How to Deal with Small & Inaccurate Datasets in Neural Networks

Mark-Daniel Leupold

Achieving fully autonomous driving requires AI models that can handle the complexities of real-world roads, from varied weather conditions to unpredictable human behavior. But limited and noisy data often undermines this goal, leading to models that **overfit** and **fail to generalize** in real-world environments.

Advanced deep learning techniques like **BNNs and Monte Carlo Dropout** are addressing these problems by improving uncertainty estimation, robustness, and adaptability — critical for safe decision-making on the road. Additionally, methods like **Contrastive Learning and Knowledge Distillation** enable models to extract maximum value from available data, bringing autonomous driving closer to safe deployment. This article explores how these methods are transforming Neural Networks in general as well as autonomous driving to be more resilient, adaptive, and road-ready.

In this article, we'll explore the **top 20** techniques I found most interesting or commonly used. But before jumping to solutions, we first have to know **what**

problems we need to fix when working with limited and partially inaccurate datasets.

*Note that this is **not** a replacement for a sufficient and good dataset. Just a nice add-on to achieve better outcomes with limited resources.*

Skip to the Technique of your choice (☆ = Personal Favorites):

1. **Dropout** ☆
2. **Data Augmentation**
3. **Transfer Learning**
4. **Bayesian Neural Networks** ☆
5. **Regularization Techniques**
6. **Ensemble Methods**
7. **Monte Carlo Dropout**
8. **Noise Robustness Techniques**
9. **Gradient Clipping**
10. **Synthetic Data Generation**
11. **Adversarial Training**
12. **Knowledge Distillation**
13. **Active Learning**
14. **Contrastive Learning**
15. **Semi-Supervised Learning**
16. **Self-Supervised Learning**
17. **Curriculum Learning**
18. **Online Learning**
19. **Stochastic Weight Averaging**
20. **Optimal Transport (If confident enough)** ☆

Problems

Obviously, limited or inaccurate data results in bad predictions. But there are multiple reasons why that's the case:

Overfitting

Probably the most known one. If your dataset is small or simply **lacks variety** for another reason, the network memorizes the training data instead of learning general patterns. So the model might perform well on the training set but fail miserably afterwards. An overfitted model **cannot generalize** well.

E.g. In the training set of an AD model, the stop sign is always at the same angle, so when the car is driving in the real world it can't recognize stop signs at other angles.

Underfitting

As the name suggests, it's the exact opposite. While an overfitted model generalizes too little and memorizes the training data, an underfitted one **generalizes too much** and **fails to learn the underlying patterns**. The model is too simple or constrained to do so.

E.g. The model is too simplistic to recognize different road features and generalizes everything under the same "road", so in an extreme scenario the car might misclassify a T-junction for a straight road and doesn't turn.

High Variance

This means that the model is **sensitive to small changes** in the training data, leading to **overfitting**. If there are too few data points the model becomes highly dependent on the particular instances it sees during training, thus slight variations can lead to drastic changes in outcomes.

E.g. A small variation in angle, shadows, or lighting leads to the model not identifying the stop sign. Similar to the overfitting scenario.

High Bias

Exactly like high variance causes overfitting, high bias (the opposite) **causes underfitting**. Happening most when approximating a complex problem with a simpler model leading to too much generalization and **insensitivity to large changes**.

E.g. Misclassifying a T-junction for a straight road and not turning. Just like the underfitting example.

Bias-Variance Tradeoff

Tradeoff

This term basically refers to the tradeoff between overfitting (variance) and underfitting (bias).

Models trained on small datasets often suffer from high variance because they capture noise or irrelevant details. If the model is regularized too much to prevent overfitting, it often exhibits high bias. Noisy data can worsen the tradeoff. The model has to balance between ignoring noise and overfitting to it.

The key here is to find the **balance** between overfitting and underfitting. Between too much and too little regularization.

Generalization

In general, generalization grants models the ability to perform well on **unseen data**. However, with small datasets, this is hard to achieve since it's most likely lacking data points and variety.

E.g. An autonomous car trained in urban areas might not be able to generalize well to rural ones, maybe it misclassifies an apple on a tree for a red traffic light and stops.

Poor Model Calibration

Calibration here essentially means how well the model's **predicted probabilities** match the true likelihood of events. When you have few examples, the model frequently produces **overconfident** predictions and if the data is noisy, thus the examples corrupted, the predictions are pretty uncertain and the model is usually **underconfident**.

E.g. An AV might be overconfident that there is an empty road in front of it when an obstacle is hard to see, causing it to crash into it. Conversely, when there actually is an empty road it might be under confident and see obstacles where there are none, resulting in unnecessary braking.

Gradient Instability

Gradient instability refers to a problem in training, where the gradients (i.e., the updates to the model's weights) either become too large (leading to **exploding gradients**) or too small (leading to **vanishing gradients**). This disrupts the learning process, making it difficult for the model to converge to a good solution.

E.g. Gradient instability might lead to inconsistent learning about key driving scenarios like braking, potentially leading the car to crash.

Poor Convergence

When an AI model is training, the key thing it has to do is **reduce the loss** which essentially measures how wrong the prediction was. But in small or noisy datasets, there are too few good examples to guide it to an optimal solution. This makes training either very slow or simply ineffective. It **can't reduce the loss** by a meaningful amount or not at all.

E.g. When the model struggled to converge, it might have never learned to reliably detect pedestrians and potentially crash into one.

Complex Patterns

When data is limited, detecting complex **non-linear patterns** as well as **relationships** between data points falls short, due to a lack of examples and variety.

E.g. A simple relationship might be that if a car turns on the turn signal to the left, it changes to the left lane. An autonomous car without the ability to capture complex patterns might not be able to detect this relationship, doesn't slow down, and crashes into the car that is changing lanes.

Solutions

(source)

1. Dropout ☆

This one is pretty commonly used. Dropout randomly **"drops" neurons in training** (i.e., set to zero). But only in training and not in inference (testing). This forces the model to find different connections and relations of neurons for the same scenario, to learn redundant and robust features that **generalize**. I personally prefer a generous 50% for optimal outcomes but that varies by implementation, sometimes 25% is also sufficient.

With dropout, the model cannot rely too heavily on specific neurons, **preventing overfitting** since it reduces the tendency to memorize the training data. It also makes the training process smoother by distributing learned features across different neurons, **stabilizing gradients**. It helps **reduce variance** by making sure the model doesn't overfit to small variations in the training data.

