ML DL: remedial classes

Antoine Lavault[1,2]

[1]Apeira Technologies

[2]UMR CNRS 9912 STMS, IRCAM, Sorbonne Université

January 19, 2024

Contents I

Contents II

Contents

Outline

Regression - Definition

Definition (Regression)

Regression (in machine learning) is a predictive modeling technique that involves the estimation of a continuous outcome variable (dependent variable) based on one or more predictor variables (independent variables).
The goal is to find the relationship between variables and enable the prediction of future observations.

Remark

There are several types of regression techniques available to make predictions. These techniques are driven mainly by three metrics: the number of independent variables, the type of dependent variables, and the shape of the regression line.

Regression - Bestiary

Here are some of the most common regression models:

- Linear Regression: It is the simplest form of regression that assumes a linear relationship between the independent variables and the dependent variable.
- Polynomial Regression: A form of regression analysis in which the relationship between the independent variable and the dependent variable is modeled as an nth-degree polynomial.
- Ridge Regression (L2 Regularization): Addresses some of the problems of Linear Regression by imposing a penalty on the size of coefficients.
- Lasso Regression (L1 Regularization): Similar to Ridge Regression, it can lead to zero coefficients (i.e., feature selection).
- Logistic Regression: Despite its name, logistic regression is a linear model for binary classification that can be extended to multiclass classification.

Outline

Loss function

Definition (Loss Function)

A loss function in machine learning is a method to measure how well a model's predictions match the actual outcomes. In other words, it quantifies the difference between the predicted values and the actual values and presents it in the form of a single real number (i.e., a scalar). The objective of the training process is to minimize this loss function, which means the model's predictions should be as close as possible to the true data.

The loss function is chosen based on the type of machine learning task at hand.

Least Square Estimator

A least squares estimator is a statistical method used to estimate the coefficients of a model by minimizing the sum of the squares of the differences between the observed values and the values predicted by the model. This method is widely used for fitting a statistical model to data, and it's widespread in linear regression. The least-squares method finds the optimal parameter values by minimizing the sum of squared residuals for a parametrized model $f(\cdot, \beta)$:

$$S = \sum_{i=1}^{n} r_i^2; \quad r_i = y_i - f(x_i, \beta).$$

In multiple linear regression with several independent variables, the least squares estimates are often computed using matrix operations. The vector of estimated coefficients $\hat{\beta}$ is found by:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

where X is the matrix of input features, and Y is the vector of observed values.
When the least squares method is used without regularization or constraints, it is often referred to as Ordinary Least Squares.
Note: a model estimated using OLS will indirectly minimize the MSE.

In practice!

It's notebook time!

Figure: The ESIREM 3rd-year students have already seen these. So, get ready! And glasses don't make you smart.

Outline

Regularization

Regularizers are used in machine learning to prevent overfitting and to impose specific desirable properties on the model, such as simplicity or robustness. Here are the primary reasons for using regularizers:

- Control Overfitting: Regularization techniques penalize the complexity of the model, thereby discouraging the learning of a model that fits the noise in the training data.
- Handle Multicollinearity: Regularization can reduce the variance of coefficient estimates in multicollinearity when independent variables are highly correlated.
- Improve Model Generalization: Regularizers help improve the model's generalization ability on unseen data by constraining the model's capacity.
- Feature Selection: Some regularizers, like the L1 regularizer used in Lasso Regression, can shrink the coefficients of less important features to precisely zero, effectively performing feature selection and allowing the model to focus on the most relevant features.
- Bias-Variance Trade-off: Regularization introduces bias into the model estimates, but it can significantly reduce variance, leading to better long-term predictions.
- Numerical Stability: Regularization techniques can improve the numerical stability of the optimization problems by ensuring that the optimization algorithms work on a well-scaled and well-conditioned problem.

Regularization - The Math

Well, you add a positive term to the loss to be minimized. Just ensure it does what it's supposed to do.

