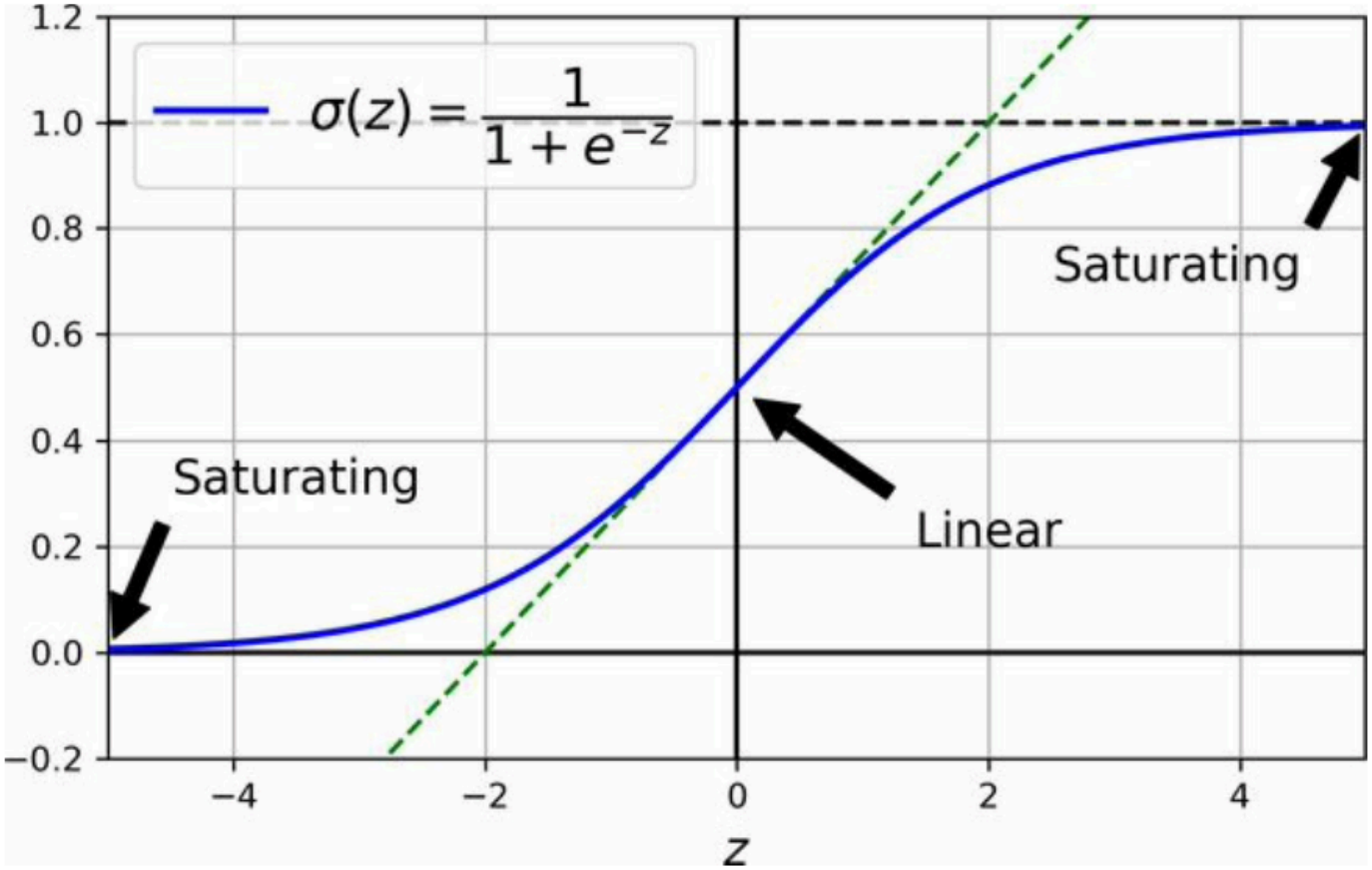


Figure 11-1. Sigmoid activation function saturation

Glorot and He initialization

We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when back propagating gradients. We don’t want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs. the gradients to have equal variance before and after flowing through a layer in the reverse direction.
It is actually not possible to guarantee both unless the layer has an equal number of inputs and outputs.

Example : if you set a microphone amplifier knob too close to zero, people won't hear you voice , but if you set it too close to max, you voice will be saturated and people won't understand what you are saying. Now imagine a chain of amplifiers: they all need to be set properly in order for your voice to come out loud and clear at the end of the chain. Voice must come out at the same amplitude it came in.

The connection weights of each layer must be initialized randomly as described in
fan = (fan + fan avg in out) / 2. This initialization strategy is called Xavier initialization or Glorot initialization.

If you replace fan with fan avg in you get an initialization strategy that Yann LeCun proposed in the 1990s. He called it LeCun initialization.