However, you have to be very careful with the dropout rate. If it's too low, the regularization effect is insufficient. But if it's too high, then it slows down training and convergence since the model only uses a fraction of its neurons in each forward pass. However, it's still very commonly used and can work wonders if implemented right.

There are multiple variations of it, **Monte Carlo Dropout**, which we will explore later in this article, and **Spatial Dropout** for example. In the latter you drop entire feature maps in convolutional layers of CNNs, reducing the reliance on different image features.

That's broadly the idea. But the implementation works as follows:

For every layer *p* neurons are selected and their **activations set to zero**. But usually, the neuron's activation, whatever it is, is either scaled by *1-p* in inference or by *1/(1-p)* in training, the latter is used in PyTorch, to ensure that the sum of the active neurons is consistent with the full model and that the output doesn't vastly differ from inference during training. For example (if it's scaled in training):

$$y = \frac{1}{1 - p} \times f(z)$$

Dropout-Adjusted Neuron-Output

Where the output *y* is equal to the product of *1/(1-p)* and the activated sum of weighted inputs (i.e., the usual neuron output with an activation function f, weight w, input x, and bias b), and if the neuron is dropped then the activation function *f(z) = 0* and it therefore doesn't influence the output anymore.

Pytorch Documentation (for practical usage)

Also, feel free to look at how I used it for Comma.ai's Calibration Challenge.

2. Data Augmentation

Very simple actually. When you lack variation, why not just increase it? In data augmentation, you increase the diversity of the training data set with **transformations**. This also helps with **generalization** since the model sees varied examples during training.

It **can't overfit** to specific angles or conditions when the model is exposed to such variations. But at the same time, it also **can't underfit** (at least it's unlikely) since the model is exposed to a wider range of scenarios that help it

capture **complex patterns** and relationships it might not notice without the augmented data.

If you want to apply this, these are the most common techniques:

- **Image Data:** Rotation, flipping, scaling/zooming, color jittering/applying filters, cropping
- **Text Data:** Synonym replacement, random word deletion/insertion, word order changing in sentences
- **Time Series Data:** Time warping (stretching/compressing of intervals), jittering/adding noise

TSGM Documentation (for practical usage with TensorFlow)

3. Transfer Learning

Transfer Learning involves using a **pre-trained model**, which was previously trained on a large dataset but for a different purpose/task **(source domain)**, as a starting point for **another task (target domain)**. Sort of "transferring the learning" from one task to another. For feature extraction in CNNs, the early layers of the pre-trained model, which capture general features, are **frozen** and the final ones are **fine-tuned**/retrained on the new specific task, but with a **low learning rate** to avoid losing previous knowledge.

E.g. A model previously trained on identifying cats in pictures could be fine-tuned to do the same with dogs, which would be especially useful when there's a limited amount of dogs and an abundance of cat images. But normally this would for example be on large-scale image datasets pre-trained CNN that's fine-tuned for specific object detection or segmentation.

Since the model starts off with useful pre-learned features and weights it's extremely **unlikely to underfit** on limited data. And because it has **already learned complex patterns**, it can **generalize and converge better** to smaller domain-specific tasks leveraging its previous knowledge. It's often used in medical imaging since there's usually a lack of data here.

That's it for the idea. Here's the math behind it:

First, the ANN is pre-trained on the dataset D□ minimizing the source domain loss L□:

$$L_s = \frac{1}{N_s} \times \sum_{i=1}^{N_s} l(f(x_i; \theta),\ y_i)$$

Average Loss (Source Domain)

This is essentially just a normal training process with this formula computing the average loss in the source domain. But although libraries would usually handle this, this formula is important here to understand how it changes with the target domain.

Quite the beginner formula. But for refreshment, it computes the average loss by first summing up the individual losses of each data point in the dataset, which is calculated by comparing the model output $f(x_i;\theta)$ with the label/wanted output z_i in the loss function l, whichever is used, and dividing by the sum by the datapoint count N□ to get the average.

After pre-training it's fine-tuned for the target domain with the dataset D□ minimizing the loss L□:

$$L_t = \frac{1}{N_t} \times \sum_{j=1}^{N_t} l(f(x_j; \theta'),\ z_j)$$

Average Loss (Target Domain)

Notice that although the dataset changes, the model f stays the same, and the parameters θ (theta) get updated to θ'.

For more, PyTorch's tutorials for Object Detection Finetuning and Transfer Learning for CV are really helpful.

4. Bayesian Neural Networks — BNNs ☆

Probability

Instead of having fixed values for weights and activations, BNNs treat them as **probability distributions**, capturing **uncertainty** in their predictions.

Weights are treated as posterior probability distributions instead of number estimates, taking the inherent uncertainty of predictions into account, which reduces sensitivity to noise and minor data variations, therefore **reducing variance** and with it overfitting. With uncertainty estimates also comes **better generalization** to unseen scenarios as well as better calibrated predictions resulting in **less overconfidence**.

Posterior means that we combine our prior beliefs or knowledge about the parameters with new evidence/data observed through logic and Bayesian inferences.

However, these probability distributions and Bayesian inferences are **computationally expensive and time-consuming**, making the training process slower. And this only gets worse when the networks get larger and larger. But for certain tasks, this is definitely worth it.

BNNs are particularly useful in safety-critical applications like autonomous driving, where uncertainty quantification can help identify when the model is unsure of its predictions (e.g., uncertainty-aware object detection, path planning with uncertainty, sensor fusion, or simply ambiguous road signs and difficult weather conditions). And BNNs deal with two types of uncertainty:

- **Aleatoric Uncertainty**: The uncertainty inherent in the data (e.g. noisy measurements), which BNNs deal with by assuming a probability distribution over the outputs.
- **Epistemic Uncertainty**: The uncertainty about the model's parameters due to limited data, which is dealt with via the posterior distribution of the weights.

Enough theory, here's the math behind it: The main goal is to calculate the **posterior predictive distribution**, which represents the probability distribution for possible outputs y* for a new input/data point x*, given the training data D by integrating over the posterior distribution of the weights as follows:

$$p(y^* \mid x^*, D) = \int p(y^* \mid x^*, W) \times p(W|D) \times dW$$

Posterior Predictive Distribution

As you can see, we calculate the posterior predictive distribution **p(y*|x*,D)** by summing up the product of the likelihood and posterior distribution. We multiply the **likelihood p(y*|x*,W)** of the output y* given the new input x* and the neural network weights W, which is essentially the prediction our BNN makes, with the **posterior distribution p(W|D)** of the weights W given the training data D, and with the differential dW since we're integrating.