Regularization - Ridge Regression

The L2 penalty is used in Ridge Regression to prevent overfitting, a common problem in machine learning where a model performs well on the training data but poorly on unseen data. Overfitting often occurs when the model is too complex and captures noise in the training data as if it were a true underlying pattern.
Ridge Regression addresses this issue by adding a penalty to the loss function—a penalty proportional to the square of the magnitude of the coefficients (L2 norm). This discourages the model from fitting the noise in the training data, leading (hopefully) to a more generalized model. The primary reasons for using the L2 penalty are:

1. Shrinkage of Coefficients: By penalizing the sum of the squares of the coefficients, Ridge Regression shrinks them towards zero, but not exactly zero. This means that while the influence of less important features is reduced, they are still part of the model, allowing for a small contribution to the prediction.
2. Reduction of Model Complexity: The L2 penalty discourages the fitting of a model with an excessively complex structure, which is characterized by a large number of parameters or by large values of parameters, thereby simplifying the model.

Here's how the L2 penalty is applied in practice in the case of the Ridge Regression:
The cost function for Ridge Regression is the sum of the squared residuals (as in ordinary least squares) plus the L2 penalty term. Mathematically, the cost function J can be expressed as:

$$J(\theta) = \text{MSE}(\theta) + \lambda \sum_{j=1}^{n} \theta_j^2$$

Where:

- $\text{MSE}(\theta)$ is the mean squared error of the predictions.
- θ represents the coefficient vector.
- λ is the regularization parameter that controls the strength of the penalty. A value of $\lambda = 0$ corresponds to ordinary least squares regression. As λ increases, the impact of the penalty increases, and the coefficients become smaller.

The L2 penalty term is $\lambda \sum_{j=1}^{n} \theta_j^2$, which is the sum of the squares of the coefficient values.
The regularization parameter λ is chosen through cross-validation, a process in which the model is trained and validated on different subsets of the data to find the value that yields the best model performance.

It's notebook time!

Figure: An average professor when doing anything.

Outline

Definition (Classification)

A classification task in machine learning is supervised learning, where the goal is to predict the categorical class labels of new instances based on past observations. These class labels are discrete, finite, and typically unordered.

Remark

Instead of producing a simple numerical index, most classification models in machine learning focus on estimating the likelihood that a given input corresponds to a specific class or category.

Outline

Definition (Logistic Regression)

Logistic Regression is a statistical method for predicting binary outcomes based on one or more predictor variables (features). It estimates the probability that a given input point belongs to a certain class.

- Binary Outcome: Logistic regression models the probability of the default class (e.g., "success" or "failure").
- Sigmoid Function: It uses the logistic sigmoid function to return a probability value. A threshold value, typically 0.5, is used to determine the class assignment

The probability of the default class '1' can be written as:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n)}}$$

Where: $P(Y = 1|X)$ is the probability of the class label being 1 given the features X, $\beta_0, \beta_1, \ldots, \beta_n$ are the coefficients, $X_1, \ldots, X_n$ are the feature variables, e is the base of the natural logarithm.
The coefficients are estimated using Maximum Likelihood Estimation (MLE), which aims to find the parameter values that make the observed data most probable.

The Logistic regression is trained using Maximum Likelihood Estimation (MLE). The goal of MLE is to find the set of parameters (coefficients) that make the observed data most probable. Here's how the training process typically works:

1. Initialization: Start with initial guesses for the parameters (coefficients) of the model.
2. Model the Probability: Logistic regression models the probability that an instance X belongs to the positive class $Y = 1$ using the logistic function (also known as the sigmoid function), which maps any real-valued number into the range (0, 1):

$$P(Y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n)}} = \sigma(\beta X + b) = h_\beta(X)$$

3. Maximize the Log-Likelihood: The parameters are estimated by finding the values that maximize the log-likelihood function ℓ. This is typically done using optimization algorithms like Gradient Descent or more sophisticated methods like Newton-Raphson or quasi-Newton methods like BFGS.

$$\ell(\beta) = \sum_{i=1}^{m} [y_i \log(P(y_i|x_i; \beta)) + (1 - y_i) \log(1 - P(y_i|x_i; \beta))]$$

- Iterative Optimization: The optimization algorithms iteratively adjust the parameters to find the maximum log-likelihood. They do this by computing the log-likelihood gradient for each parameter and updating the parameters in the direction that increases the log-likelihood.
- Convergence: The process is repeated until the algorithm converges, meaning that the change in the log-likelihood between iterations is below a predetermined threshold or a maximum number of iterations is reached.