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, sigmoid, softmax	$1 / fan_{\text{avg}}$
He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2 / fan_{\text{in}}$
LeCun	SELU	$1 / fan_{\text{in}}$

Better Activation functions

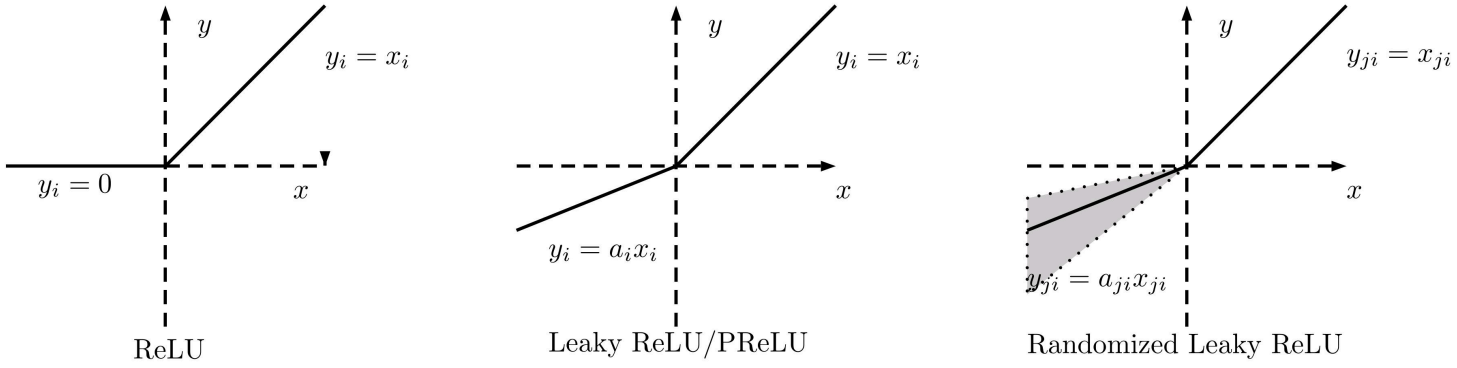
Activation functions are use to introduce non-linearity to the neuron.
The Relu is mostly is used , it does not saturate for positive values and also is fast to compute.

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the dying ReLUs: during training, some neurons effectively “die” , meaning they stop outputting anything other than 0.
In some cases, you may find that half of your network’s neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the input of the ReLU function (i.e., the weighted sum of the neuron’s inputs plus its bias term) is negative for all instances in the training set.

To solve this problem a variant of Relu is used called Leaky Relu.

Leaky Relu

A dead neuron may come back to life it its input evolve over time and eventually return within a range where the Relu activation function gets a positive input again.



The leaky ReLU activation function is defined as LeakyReLU (z) = max(αz,z). The hyperparameter α defines how much the function “leaks”: it is the slope of the function for z < 0. Having a slope for z < 0 ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up.

Leaky Relu always outperformed the strict Relu.

Three variations of the Leaky ReLU activation function:
the randomized leaky ReLU (RReLU),
the parametric leaky ReLU (PReLU),
and the standard Leaky ReLU.

RReLU, where the slope α is randomly chosen during training and fixed during testing, acted as a regularizer and helped reduce overfitting. PReLU, where α is learned during training instead of being fixed, outperformed ReLU on large datasets but was prone to overfitting on smaller ones. Overall, these variations showed potential improvements over the standard ReLU, especially in terms of regularization and adaptability.

ReLU, leaky ReLU, and PReLU all have discontinuities in their derivatives at z=0z = 0z=0, meaning their gradients change abruptly. This lack of smoothness can cause issues during gradient descent optimization, leading to oscillations around the optimal solution and slower convergence. Such discontinuities make the training process less stable and efficient, particularly when fine-tuning model parameters or using optimization algorithms that rely on smooth gradients for faster convergence.

ELU

The ELU function is defined as:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0 \end{cases}$$

Where:

- α is a hyperparameter that controls the value for negative inputs (typically set to 1).
- For positive values of x, the function behaves like a regular identity function (i.e., $f(x) = x$).
- For negative values of x, the function has an exponential decay, making it smoother than the ReLU function and preventing the zero-gradient issue for negative inputs.

- It takes on negative values when z < 0, which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter α defines the opposite of the value that the ELU function approaches when z is a large negative number.
- It has a nonzero gradient for z < 0, which avoids the dead neurons problem.
- If α is equal to 1 then the function is smooth everywhere, including around z = 0, which helps speed up gradient descent since it does not bounce as much to the left and right of z = 0.

The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function). Its faster convergence rate during training may compensate for that slow computation, but still, at test time an ELU network will be a bit slower than a ReLU network.

Selu

SELU (Scaled Exponential Linear Unit) is a variant of the **ELU** (Exponential Linear Unit) activation function designed to help neural networks maintain stable activations and gradients during training, specifically in deep networks.

SELU is particularly notable for its **self-normalizing** property, meaning that it helps to keep the mean and variance of the activations close to 0 and 1, respectively, throughout the network.

If you build a neural network consisting solely of dense layers (MLP) with all hidden layers using the **SELU (Scaled Exponential Linear Unit)** activation function, the network will **self-normalize** during training. This means that the output of each layer will tend to maintain a mean of 0 and a standard deviation of 1, helping to solve issues like **vanishing** or **exploding gradients**.