The integral sums over all possible values for the weights W in the BNN because we have probability distributions and not single fixed point estimates. So we consider all possible weights weighted by their posterior probability.

Before seeing any data, we assign a prior belief over the weights. A common choice is a **Gaussian prior**, $W \sim N(0, \sigma^2)$, assuming weights are normally distributed around zero with some variance.

For calculating the posterior distribution, we can use **Bayes' theorem**, should be familiar to anyone experienced in logic:

$$p(W|D) = \frac{p(D|W) \times p(W)}{p(D)}$$

Posterior Distribution

So the probability of the weights W given the data D equals the probability of the data given the weights p(D|W) times the probability of the weights p(W) divided by the probability of the data p(D).

p(D) is referred to as the **marginal likelihood** (or evidence), essentially just a normalization factor ensuring the posterior distribution integrates to 1. And p(W) embodies the beliefs about the weights before seeing the data. So the only missing part is p(D|W), the likelihood of the data given the weights, which we still have to calculate.

If we assume that the observed outputs y_i follow a Gaussian (normal) distribution around the neural network's prediction $f(x_i,W)$ with some variance σ^2, we can calculate the likelihood of the observed data given the weights as follows:

$$p(D|W) = \prod_{i=1}^{N} N\left(y_i | f(x_i, W), \sigma^2\right)$$

Likelihood

So the total likelihood is equal to the product of the likelihood of each individual data point y_i compared to the neural network output $f(x_i,W)$ with the variance σ^2. The function N() indicates it's a normal distribution centered around $f(x_i,W)$ as explained earlier.

So with all this, we can update our initial formula:

$$p(y^*|x^*, D) = \int p(y^*|x^*, W) \times \frac{\prod_{i=1}^{N} N(y_i|f(x_i, W), \sigma^2) \times p(W)}{p(D)} \times dW$$

Posterior Predictive Distribution

This is very simple actually, even though it first looks complicated. It's still the same formula, the posterior distribution and the likelihood simply got replaced by their individual formulas.

It's basically just **averaging** over all possible weights W, where each weight configuration is weighted by how probable it is given the observed data D (i.e., the posterior), which is captured implicitly through the product of the likelihood and the prior p(W).

So obviously, computing the posterior distribution requires a ton of computing power. But if you don't have access to that, there are other ways to approximate it, for example this one:

$$L_{VI} = KL(q(W) \parallel p(W \mid D))$$

Variational Inference

We approximate the posterior p(W|D) with a simpler variational distribution q(W), often chosen as a Gaussian distribution. The goal is to minimize the **Kullback-Leibler (KL) divergence** between the variational approximation q(W) and the true posterior p(W|D).

But that's how we make predictions in a Bayesian Neural Network by considering all possible weight configurations and their corresponding likelihoods. This makes it highly effective for safety-critical situations, accounting for uncertainty, but also computationally expensive, sometimes making it overkill for some tasks, although effective.

5. Regularization Techniques

They're crucial for controlling model complexity, **preventing overfitting**, and **enhancing generalization**.

Two widely used methods are **L2 Regularization** (also known as Ridge Regression or weight decay) and **Early Stopping**. Below, we'll explore both techniques in depth, how they work, their underlying mathematics, and the purpose of their formulas.

L2 Regularization (Weight Decay)

Sometimes, especially when dealing with limited data, models develop high variance and end up with **large weights**, causing them to memorize the training data and overfit. Whereas smaller weights often lead to smoother and more generalized models, which are way more pleasing to train.

However, it does not address overfitting to noise or handling certain types of complexity in the data. It only controls the magnitude of the weights and prevents overfitting resulting from that.

L2 differs from L1 by adding a term proportional to the square of the weights in a loss function, **penalizing large weights more heavily** for errors:

$$L(\theta) = \frac{1}{N} \times \sum_{i=1}^{N} l(f(x_i; \theta),\ y_i) + \lambda \times \sum_{j=1}^{M} ||\theta_j||^2$$

Loss Function in L2 Regularization

The first part should look familiar, as already discussed in the transfer learning section. But to catch up: It computes the **average loss** over the dataset by comparing the model's prediction to the real data for each data point i and averaging that.

But in L2 Loss, a second **regularization term** is added. And if you take a closer look, it gets pretty self-explanatory: we add a product to the loss function that increases the loss for larger weights by **summing up the**

squares of each weight θ□ in M (the number of model parameters) and multiplying that by the **regularization factor λ** (lambda), also called the "strength" of the regularization.

As mentioned, too much regularization can cause underfitting. But too little results in overfitting, which is why the regularization factor λ has to be chosen carefully.

If unfamiliar with the two vertical bars in $||\theta\square||^2$ They just represent the L2 norm (Euclidean norm) of the weight vector. And when we square it we get $\theta\square^2$, the squared value of each individual weight. If θ□ is a vector, it is the sum of the squares of all the components in that vector.

This quadratic penalty grows rapidly as the weights increase, so L2 encourages smaller, more distributed weights.

During training, the gradient-based optimization algorithm (e.g. stochastic gradient descent) updates the model parameters based on both the data loss and the regularization term. For that reason we also need a second term when updating the gradient:

$$\theta_j^{new} = \theta_j^{old} - \eta \times \left(\frac{\partial l}{\partial \theta_j} + \lambda \times \theta_j^{old} \right)$$

Gradient Update in L2

Normally, we'd calculate the the updated weight parameter after applying the gradient step by subtracting from the old weight parameter the product of the **learning rate η** (eta) and the gradient of the data loss with respect to the weight (the fraction), it tells us how much the loss would change if we slightly adjusted θ.

But in L2 Loss, we have to add the gradient of the L2 penalty with respect to the weight, the product of the regularization factor lambda, and the old weight parameter, pulling the weight closer to zero and **shrinking each weight by a factor of (1−ηλ) at every step.**

If you want to use it, look at the weight-decay part in this PyTorch document about optimization algorithms.

Early Stopping

This one is very straightforward: **training halts when the current performance on the validation set is worse than its previous one**:

$$L_{val}(\theta_{epoch}) \geqslant L_{val}(\theta_{best})$$

Early Stopping Check

So the moment the loss of the current epoch (run through a dataset) in inference is larger than on its best one, training stops.

Usually, you train an ANN on the training set on a set amount of epochs to reduce the loss with backpropagation and then run the same model on the validation set during inference without adjusting its parameters to check its performance on unseen data. This process gets repeated over and over again until the ANN is trained sufficiently. Early Stopping just stops this process and resets the parameters to where they were an iteration before once it detects the model's performance on the validation set was worse than earlier, which usually indicates it overfitted to the training data.