The output of the training process is a set of coefficients that best fit the model to the data, maximizing the likelihood of the observed data under the model. These coefficients can then be used to make predictions on new data. The probability output by the logistic model for a given input can be converted to a binary category by selecting a threshold value (commonly 0.5) such that values above the threshold are classified as the positive class and values below as the negative class.

A Gradient-descent step

As seen in the previous slides, a gradient step is used to update the parameters in a way that minimizes the cost function. The cost function for logistic regression is typically the negative log-likelihood, also known as the logistic loss or binary cross-entropy loss (reminder: we want to maximize the log-likelihood, hence minimizing the negative log-likelihood).
For a single gradient step, the following occurs:

- Compute the Gradient: The gradient of the cost function for each parameter β_j is computed. This involves calculating the partial derivative of the cost function for each parameter.
  The gradient for parameter β_j is given by:

$$\frac{\partial}{\partial \beta_j} \text{Cost} = \frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

  Where:
  - m is the number of training examples,
  - $h_\beta(x^{(i)})$ is the hypothesis function for input $x^{(i)}$, which is the predicted probability that $Y = 1$ for the i-th example,
  - $y^{(i)}$ is the actual label for the i-th example,
  - $x_j^{(i)}$ is the value of feature j for the i-th example.

- Update the Parameters: The parameters are updated by stepping in the opposite direction of the gradient. The step size is determined by the learning rate α.
  The update rule for parameter β_j is:

$$\beta_j := \beta_j - \alpha \frac{\partial}{\partial \beta_j} \text{Cost}$$

- Repeat Until Convergence: This gradient step is repeated iteratively for all parameters until the cost function converges to a minimum or until another stopping criterion is met, such as a maximum number of iterations or a minimum change threshold in the cost function.
  The learning rate α is a hyperparameter that controls how big a step is taken.
  - If α is too large, the algorithm may overshoot the minimum or diverge.
  - If α is too small, the algorithm will converge slowly.

  In summary, a gradient step in logistic regression involves computing the gradient of the cost function with respect to the parameters and then updating the parameters in the direction that reduces the cost. This process leverages the data and labels from the training set to improve the model's parameters iteratively.

It's notebook time!

Figure: Friendly reminder. But I'm not even sure you are all younger than me...

Outline

Overfitting and underfitting

Overfitting and underfitting are concepts related to the performance of a machine learning model:

Definition (Overfitting)

Overfitting occurs when a model learns the training data too well, including its noise and outliers, and ends up with a complex model with high variance and low bias. This results in good performance on the training data but poor generalization to new, unseen data. Overfitting is like memorizing the answers to specific questions on a test rather than understanding the subject broadly.

Definition (Underfitting)

Underfitting happens when a model is too simple, characterized by having high bias and low variance, and is unable to capture the underlying pattern of the data. This results in poor performance on both the training data and unseen data. Underfitting is akin to not studying enough for a test and not understanding the subject matter sufficiently to answer the questions correctly.

In practice!

It's notebook time!

Figure: TL; DR of the section: overfitting hides ugly things. Especially to clients.

Contents

Definition (Perceptron)

A perceptron is a type of classifier used in machine learning and the simplest form of a neural network. The fundamental operation of a perceptron can be defined with the following equations:

- Linear Combination: $z = w^T x + b$
  Where:
  - w is a vector of weights.
  - x is a vector of input values.
  - b is the bias.
- Activation Function: The final output $\hat{y}$ is $\hat{y} = f(z)$
  Where f is an activation function that maps the input z to the output $\hat{y}$.
  For the classical perceptron, this is typically a step function that outputs either 1 or 0:

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

The weights and bias in the perceptron are adjusted during the training process. The training aims to find the correct values of weights and biases that will cause the perceptron to output the correct prediction for inputs.