As a result, SELU can often outperform other activation functions, especially in **deep MLPs**, by ensuring stable gradient flow, which leads to more effective learning and faster convergence. Additionally, SELU eliminates the need for batch normalization, simplifying the network while still maintaining training stability.

Gelu Swish Mish

GELU introduces a smooth non-linearity and allows for small negative values in its output, unlike ReLU which completely zeroes out negative inputs. It has become popular in modern deep learning, especially in transformer models like BERT, due to its smooth nature, which helps in improving convergence and enabling better gradient flow.

It is the smooth variant of the Relu activation function.
It is computationally expensive and the performance boost does not always justify the sufficient cost.

Swish is a self-gated activation function where the input is multiplied by a sigmoid function of the input itself. Unlike ReLU, which can cause dead neurons for negative values, Swish allows smooth, non-zero gradients for all inputs.

Mish is designed to improve upon both ReLU and Swish by providing smoother gradients and enabling better flow of information throughout the network. The function allows for negative values, but unlike ReLU, it prevents dead neurons by ensuring that gradients do not vanish for negative inputs.

For normal use case Relu is good , for complex case swish works best , mish can give better result but is expensive.

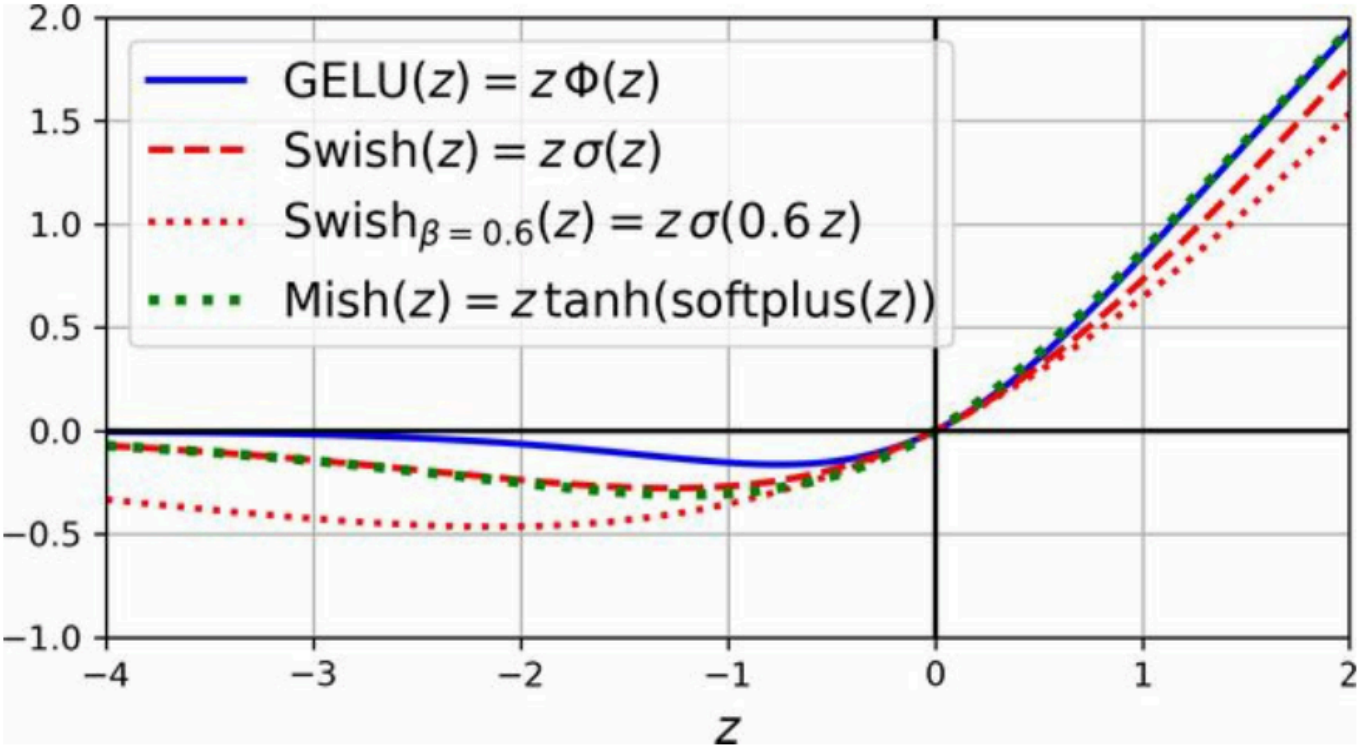


Figure 11-4. GELU, Swish, parametrized Swish, and Mish activation functions

Batch Normlaization

The technique consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero- centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs.

In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set. That is, there's no need for StandardScaler or Normalization; the BN layer will do it.

Batch Normalization (BatchNorm) is a technique used to improve the training of deep neural networks by normalizing the activations of each layer during training. It helps address several common issues, such as internal covariate shift, vanishing/exploding gradients, and allows for faster training with higher learning rates.