This automatically **prevents overfitting** and is obviously **very efficient**.

6. Ensemble Learning

An ensemble model consists of N individual models (in our case N neural networks) trained independently. They differ in architecture, training data subsets, and hyperparameters. They all make predictions, which are then **combined** to get a more accurate output.

The training should be pretty self-explanatory, we simply do the same thing but with **multiple models**. But there are two different ways to combine those predictions. Starting with the most simple:

1. Averaging

We simply average each of the N model's prediction $\hat{y}_i$ to get the overall output $\hat{y}$:

$$\hat{y} = \frac{1}{N} \sum_{i=1}^{N} \hat{y}_i$$

Averaging

1. Voting

Or, in classification tasks, a majority or soft vote can be used. In a **hard vote**, the model casts a vote for its predicted class and the one with the most votes is selected. Or, in a **soft vote**, the final class probabilities are averaged like this:

$$P(y = k \mid x) = \frac{1}{N} \sum_{i=1}^{N} P_i(y = k|x)$$

Voting

Where the probability of class k from each of the N models is averaged to get the overall probability for it. Besides the discussion, there are multiple additional approaches. Some common variations of ensemble techniques include:

- **Bagging** (e.g., Random Forests): Each model is trained on a different random subset of the training data, often with replacement (bootstrapping).
- **Boosting** (e.g., AdaBoost, XGBoost): Models are trained sequentially, with each new model focusing on the errors made by the previous ones.
- **Stacking**: Combining the predictions of multiple models using another model (meta-learner), which learns how to combine the outputs of the base models best.

Ensembles inherently provide two types of uncertainty, **aleatoric** and **epistemic** uncertainty (look at their explanations in the BNN section if unfamiliar). Aleatoric uncertainty is dealt with by training different models that make different predictions, thus accounting for data variability. Epistemic uncertainty is indicated if the models disagree with each other too much, likely due to limited knowledge:

$$Var(\hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - \bar{y})^2$$

Epistemic Uncertainty

The variance of the mean prediction gives us a measure of epistemic uncertainty. To get that, we take the average of the squares of the difference between each model's prediction and the average prediction. The higher it is, the more likely is a lack of knowledge regarding the input.

Since we use multiple different models, this can help a lot with noisy and limited data, with each model capturing different features. But it's also more computationally expensive due to more models used, making it not always a good choice, especially when computing power is limited.

Look at the PyTorch documentation if interested in programming it.

7. Monte Carlo Dropout

Monte Carlo

Exactly like dropout, you “drop” neurons in training to produce more generalized outputs as well as reduce overfitting and reliance on specific neurons. But unlike traditional dropout, in MC-Dropout, the process is also performed in **inference** to get more accurate predictions. Overall Monte Carlo Dropout works as follows:

1. **Train the Model with Dropout**: Train a neural network as usual with dropout layers where a certain percentage of neurons are deactivated in training
2. **Perform Multiple Stochastic Forward Passes**: During inference, perform **T** forward passes, each time randomly dropping out different neurons.
3. **Aggregate Predictions**: Average the predictions from these stochastic passes to obtain the final prediction and estimate the uncertainty.

So during inference, we calculate T different outputs, each time dropping out different neurons to get (hopefully slightly) different results. Then, those results are averaged as usual:

$$\hat{y} = \frac{1}{T}\sum_{t=1}^{T}\hat{y}^{(t)}$$

Mean Prediction

Once we’ve obtained our averaged prediction, we can calculate how unlikely that prediction is using epistemic uncertainty, exactly like we did with the ensembles in the last chapter:

$$Var(\hat{y}) = \frac{1}{T}\sum_{t=1}^{T}\left(\hat{y}^{(t)} - \hat{y}\right)^2$$

Epistemic Uncertainty

But the loss function, whatever it may be, needs to be regularized in order to account for the inactive neurons like this:

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(f(x_i;\theta),\ y_i) + \lambda\ ||\theta||^2$$

Loss Function in MC Dropout

Where we have our normal loss function for the data loss and on the left add a second term to it: $\lambda||\theta||^2$. With λ being a regularization parameter. This accounts for our epistemic uncertainty in the prediction by penalizing large deviations from the prior distributions, making sure our model is not overly confident about its weight values.

MC Dropout enables **uncertainty estimation** without requiring the full complexity of Bayesian inference like it's the case with BNNs, making it suitable for large, deep networks. But it also slows down inference since multiple predictions have to be made, so you have to evaluate if the extra generalization and uncertainty estimates are worth the speed reduction.

8. Noise Robustness Techniques

Robustness to noise is as important as regularization. You will never find a perfect dataset. There will always be inaccurate measures and noise. But luckily there are a lot of ways to deal with them, two of which, label smoothing and noisy student training, we'll explore here:

Label Smoothing

In a standard classification task, you'd usually assign the label 1 for the true label and 0 to the rest. But this can cause **overconfidence** and the model to assign excessively high probabilities to the predicted class, leading to poor **generalization**.

Label smoothing addresses both of those problems by **softening the hard 0 or 1 targets**. Instead, the true label gets a slightly lower value assigned and the wrong ones a slightly higher one.

But if precise label information is critical, for example when you have to differentiate human faces, this will hurt the model. Plus if the data is very clean, this might also lower the accuracy. For these sorts of tasks, label smoothing would be a bad choice. But when dealing with noisy data and for

tasks like identifying a pedestrian in autonomous driving and not identifying the exact human, this might be extremely beneficial.

Usually in a classification class with cross-entropy loss, the loss function for a single data point is:

$$L = -\sum_{k=1}^{C} y_k \times \log(p_k)$$

Cross Entropy Loss

Where we sum up for every class k of C the product of the logarithm of the predicted probability p and the actual label y (0 or 1) before multiplying that sum by -1.

For more about Cross-Entropy Loss, take a look at the PyTorch documentation here.

With label smoothing the true label y is replaced by a "smoothed" version as follows:

$$y_k^{smooth} = y_k(1-\epsilon) + \frac{\epsilon}{C}$$

Label Smoothing

Here the old actual label is multiplied by 1 minus the **smoothing parameter** ϵ **(epsilon)** between 0 and 1 and then the ϵ divided by the number of classes is added. This results in every wrong label 0 getting larger by the factor ϵ/C and the true label of 1 getting shrunk by the factor ϵ before the same ϵ/C is also added to it so that the overall sum of probabilities still adds up to 1.