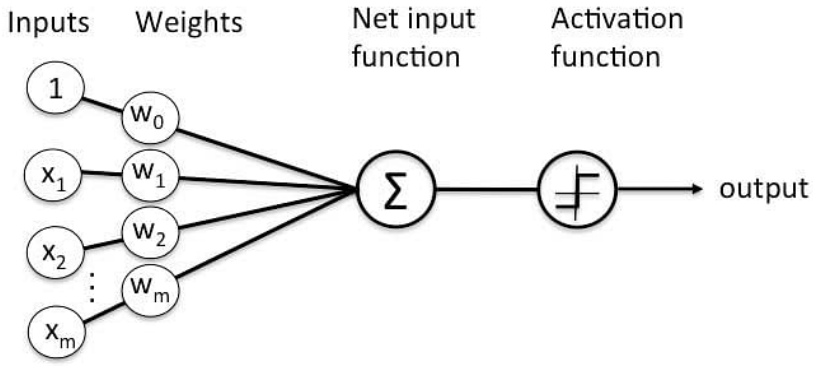

Figure: Illustration of a perception

Definition (Multi-Layer Perceptron)

An MLP, or Multilayer Perceptron, extends the concept of the simple perceptron to create a more robust model capable of solving complex problems, including non-linear classification and regression. An MLP consists of multiple layers of neurons (or perceptrons) organized into three types of layers:

- Input Layer: This is where the model takes its input data. Each neuron in this layer represents one of the input features.
- Hidden Layers: One or more layers of neurons that apply transformations to the inputs. Each neuron in a hidden layer receives a weighted sum of outputs from the previous layer, uses an activation function, and passes the result to the next layer.
- Output Layer: The final layer where the model outputs its predictions. In classification tasks, this layer often has one neuron for each class and uses a softmax activation function for multi-class problems or a sigmoid function for binary classification.

MLP illustrated

This is what Google says about "MLP illustrated":

Figure: MLP Illustrated, according to Google

MLP illustrated

What really is an MLP:

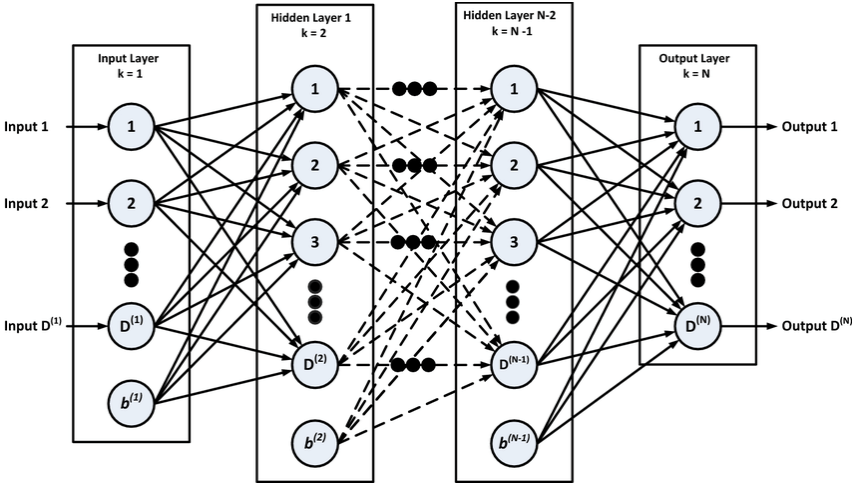

Figure: Multi-layer perceptron

We need to go deeper! Wink, wink.

A "deep" MLP has many layers (depth), while a "wide" MLP has many neurons in each layer (width). Most mathematical results are proven for infinitely wide networks, not infinitely deep ones. However, deep networks are preferred in practice for a variety of reasons:

- Hierarchical Feature Learning: Deep networks can learn a hierarchy of features. Lower layers might detect simple patterns, while deeper layers can learn to recognize more abstract concepts.
- Representation Power: With the same number of parameters, a deep network can represent more complex decision boundaries, making them suitable for problems with high complexity.
- Parameter Efficiency: Deeper architectures can be more parameter-efficient. They can achieve higher performance levels with fewer neurons than a wide, shallow network, which may require exponentially more neurons to achieve similar performance.
- Improved Generalization: Deep networks have the potential to generalize better to new samples, especially on tasks that benefit from high-level feature extraction. This is because they can abstract features at multiple levels.
- Success in Practice: Deep learning has achieved state-of-the-art results in many domains, such as computer vision and natural language processing, encouraging a preference for deeper architectures.