Drawbacks:

1. **Increased Computational Cost**: BatchNorm introduces additional computations during both training and inference, such as calculating mean and variance for each mini-batch. This adds some overhead to both the training and prediction time.
2. **Batch Size Sensitivity**: The effectiveness of BatchNorm can be impacted by the batch size. If the batch size is too small, the estimates of mean and variance may not be stable, leading to suboptimal normalization.
3. **Inference Overhead**: During inference, BatchNorm requires storing the moving averages of mean and variance, and each layer still has to perform normalization using these statistics, which can be slower than a straightforward forward pass in networks without BatchNorm.

Gradient Clipping

Gradient clipping is a technique used to prevent the **exploding gradients** problem, which occurs when gradients during backpropagation become excessively large, causing the model's weights to update too dramatically and destabilizing the training process. The idea behind gradient clipping is simple: during backpropagation, if the gradient of a parameter exceeds a pre-defined threshold, it is scaled down to ensure it remains within a manageable range.

How Gradient Clipping Works:

1. **Set a Threshold**: A predefined scalar threshold (e.g., clip_value\text{clip_value}clip_value) is set, which determines the maximum allowed value for the gradient.
2. **Calculate the Gradient**: After calculating the gradients during backpropagation, check the norm of the gradient vector (usually the L2 norm).
3. **Clip the Gradient**: If the norm of the gradient exceeds the threshold, scale the gradient vector so that its norm equals the threshold. This can be done by multiplying the gradients by the ratio of the threshold to the current norm.

This optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0

Reusing Pretrained Layers

Reusing Pretrained Layers is a key technique in **transfer learning**, where you leverage an existing neural network that has already been trained on a similar task. Rather than training a large deep neural network (DNN) from scratch, which is computationally expensive and requires a massive amount of data, you can reuse most of the pretrained layers of a network.

Typically, you will fine-tune only the **top layers** (the last few layers), which are more specific to the task at hand. This allows you to transfer knowledge from one domain to another, accelerating the training process and reducing the need for large amounts of labeled data.