Noisy Student Training

This is a **semi-supervised learning** approach, which we'll go over later. It extends the idea of **self-learning** by adding noise to the student model in training, forcing the model to **generalize more** while also **dealing with the lack of data**.

The process typically goes like this:

1. A **teacher model** is trained on a labeled dataset
2. The trained teacher model **generates pseudo-labels** for unlabeled data
3. The **student model** is trained on both the labeled data and the pseudo-labeled data while noise (e.g., dropout, data augmentation, or noisy inputs) is added to make it more robust.

Let's go over this step by step. Starting with the teacher model:

$$L_t = \frac{1}{|D_L|} \sum_{(x,y) \in D_L} l(f_t(x), y)$$

Teacher Model

The teacher is trained on the labeled data DL, as usual, using standard loss like cross-entropy. The total loss is, again, the averaged loss across all labeled data points compared to the corresponding model prediction.

Then, we'd normally use the model to generate labels for unseen data and that would be the application. But here, we're not done yet, we generate the pseudo-labels, but only to expand our existing dataset:

$$\hat{y}_u = f_t(x_u)$$

Pseudo Labels

These pseudo-labels are generated by the teacher model predicting the labels of unseen data, as it normally would in practice. With the teacher model f predicting the label y for the corresponding input x.

Then, the new student model is trained on the labeled and pseudo-labeled data as follows:

$$L_s = \frac{1}{|D_L|} \sum_{(x,y) \in D_L} l(f_s(x), y) + \frac{1}{|D_U|} \sum_{x_u \in D_U} l(f_s(x_u), \hat{y}_u)$$

Student Model

Notice that the first part is exactly the same as the loss function of the teacher model since this is the part where both train on the same labeled data DL. But then we also add the loss for the pseudo-labeled data DU with the pseudo-labels created by the teacher model and the input data modified with noise.

However, while this has its benefits, this technique **highly depends on the quality of the teacher model**. If the initial dataset is too small or too noisy to train a partially accurate model, this is not the technique to use. This should only be done when the initial model is already quite accurate and you simply want to improve it, then it could result in excessive performance increases.

9. Gradient Clipping

Particularly in deep neural networks and RNNs gradients can become too large during backpropagation (i.e., i.e., the updates to the model's weights), which is known as a problem called **exploding gradients**, causing divergence or extremely slow training and gradient instability.

Gradient clipping ensures they stay in a manageable range when updating the weights, for example by preventing the gradients from exceeding a certain threshold. It **limits the magnitude of the gradient updates** when the gradient of the loss function with respect to the model parameters becomes too large.

There are two primary types of gradient clipping:

1. Norm-Based Gradient Clipping

This one is the more common form. If the gradient's norm exceeds a predefined threshold T, we rescale the gradient vector g so that its norm equals T. Basically we **prevent it from exceeding the threshold T**.

If ||g||2 > T, we rescale the gradient like this:

$$g' = g \times \frac{T}{\|g\|_2}$$

Norm-Based Gradient Clipping

The **clipped gradient g'** is the product of the original gradient g and the quotient of the threshold T and the Euclidean norm of the original gradient g. After clipping the gradient the updated parameters at each time-step t are:

$$\theta_{t+1} = \theta_t - \eta \times g'$$

Updated Parameters

So the updated model parameters θ are equal to the difference between the previous ones and the product of the learning rate η (eta) and the clipped gradient g'.

2. Value-Based Gradient Clipping

In this case, instead of clipping based on the norm, we clip each individual component of the gradient vector to a maximum value v. Each element g_i of the gradient vector g is clipped as follows:

$$g_i' = \max(\min(g_i, v_{\max}), -v_{\max})$$

Value-Based Gradient Clipping

So that each gradient element cannot exceed the threshold v, be it in the positive or negative realm. This approach prevents any single gradient component from having a disproportionate effect on the update as it could happen with the norm-based approach.

But as always when dealing with set constants like the threshold T or v, they require careful tuning so that they are neither too low, causing slower convergence, nor too high, creating too little impact.

For a short guide on how to do it in PyTorch, check this out.

10. Synthetic Data Generation

Same as data augmentation. If you lack data, why not just increase the amount of it? Pretty straightforward, synthetically generating training data.

General Adversarial Networks — GANs

GANs consist of two neural networks, a **Generator** (G) and a **Discriminator** (D), which are trained in opposition. The generator attempts to create realistic data, while the discriminator evaluates whether data is real or generated. The goal is for the generator to produce samples so realistic that the discriminator cannot distinguish between real and fake data. Both **compete** against each other, which is why the training process can be described as a simple **minimax** game:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log(D(x))] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z))]$$

Minimax

The generator learns by sampling noise z~pz(z) and attempting to map it to data that resembles the real data. The discriminator learns by distinguishing between real data x~pdata(x) and generated data G(z). The generator is updated to minimize log(1−D(G(z))), which is the minimization of the probability that the discriminator correctly identifies generated data G(z) as fake. And the discriminator is updated to maximize logD(x)+log(1−D(G(z))), which is the maximization of the probability of correctly classifying real data x as real and generated data G(z) as fake.

By creating artificial examples and training with them, we solve the lack of data. But the outcome heavily depends on the quality of the data created by the generator.

If you want to learn how to build GANs, I highly recommend the PyTorch tutorial for it.

Simulation Environments

This one is very commonly used in robotics and AV training. If you train your self-driving cars in the real world, they would crash a lot and it would be a time-consuming process, since techniques like **parallelization** can't be used here. For more depth check out my simple simulation of an autonomous car here.

So instead of training in the real world, simulations of it are created. With laws of physics, about e.g. object movement and kinetic energy, to **resemble the real world as closely as possible**.

Typically, reinforcement learning (RL) is used in these simulations. If that's the case, we usually deal with the **Markov Decision Process (MDP)**, which provides a mathematical framework for decision-making in environments where outcomes are partly random. The objective is to find a policy π(a|s) that maximizes the **expected cumulative reward** (discounted sum of future rewards):

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$$

Expected Cumulative Reward

Here, R(s,a) is the received reward by taking action in the state s. That reward is multiplied by the discount factor γ to the power of the iterator, which reduces the impact of rewards further in the future since γ is set to a value between 0 and 1. And finally, this product is summed to get the expected future rewards of an action, the expected cumulative reward.