Notebook time!

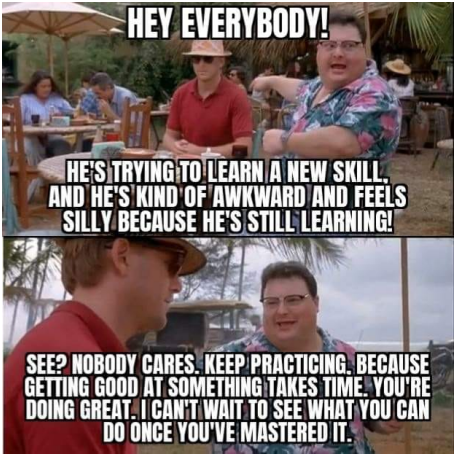

Figure: I can be wholesome sometimes ;)

Outline

Q: Why do we use activation functions?

Activation functions in (Multilayer) Perceptrons play a crucial role in neural networks, with the most important being:

- Non-Linearity of real data: MLPs are designed to approximate any function (the universal approximation theorem). However, without a non-linear activation function in between the layers, no matter how many layers the network has, it would still behave like a single-layer perceptron because the composition of two linear functions is linear. Non-linear activation functions allow the network to capture complex patterns in the data.
- Introducing Dynamics: Activation functions help to decide whether a neuron should be activated or not. This is important for the model to adapt its weights as it learns, and the decision-making capability is vital for tasks like classification, where the output is a non-linear decision boundary.

Choosing an activation function

There are two classes of activations functions:

- Sigmoid-like: the sigmoid, tanh, softmax
- ReLU-like: ReLU, Leaky ReLU, Parametric Leaky, ELU

Sigmoids are an "old" deep-learning style and are only used when dealing with things that need to be constrained, like the output of a GAN-discriminator. Due to being bounded functions, they tend to exhibit gradient vanishing when values are large in absolute value.
The Rectified Linear Unit, first introduced in the late '60s, is said to replicate better the behavior of biological neurons than a sigmoid. In modern-day deep learning, they showed impressive performances. Compared to the sigmoid, they are not subject to a vanishing gradient, even if the ReLU can get stuck with negative values. This is why the Leaky ReLU was introduced, and it showed even better results.
It has been proven that ReLU-based MLPs are also universal approximators, just like the sigmoid-based ones.
TL;DR: Leaky ReLU=good.

Outline

Convolutional layers

The defining element of CNNs is the convolutional layer, which applies a convolution operation to the input, passing the result to the next layer. This operation captures the spatial and temporal dependencies in an image by applying relevant filters. The network can thus preserve the relationship between pixels by learning image features using small squares of input data.
A CNN transforms the input data layer by layer from the original pixel values to the final class scores. Unlike an MLP, the layers of a CNN have neurons arranged in 3 dimensions: width, height, and depth. Different layers may transform this 3D input volume to a 3D output volume of neuron activations.
The actual learning takes place during the training process, where the network uses backpropagation to adjust its weights and biases to minimize the loss function, improving its prediction accuracy over time.

Outline

LeNet

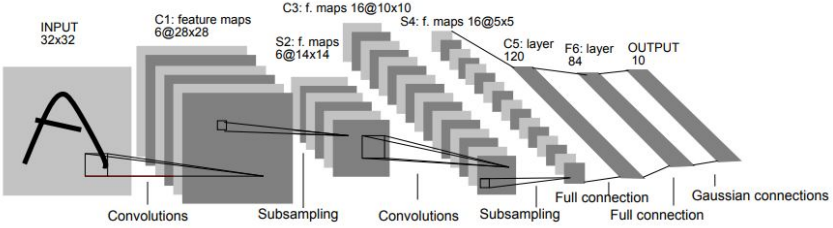

Figure: LeNet, the first CNN

Convolution Layers

In the LeNet architecture, convolutional layers serve several key functions:

- Feature Detection: These layers learn to detect features within the input images. Initially, they might detect simple features such as edges or corners.
- Local Receptive Fields: Each neuron in a convolutional layer is connected to only a small input region, known as the local receptive field, allowing the network to focus on local features in the first layers. This allows the network to detect the same feature across different parts of the input image, providing translation invariance.
- Creation of Feature Maps: The shared weights are applied across the entire input space to produce feature maps. Each map represents the presence and location of a particular feature detected across the input image.
- Activation Function: After the convolution operation, an activation function is applied to introduce non-linearity into the model, allowing it to learn more complex patterns.
- Hierarchical Feature Learning: As the input moves deeper into the network through successive convolutional layers, the architecture allows for detecting more complex and abstract features, building a hierarchy from simple to complex.

The convolutional layers in LeNet are fundamental for their ability to perform image recognition tasks, as they transform the raw pixel data into a form that can be used for classification by the fully connected layers at the end of the network.

Pooling Layers

Pooling, particularly in the context of convolutional neural networks (CNNs), is used for several reasons:

- Dimensionality Reduction: Pooling reduces the spatial size (i.e., width and height) of the input volume for the next convolutional layer. This decreases the number of parameters and computations in the network, thereby reducing the computational cost.
- Feature Down-Sampling: By providing a compressed form of the representation, pooling layers reduce the sensitivity of the output to small changes, noises, and distortions in the input. This helps to prevent overfitting, where the model performs well on training data but poorly on unseen data.
- Learning Hierarchical Representations: In deep networks, pooling allows the model to learn hierarchical representations by creating spatial hierarchies. As we go deeper into the network, pooling helps the network recognize patterns and objects at multiple scales and complexities.

Pooling is applied to each feature map independently, ensuring that the reduction of spatial dimensions does not affect the depth dimension (the number of feature maps).
Pooling layers, particularly max pooling and average pooling, are a staple in CNN architectures and contribute significantly to the network's ability to generalize from visual data.

Fully-Connected Layers

In the LeNet architecture, fully connected layers play an important role towards the end of the network:

- High-level Reasoning: Fully connected layers at the end of a neural network process complex features identified by earlier convolutional and pooling layers to perform high-level reasoning, allowing the network to make decisions based on the entire image, not just local features.
- Classification: Fully connected layers are typically used to map the reduced and abstracted representation of the input (from the convolutional and pooling layers) to the final output categories. In the case of LeNet, which was initially designed for MNIST, the fully connected layers are responsible for determining which digit the input image most likely represents.
- Learning Non-Spatial Hierarchies: They consider all neurons from the previous layer as a single vector, allowing the network to learn non-spatial and global hierarchies of features.
- Versatility for Different Outputs: They are adaptable to various outputs, not just classification. For example, they can be modified for regression tasks, multi-label classification, or even output a new image.

In summary, fully connected layers in LeNet-5 and similar architectures serve as a final decision-making body that takes all the complex features extracted by the convolutional and pooling layers and uses them to classify the input into appropriate categories.