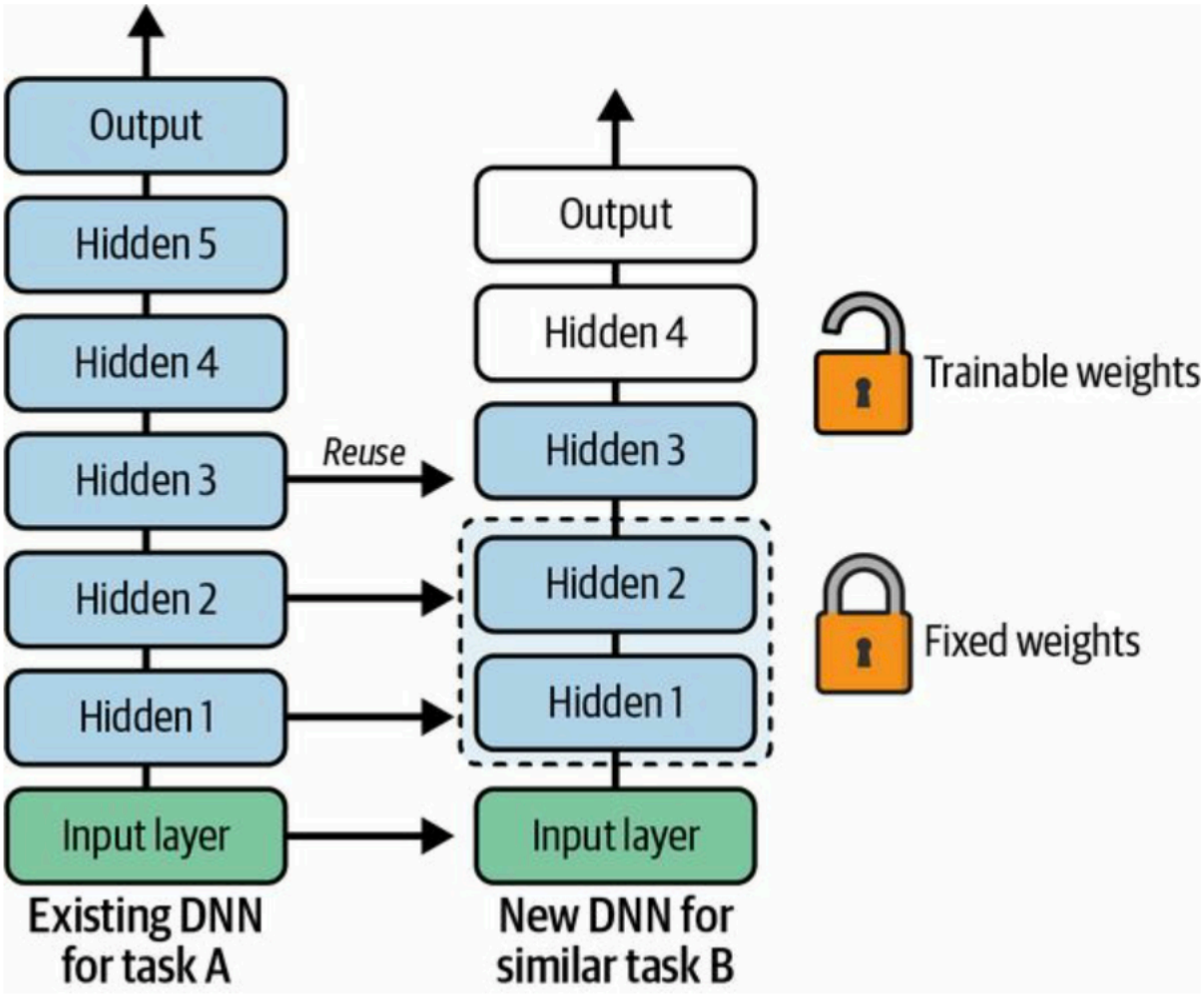


Figure 11-5. Reusing pretrained layers

The output layer of the original model should usually be replaced because it is most likely not useful at all for the new task, and probably will not have the right number of outputs.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task

Try freezing all the reused layers first , then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves.

Unsupervised Pretraining

Unsupervised Pretraining is a valuable technique when you have a complex task to solve but limited labeled training data, and no similar pretrained models are available. Instead of starting from scratch with supervised learning, you can gather a large amount of **unlabeled** data, which is often cheaper and easier to obtain. Using this data, you can train an **unsupervised model** like an **autoencoder** or a **generative adversarial network (GAN)**. These models learn to capture the underlying structure and features of the data without requiring labels.

Once the unsupervised model is trained, you can reuse the **lower layers** of the model (which capture general features of the data) and add a new output layer tailored to your specific task. This output layer can then be trained using **supervised learning** with the limited labeled data you have. This process of combining unsupervised pretraining with supervised fine-tuning allows you to leverage large amounts of unlabeled data, making it possible to solve tasks with limited labeled data.

Although unsupervised pretraining has evolved, and methods like **autoencoders** and **GANs** are now commonly used, the fundamental idea remains the same: **learn useful representations of the data in an unsupervised manner** before applying supervised learning on top for the specific task. The older **greedy layer-wise pretraining** method, which involved training each layer of the network sequentially and freezing it before adding new layers, has been largely replaced by more efficient approaches that train the entire model in one go. Nonetheless, unsupervised pretraining remains a powerful tool when labeled data is scarce, and it can significantly enhance the performance of deep neural networks.