Simulation environments are extremely helpful due to their high scalability. On top of the fact that nothing physically breaks as it's likely to happen in real-world training, a robot or self-driving car is able to go through thousands of hours in training by simultaneously training in hundreds of simulation environments at the same time through parallelization. Potentially even different simulations for better generalization. But all this depends on the use case. For a simple image classifier, a simulation is not the way to go, but for a robot, it can be really beneficial. Unitree's robot dog Go2 for example heavily makes use of training in realistic simulations.

11. Adversarial Training

Instead of training the model solely on clean data, we include **adversarial examples**, which are **inputs that are intentionally modified** to lead the model to incorrect predictions, forcing it to **generalize** more and learn more robust decision boundaries that are **less sensitive** to small variations in the input space.

Generating Adversarial Examples

A common method to generate such examples is the **Fast Gradient Sign Method (FGSM):**

$$x_{adv} = x + \epsilon \times \text{sgn}(\nabla_x L(x, y; \theta))$$

Fast Gradient Sign Method (FGSM)

The adversarial example is generated by adding a small amount of noise in the direction of the loss function. We do this by multiplying the magnitude of perturbation ϵ (epsilon) with the sign of the gradient ∇ of the loss function $L(x,y;\theta)$ with respect to the input x, so $\text{sign}(\nabla_x L(x,y;\theta))$, which determines the direction of the perturbation and tells us how much the loss would change if we modify the input.

The second most common generation method is Projected Gradient Descent (PGD) if you're interested in that, but the objective is the same so we won't touch it.

Training with Adversarial Examples

Training involves minimizing the following objectives:

$$\min_{\theta} \mathbb{E}_{(x,y) \sim D} \times \left[\max_{\delta : \|\delta\| \leqslant \epsilon} L(x + \delta, y; \theta) \right]$$

Adversarial Training

This is essentially a simple **min-max optimization problem** where the inner maximization finds the most adversarial example by maximizing the loss $L(x+\delta,y;\theta)$, while the outer minimization adjusts the model parameters θ to minimize the loss on both clean and adversarial examples.

The adversarial perturbation δ (delta) applied to the input x is constrained by ϵ in the expression $||\delta|| \leq \epsilon$, ensuring that the adversarial examples are similar to the original data and not too different. It assumes the worst-case scenario in $\max\delta$ where the model is optimized against the strongest perturbation that

can fool it. Finally, adversarial training is done over the entire dataset D, indicated by E(x,y)~D.

This is not just a good technique to combat noisy data, but also gradient-based adversarial attacks like FGSM or PGD. However, it is computationally pretty expensive and may hurt the accuracy of clean data, so as usual, its effects and usefulness vary by application.

Take a look at the PyTorch tutorial if interested.

12. Knowledge Distillation

Teacher & Student

We have discussed a bunch of techniques to deal with various problems resulting from limited and noisy data. But those techniques are often quite computationally expensive and sometimes that's a big issue. Knowledge distillation aims to **reduce model and computational complexity**, while also improving **generalization** through a smaller model architecture in the process.

This is done by **transferring knowledge** from a large and complex **teacher model** to a smaller and simpler **student model**. The goal is that the student retains the predictive power of the teacher model while using less resources.

This is done by training the student model not only on the true labels but also on the **soft targets** produced by the teacher model. These are probability distributions over the different outputs that the teacher model predicts. Even if they aren't all accurate, the student still learns the subtle patterns the teacher has learned.

First, the teacher model is trained and then produces the soft labels with a softmax function, which is defined as follows:

$$\sigma(z_i; T) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

Softmax

The **softmax function σ** (sigma) takes the raw outputs z of class i and a **temperature T** as input and computes the output by dividing e to the power of the quotient of the raw outputs and the temperature by the sum of e to the power of the quotient of each individual output and the temperature.

A higher temperature T produces a softer probability distribution and a lower T produces harder more confident predictions.

Then, after we have the soft targets, the student model is trained using a combination of two losses, one for the true labels and one for the soft ones. The first could for example be Cross-Entropy loss, as already discussed:

$$L_{CE} = -\sum y_i \times \log(p_i)$$

Cross Entropy Loss

With y being the true label and p the predicted one, the network is normally trained by trying to minimize the total loss, which is the negative sum of the product of all true labels y_i and the logarithm of the corresponding predicted one p_i.

The second loss function for the student model is the **Kullback-Leibler (KL) divergence loss** on the soft targets (distilled knowledge):

$$L_{KD} = T^2 \times KL\left(\sigma\left(z^{teacher}; T\right) \,||\, \sigma\left(z^{student}; T\right)\right)$$

Kullback-Leibler Divergence Loss

The KD-Loss is here equal to the square of the temperature T, which is the scaling factor that adjusts the magnitude of the gradients and arises from the derivative of the KL divergence with respect to the logits when T is applied, it's there to ensure the proper flow of gradients during backpropagation, times the KL divergence of the by the factor T softened probability distributions of the teachers and students logits, essentially the distance between these two softened distributions.

So our second loss is equal to the scaling factor times the softened distribution distance. And with both of those losses, we can calculate the overall total loss like this:

$$L_{total} = \alpha\, L_{CE} + (1-\alpha) L_{KD}$$

Total Loss

We simply weigh them by a factor alpha and (1−α) to balance the contribution of the hard labels and soft targets. As you can clearly see, the higher α, the higher the contribution of the true labels, the lower the one of the soft targets, and vice versa.

While there are often some accuracy losses, this technique finds great use in resource-constrained environments. For example, Google used it to create their model DistilBert, used for **real-time applications** like question answering or translation, which they created by distilling knowledge from their LLM BERT. But it could also be used in real-time reactions to unexpected situations or accidents in self-driving cars, where fast responses are necessary.

Look at this if interested in its implementation in PyTorch.

13. Active Learning

When labeled data is limited or especially when labeling is time-consuming, active learning is often used. First, the model is trained on a small labeled dataset. And after each training iteration, the model selects and **queries for the labels of a subset of unlabeled data points that** it's most uncertain about and believes will improve its performance the most.

These newly labeled points are then added to the training set, and the model is retrained. This is repeated iteratively, with the model becoming increasingly refined as it focuses on the **most informative examples**.

Overall, the training process in active learning could be summed up like this:

1. Train the model on the small labeled dataset
2. Select the most uncertain examples of the unlabeled data
3. Request their labels from an oracle (e.g. a human expert)
4. Retrain the model after adding the newly labeled points to the dataset

Training, querying, and labeling should be fairly straightforward. But the most interesting part of active learning is its way of selecting the most uncertain examples. And there are mainly 2 strategies used for it:

1. Least Confidence

The simplest of the three. Select the points where the model's probability of the most likely class is the lowest, so where it is most unsure about its most likely prediction:

$$x^* = \arg \min_x \max_i p(y_i|x)$$

Least Confidence

First, we look at what class the model believes is most likely to be the output of a certain datapoint with the maximization. Then we take the minimum one of those probabilities to get the most uncertain example.