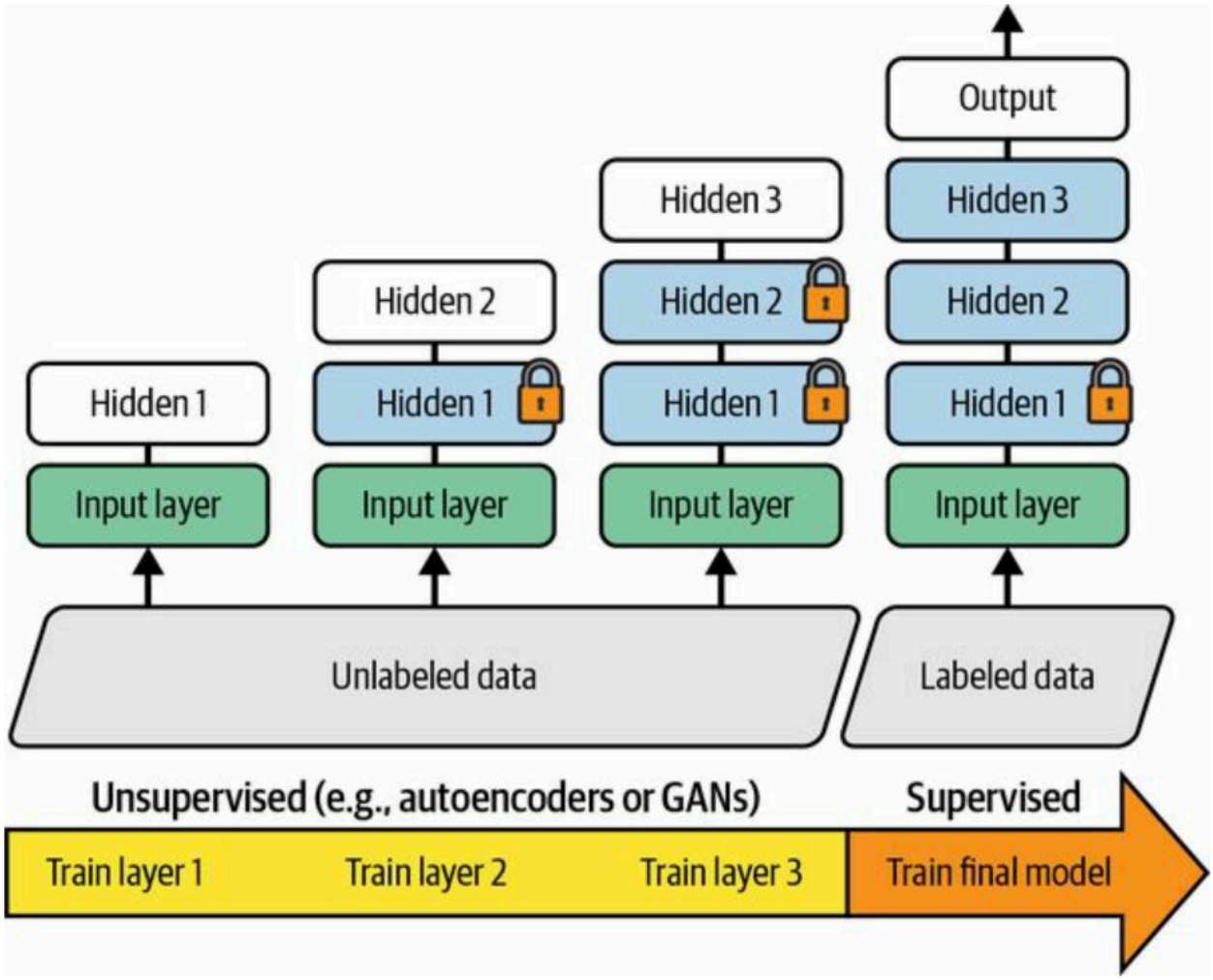


Figure 11-6. In unsupervised training, a model is trained on all data, including the unlabeled data, using an unsupervised learning technique, then it is fine-tuned for the final task on just the labeled data using a supervised learning technique; the unsupervised part may train one layer at a time as shown here, or it may train the full model directly

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

Fast Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using batch normalization, and reusing parts of a pretrained network. Another huge speed boost comes from using a faster optimizer than the regular gradient descent optimizer.

Momentum

Momentum in optimization is like giving your gradient descent a **push to move faster** by remembering the direction it was going in the past.

Imagine you're rolling a ball down a hill:

- If the ball keeps moving in the same direction, it builds up speed (momentum).
- Even if the hill flattens a bit, the ball will still roll forward because of its past speed.

In **momentum-based gradient descent**, the idea is similar. Instead of just looking at the current slope (gradient), it also considers the direction it was heading before. This helps in:

1. **Speeding up** the descent in consistent directions (like a gentle slope).
2. **Smoothing out oscillations** in directions that keep changing.

How Momentum Works:

1. The optimizer maintains a "velocity" vector that keeps track of the previous updates.
2. The current gradient update is combined with this velocity, so the model moves faster in the right direction.

The gradient is used as an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, called the momentum.

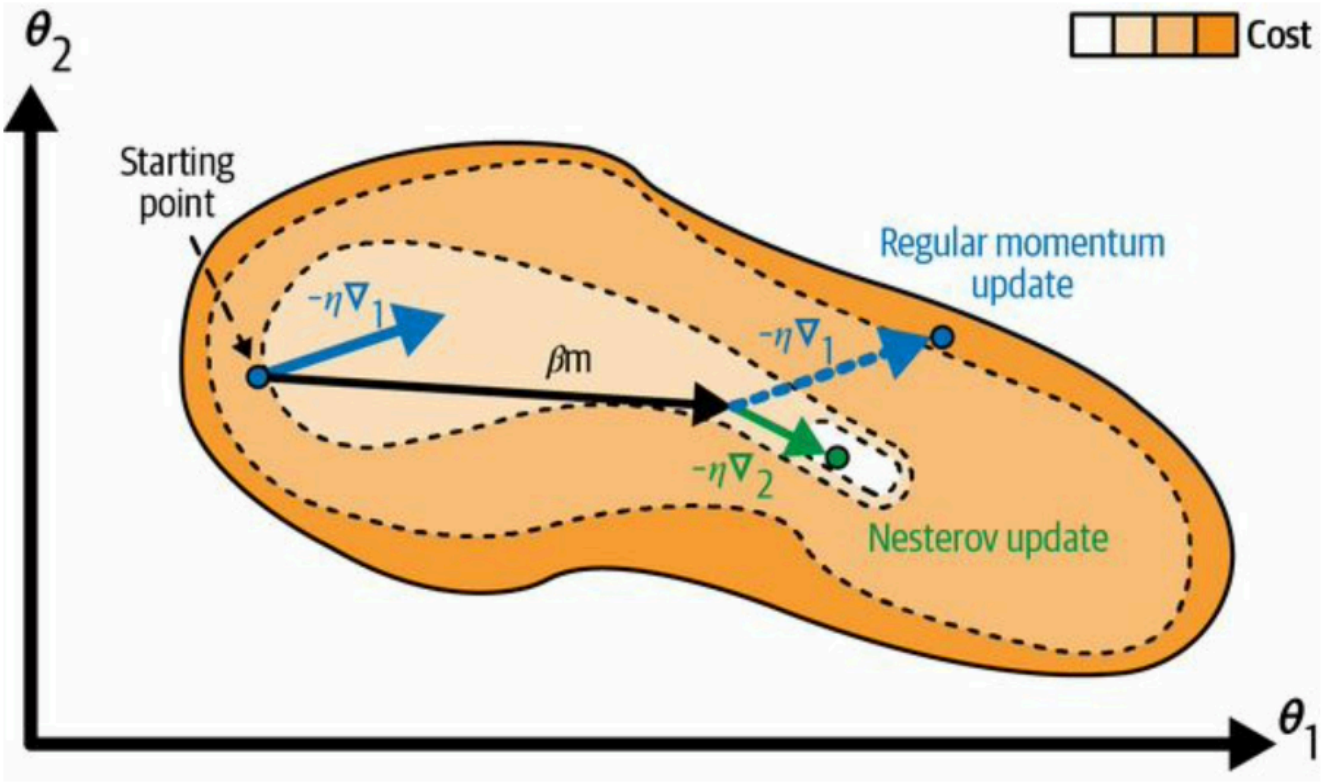
Drawback is that it adds another hyperparameter to tune, but goes faster than gradient descent.

Nesterov Accelerated Gradient

Nesterov Accelerated Gradient (NAG) is an optimization technique that improves **gradient descent** by adding momentum and "looking ahead." Unlike standard momentum, which updates the parameters based on the current gradient, NAG first makes a small step in the direction of the momentum, then computes the gradient at this new position.

This "look-ahead" helps to better estimate the gradient, leading to faster and more accurate updates.

NAG accelerates convergence by reducing oscillations and helping the optimizer move more smoothly towards the minimum. It is particularly useful in deep learning, where it helps speed up training and avoids getting stuck in local minima or taking overly large steps.



NAG is faster than regular momentum optimization.

Adam Optimization:

Adam (Adaptive Moment Estimation) is an optimization algorithm that combines the benefits of two other techniques: **Momentum** and **RMSProp**. It adapts the learning rate for each parameter based on estimates of first and second moments (mean and variance) of the gradients.

1. **Momentum**: Keeps track of the exponentially decaying average of past gradients, allowing the optimizer to build velocity in consistent directions.
2. **RMSProp**: Adjusts the learning rate for each parameter by normalizing the gradient with a moving average of squared gradients, which helps with varying gradient scales.

Adam computes two moving averages for each parameter:

- **m_t** (first moment, the mean of gradients),
- **v_t** (second moment, the uncentered variance of gradients).

The algorithm then uses these averages to update the weights, making learning more adaptive and efficient. Adam also includes a bias correction step to account for the fact that the averages are initialized as zero and are biased during the initial timesteps.

Key Advantages:

- Adapts learning rates for each parameter.
- Requires fewer hyperparameter adjustments.

- Handles sparse gradients and noisy data well.

Nadam Optimization:

Nadam (Nesterov-accelerated Adaptive Moment Estimation) combines Adam with **Nesterov Accelerated Gradient** (NAG). NAG introduces a "look-ahead" mechanism, where the optimizer takes a small step in the momentum direction before computing the gradient. This improves the accuracy of the gradient estimate, leading to more efficient updates.

Learning Rate Scheduling

Finding a good **learning rate** is crucial for effective training of neural networks. If the learning rate is **too high**, the model's updates become too large, causing the training process to **diverge**, where the loss increases or fluctuates wildly, as discussed in **gradient descent**.

On the other hand, if the learning rate is **too low**, training will **eventually converge**, but it will take a **very long time** to reach the optimal solution, which may be impractical, especially for large datasets or complex models.

If the learning rate is set **slightly too high**, the model will initially make fast progress, but then it will start to oscillate or "dance" around the optimum, never fully settling. This can lead to a suboptimal solution where the model's performance stagnates or oscillates near the best point, but never reaches it.

Additionally, if you have a **limited computing budget**, you might have to stop training before the model converges properly, resulting in a **suboptimal solution** that could have been improved with more time or a better learning rate. Therefore, carefully tuning the learning rate is essential for balancing training speed and convergence quality.

Figure 11-9. Learning curves for various learning rates η

To find a good learning rate, you can **increase it exponentially** from a small value to a large value for a few hundred iterations, then **pick the rate just before** the learning curve starts to spike back up.

Once found, **reinitialize** and train the model with that learning rate.

Instead of using a constant learning rate, you can **start with a high learning rate** and **decrease it** when progress slows, which helps reach a good solution faster. Another strategy is to **vary the learning rate** during training (e.g., start low, increase, then decrease), which is known as **learning schedules**. These schedules help optimize training and are widely used to improve performance.

1. 1Cycle Scheduling:

The **1Cycle Learning Rate Schedule** starts with a relatively small learning rate, gradually increases it to a maximum, and then decreases it again towards the end of training. This approach helps the model converge quickly by allowing rapid exploration of the solution space initially, then fine-tuning for better performance. It often leads to faster and more stable training than other schedules.

2. Power Scheduling:

Power scheduling gradually reduces the learning rate according to a power function of the epoch number. The learning rate decreases slowly over time, allowing for a steady approach to the optimal solution. It's particularly useful for models that benefit from long training periods with decreasing learning rates, such as deep networks.

3. Exponential Scheduling:

In **exponential scheduling**, the learning rate decreases exponentially with each epoch. This means the learning rate drops very quickly in the beginning and slows down as training progresses. This schedule is often used when you want to rapidly reduce the learning rate and fine-tune the model toward the end of training.

4. Performance Scheduling:

Performance scheduling adjusts the learning rate based on the model's performance, usually the validation loss or accuracy. If performance plateaus or worsens, the learning rate is reduced. This schedule is useful for refining the model when progress slows, and it can prevent overfitting by adapting the learning rate based on actual improvements in the model's performance.