2. Margin Sampling

Here the model selects the points where the difference between the two top probabilities is the smallest:

$$x^* = \arg\min_x \left(p(y_1|x) - p(y_2|x)\right)$$

Margin Sampling

It does this by first calculating the difference between the points with the two highest probabilities and then selecting the point where this difference is the smallest through minimization.

A third method would be entropy sampling, selecting the points with the highest uncertainty in terms of entropy. But you get the idea that this is about finding the most uncertain predictions and of course, there are numerous ways to do so.

Overall it's a good approach to achieve more with fewer resources but it requires a reliable oracle, and if that’s not present active learning is doomed to fail. So again, its effects vary by application.

14. Contrastive Learning

It’s a self-supervised learning technique where the model learns to **differentiate between similar and dissimilar data points**. The key idea is to pull together similar (positive) pairs and push apart dissimilar (negative) pairs in the representation space.

First, we create pairs of data points:

- **Positive Pairs**: similar instances (e.g. different views of the same image)
- **Negative Pairs**: dissimilar instances (e.g. two different images)

The goal is to train the model to produce representations where positive pairs are close to each other and negative pairs are far apart. Basic unsupervised learning essentially. The contrastive loss is calculated for example with the InfoNCE loss (with the earlier explained objective):

$$L_{contrastive} = -\log\left(\frac{\exp(sim(M(x_i),\ M(\hat{x}_i))/\tau)}{\sum_j \exp(sim(M(x_i),\ M(x_j))/\tau)}\right)$$

InfoNCE (Noise Contrastive Estimation) Loss

It's the negative logarithm of the ratio between the positive similarity and the total similarity (positive + negatives). With τ (tau) being a temperature parameter to control the sharpness of the similarity distribution. This encourages the positive pair to be much closer together than any negative pairs and the log function makes sure that when the similarity between positive pairs is already high (near 1), the gain from further increasing it diminishes, leading to more balanced updates.

This is an unsupervised learning technique, so there is **no need for labeled data**, making it excellent for scenarios where labeled data is scarce. But it becomes really **computationally expensive** the larger the dataset gets. It's still used quite often though, for example in computer vision models like SimCLR or MoCo.

15. Semi-Supervised Learning

I'd assume everyone reading this is familiar with the different ML techniques: supervised, unsupervised, and reinforcement learning. Semi-supervised learning sort of **bridges the gap between supervised and unsupervised learning**, between requiring large labeled datasets and only using unlabeled data.

When it comes to the math, semi-supervised learning combines two losses, the supervised and unsupervised one:

$$L_{total} = \alpha L_{supervised} + \beta L_{unsupervised}$$

Total Loss

We simply add them together and weigh each by a factor alpha and beta controlling the contributions of each.

Supervised Loss

The supervised loss, for the labeled data, is typically the cross-entropy loss between the model's predictions and the true labels:

$$L_{supervised} = -\sum_{i} y_i \log(p(y_i|x_i))$$

CE Loss

Again, the negative sum of the products of each label and the logarithm of the corresponding model prediction.

Unsupervised Loss

For the unlabeled data, the unsupervised loss encourages the model to produce confident, consistent predictions on the unlabeled data. Techniques to achieve this include the following two:

1. Consistency Regularization

$$L_{unsupervised} = \sum_{j} || M(x_j) - M(\hat{x}_j) ||$$

Consistency Regularization

The model is trained to produce similar predictions when inputs are perturbed or augmented. Where x^☐ is a perturbed version of x☐ and the sum of all the differences between two outputs (when given the perturbed and not perturbed input) is our loss which we aim to minimize.

2. Entropy Minimization

$$L_{unsupervised} = -\sum_{j}\sum_{k} p(y_k|x_j) \log(p(y_k|x_j))$$

Entropy Minimization

The model is trained to produce confident predictions by minimizing the entropy of the predictions on the unlabeled data.

You can probably see this comes in very handy when a **limited amount of labeled data** but a lot of unlabeled data is available and it also improves generalization.

16. Self-Supervised Learning

In contrast to the name, this is a form of unsupervised learning. The model is trained using **pretext tasks**, which are tasks that require predicting some aspect of the input data like missing parts or transformed versions. Once the model learns good representations from these tasks, these representations can be fine-tuned or used as features for **downstream tasks**, e.g. classification or detection.

An example of such a pretext task would be the use of **contrastive learning**, as discussed earlier it involves learning representations by making similar (positive) pairs closer in representation space while pushing apart dissimilar (negative) pairs:

$$L_{contrastive} = -\log\left(\frac{\exp\left(sim\left(f(x),\ f\left(\hat{x}^{+}\right)\right)\right)}{\sum_{\hat{x}^{-}}\exp\left(sim\left(f(x),\ f\left(\hat{x}^{-}\right)\right)\right)}\right)$$

Contrastive Learning

While f(x) and f(x^+) are representations of a positive pair (e.g., an image and its augmented version), f(x) and f(x^-) are negative examples compared in a similarity measure via the function sim(), for example, cosine similarity.

The main benefit is that no labeled data is required, particularly useful in Computer Vision models, reducing reliance on labeled image datasets. But its success heavily depends on the design of the pretext tasks, which when poorly chosen results in unuseful representations for the downstream tasks.

17. Curriculum Learning

Normally you'd train your ANN like this:

$$\min_{\theta} L(M(x_i), y_i),\ (x_i, y_i) \in D$$

Training

You're adjusting the model parameters θ to minimize the loss L between the output y generated by your model M with a certain input x and the real output of the datapoint (x,y) in the labeled dataset D.

However, this is not always the optimal approach. **It would be like trying to teach a student Calculus and basic addition at the same time.**

So in curriculum learning, as its name suggests, the model first gets trained on the easy examples and then progresses to more and more difficult and complex ones.

But, defining a curriculum and **organizing the dataset on difficulty** takes a lot of time. And while you could train a second model to do this, this technique is mostly used with **smaller datasets**. Not only because the organizing is less time-consuming, but also because it leads to **extremely fast convergence** and improved generalization, which is often a bigger issue here. Its effects are most notable in **reinforcement learning** tasks where agents can start in simpler environments to progress to more complex ones, like levels in a video game.