Dropout

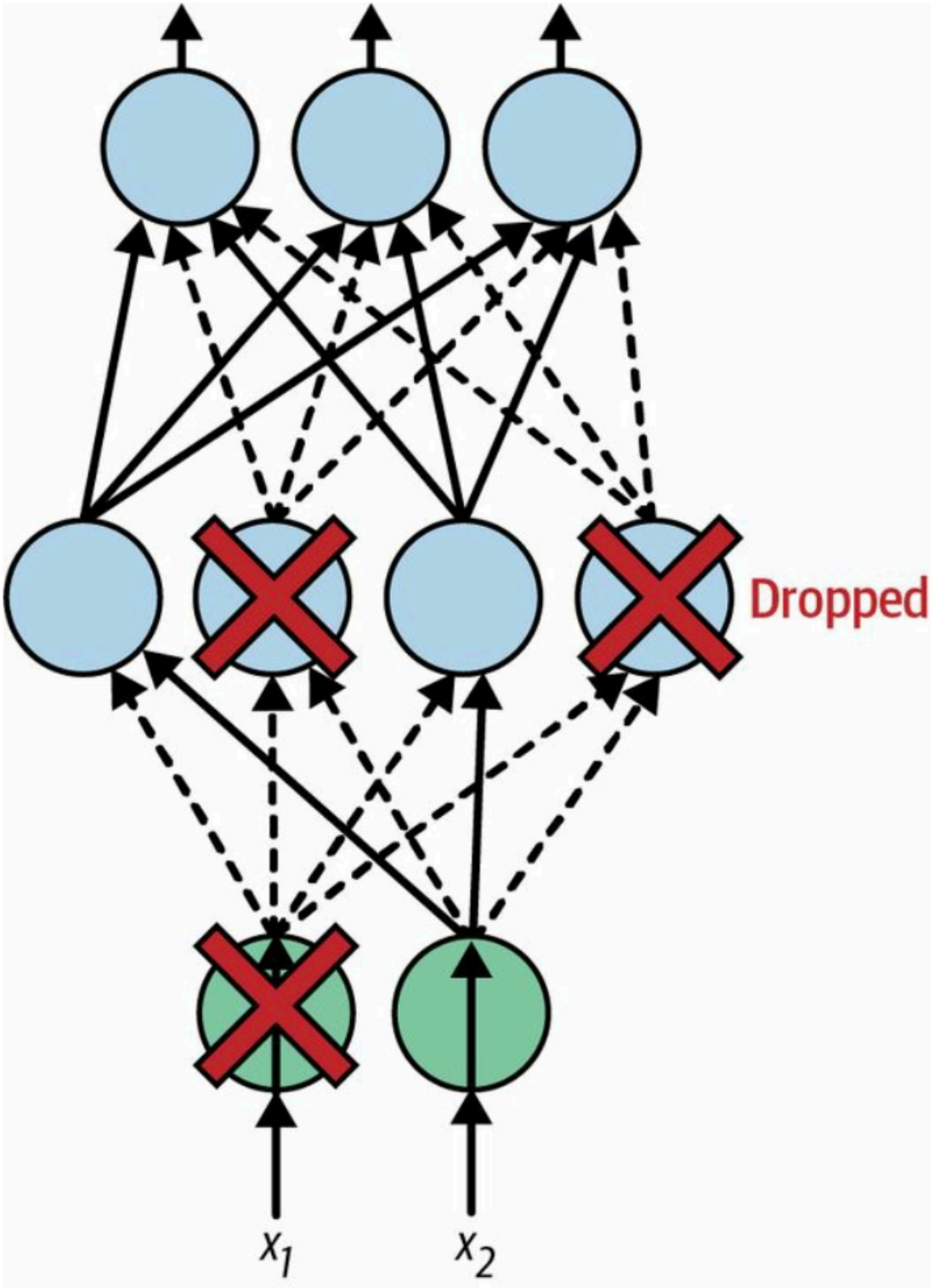
Dropout is a technique used in training neural networks to prevent **overfitting**. During training, it randomly "drops" or **deactivates** a percentage of neurons (along with their connections) in each layer on each forward pass. This forces the network to learn more robust features, as it cannot rely on specific neurons being present all the time. Dropout helps the model generalize better to unseen data by reducing the risk of memorizing the training set.

1-2% accuracy boost

At each training step, every neuron (including the input neurons, but excluding the output neurons) has a probability **p** of being temporarily "dropped out." This means that during that step, the neuron will be completely ignored, but it may be active again in the next training step.

The hyperparameter **p** is called the **dropout rate**, and it is typically set between 10% and 50%. For **recurrent neural networks (RNNs)**, the dropout rate is usually closer to 20%–30%, while in **convolutional neural networks (CNNs)**, it tends to be higher, around 40%–50%. After the training process is complete, dropout is no longer applied, and all neurons are used during inference.

Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end, you get a more robust network that
generalizes better.



Dropout does tend to significantly slow down convergence, but it often results in a better model when tuned properly. So, it is generally well worth the extra time and effort, especially for large models.