18. Online Learning

Online Learning

Surprisingly, this doesn't have anything to do with actual online learning. It's an ML method where the model learns on a stream of data one sample at a time instead of in batches or on the entire dataset at once.

For every data point, the model calculates its predicted output as usual. Observe the true output and update its parameters at every single data point:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t, x_t, y_t)$$

Training

The updated parameters are equal to the old ones minus the **learning rate η** (eta) times the **gradient of the loss with respect to the model's parameters** ∇θL.

Updating the gradient at every new data point makes the model **highly adaptive** to changing environments, but it also might result in forgetting previous data if the distributions change over time and it can cause instability, especially with noisy data. But the high adaptability is making it very useful in applications like robotics where it's used to constantly update robots on the go as they receive more sensory inputs.

19. Stochastic Weight Averaging — SWA

If none of the techniques above **generalize** enough for you, then Stochastic Weight Averaging (SWA) might be what you're looking for. It works like this:

1. **Base Training**: Train the ANN to a point where it's quite accurate and converges to a good point using your usual methods.
2. **Weight Sampling**: After the model reaches a flat region of low training loss, SWA saves the weights at regular intervals from multiple epochs or cycles.
3. **Weight Averaging**: When training is complete, the final model doesn't use the weights from the last training epoch, but averages all in the previous step collected weights and uses that.

Actually quite a simple process. The saved weights W from training at the time period t are averaged:

$$W_{SWA} = \frac{1}{k} \sum_{t=t_1}^{t_k} W_t$$

Weight Averaging

And that average, the W SWA, is used as the weight. This comes with a lot better generalization as mentioned, but it's also easy to implement since it doesn't require any hyperparameter tuning. Plus it also helps with sharp minima issues, where the performance drops significantly when the model moves slightly from the optimal point. So overall it's a good technique to "explore more of the coast landscape".

If interested in its implementation in Pytorch, take a look at this.

20. Optimal Transport ☆

One of my personal favorites. When e.g. autonomous cars are trained in simulation environments, that knowledge gained there has to be **transferred** or transported to the real world. Optimal transport is a powerful mathematical framework for measuring the difference between two probability distributions, which is key in these situations.

The goal is to align the **source domain** (where we have plenty of data, e.g. a simulation) with the **target domain** (where we have limited data, e.g. the real world) in a way that minimizes the cost of transforming the source data to match the target distribution.

Wasserstein Distance

The **Wasserstein distance** (a form of OT distance/cost) between two distributions μ and ν is the minimum cost of transporting probability mass from μ to ν, given a cost function c(x□,x□) that measures the cost of moving a point x□ from the source to x□ in the target. Common choices for c include the Euclidean distance ||x□-x□||².

The **Monge-Kantorovich formulation** of OT solves for the Wasserstein distance:

$$W_c(\mu, \upsilon) = \min_{\gamma \in \prod(\mu,\upsilon)} \int_{X \times X} c(x_s, x_t)\, d\gamma(x_s, x_t)$$

Monge-Kantorovich Formulation

Couples, ∏(μ,ν), are probability distributions on the product space x*x that have marginals μ and ν. They represent different ways to “pair up” points from the two distributions. Finding the right pairing (or coupling) that minimizes the cost is the essence of the OT problem. The integral computes the total cost of a particular **transport plan** γ, and the minimization ensures we find the optimal plan that gives the smallest possible transport cost, i.e., the **Wasserstein distance**.

This is great, but in practice, computing OT is expensive, especially in high-dimensional spaces, so regularization techniques like **entropy regularization** are applied:

$$W_c^{\lambda}(\mu, \nu) = \min_{\gamma \in \prod(\mu,\nu)} \int c(x_s, x_t)\, d\gamma(x_s, x_t) + \lambda H(\gamma)$$

Entropic Regularized Wasserstein Distance

Which is basically still the same thing, but we add λH(γ) as a regularization term to it. It regularizes the problem by penalizing transport plans that are too concentrated/sharp and encourages plans that distribute the transported mass over more locations in a more balanced way. But besides that, adding entropy makes the optimization problem convex and solvable using **Sinkhorn iterations**, which are faster and scalable.

Whether you use the regularized version is completely up to you. Both are good ways to compute the transportation cost and find the best transportation plan.

In ML, the Wasserstein distance has also been used in Generative Adversarial Networks, where it helps to improve the training stability by better capturing differences between the generated and real data distributions, so-called **Wasserstein GANs**. And in **domain adaptation** tasks, OT can be used to map data points from one domain to another by minimizing the Wasserstein distance between the source and target distributions, as mentioned in the example with the simulation.

TL;DR

Issues resulting from limited and noisy data can be dealt with by advanced techniques in deep learning designed to improve model performance, robustness, and generalizability. **Dropout** and **Monte Carlo Dropout** reduce overfitting by randomly deactivating neurons, while **Regularization** (e.g., L2, early stopping) constrains model complexity.

Data Augmentation and **Synthetic Data Generation** expand training sets by creating modified or artificial data, while **Transfer Learning** leverages pre-trained models to accelerate learning in similar domains. **Bayesian Neural Networks** (BNNs) provide probabilistic interpretations of predictions, enhancing uncertainty quantification, and **Ensemble Methods** combine multiple models to improve robustness.

To further enhance model stability, **Gradient Clipping** mitigates gradient explosion, and **Adversarial Training** strengthens resilience against adversarial examples. **Noise Robustness Techniques** like Label Smoothing aid in dealing with noisy labels, while **Knowledge Distillation** transfers knowledge from complex to simpler models. **Active Learning** selectively samples informative data points, optimizing data efficiency, and **Contrastive Learning** shapes representations by maximizing similarity within positive pairs and dissimilarity with negatives, enhancing feature richness.

Semi- and Self-Supervised Learning leverage unlabeled data effectively, while **Curriculum Learning** gradually increases data complexity for smoother training. **Online Learning** continuously updates the model with incoming data, constantly adapting to new information.

Finally, **Stochastic Weight Averaging (SWA)** stabilizes training by averaging model weights, and **Optimal Transport** aligns distributions, enhancing latent space regularity for generative models, such as in **Wasserstein Autoencoders** or optimizing knowledge transfer between source and target domains.

All these techniques have their own individual use cases and effects. Often multiple ones are combined with each other for optimal outcomes. Look at their documentation and tutorials if you wish to dive deeper into their exact implementation in code.

Hope this article gave you a good idea of what techniques to use to combat certain problems resulting from limited and noisy data, how exactly they work, and what the underlying mathematics behind them are. Have fun programming.