In PyTorch

```
class LeNet(nn.Module):
    """LeNet-5 model."""
    def __init__(self,num_classes=10):
        super().__init__()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

Backpropagation

Backpropagation, short for "backward propagation of errors," is a fundamental algorithm to train artificial neural networks. It is a supervised learning algorithm that uses a method known as gradient descent to optimize the network weights. Here's a definition in parts:

- Error Calculation: It starts by computing the error between the predicted output and the actual output after a forward pass through the network.
- Gradient Computation: The algorithm then calculates the gradient of the error with respect to each weight in the network by applying the chain rule for derivatives, effectively determining how much each weight contributed to the error.
- Backward Pass: This gradient is then propagated backward through the network, from the output layer through each hidden layer down to the input layer, hence the term "backpropagation."
- Weight Update: The computed gradients are used to update the weights in the opposite direction of the gradient (since we're minimizing the error) by a small step scaled by a learning rate.
- Iterative Process: The process is repeated, with the forward pass and backward pass being executed iteratively over many epochs or passes through the entire training dataset until the network performs satisfactorily.

Backpropagation is the cornerstone of learning in multilayered neural networks, enabling them to learn from their mistakes and improve their predictions. It is widely used in deep learning models that require training on large datasets.

Why do we use it?

The big advantage of back-propagation is its scalability. It can scale to networks with many parameters and layers, which is essential for tasks that require the processing of high-dimensional data like images, videos, and audio.

Inside of a ConvNet!

Notebook time!

Figure: Or not wholesome at all...

Contents

Outline

We promised at the start of this tutorial we'd explain through examples each of `torch.nn`, `torch.optim`, `Dataset`, and `DataLoader`.

Let's summarize the most important parts of PyTorch:

- `torch.nn`:
  - `Module`: An object similar to a function in its callability but also holds state, such as the weights of layers in a neural network. It is aware of its `Parameter`(s), capable of resetting their gradients to zero and iterating over them for training updates.
  - `Parameter`: Acts as a signal to a `Module` that it possesses tensors which are weights requiring adjustment during backpropagation. This update process is reserved for tensors designated with the `requires_grad` attribute.
  - `functional as F`: A submodule that offers a suite of functions such as activation and loss functions, as well as stateless versions of certain layers, including convolutional and linear types.
- `torch.optim`: Encompasses a range of optimizers, like SGD, responsible for updating the weights of `Parameter` objects throughout the backpropagation process.
- `Dataset`: An abstract class that represents collections that are indexable and have a defined length (i.e., `__getitem__` and `__len__`), akin to lists and NumPy arrays.
- `DataLoader`: Constructs an iterator from any given `Dataset`, which then yields batches of data.

Outline

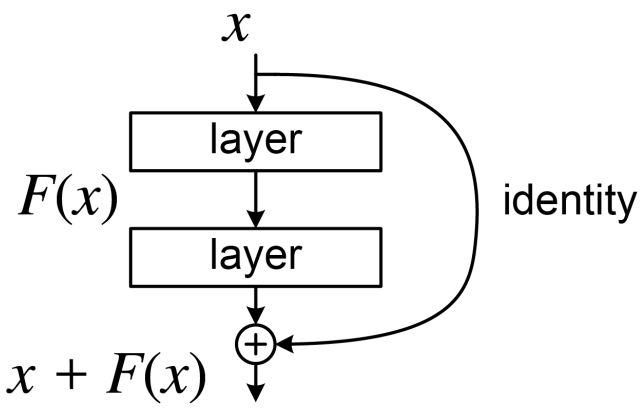

Figure: ResNet Block

In PyTorch

A good idea would be to create a Module that encapsulates everything.

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None, activation=nn.ReLU()):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels, out_channels, kernel_size=3, stride=stride, padding=1
            ),
            nn.BatchNorm2d(out_channels),
            activation,
        )
        self.conv2 = nn.Sequential(nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1
            nn.BatchNorm2d(out_channels),)
        self.downsample = downsample
        self.activation = activation
        self.out_channels = out_channels

    def forward(self, x):
        residual = self.downsample(x) if self.downsample else x
        out = self.conv1(x)
        out = self.conv2(out)
        out += residual
        out = self.activation(out)
        return out
```

Practice time?

No, not yet. We will reuse the ResNet in a "longer" example on CIFAR-100.

Definition (Normalization Layer)

A normalization layer in deep learning is defined as a component within a neural network that standardizes the distribution of layer inputs or activations across specific dimensions (such as batch, channels, or groups of channels) to have a mean of 0 and a standard deviation of 1.

Remark (Learnable Parameters)

Note that normalization layers in deep learning (often) have learnable parameters to maintain the representational capacity of the network:

- a scale parameter (usually denoted as γ),
- a shift parameter (usually denoted as β)

The reasons for including these parameters are to adjust the standardized inputs to an optimal scale and mean, preserve the model's ability to represent complex functions, and do a form of feature selection.

Why using normalization layers?

Normalization layers in deep learning stabilize and accelerate the training of neural networks. Here's why:

- Consistency: Normalization ensures that the distribution of the inputs to layers deep in the network remains more stable during training. This stabilizes the learning process (i.e., avoid vanishing or exploding gradients)
- Faster Learning: It can allow for higher learning rates because normalization would ensure that the backpropagated gradients do not explode or vanish.
- Regularization: Some normalization techniques also have slight regularizing effects, reducing the need for other regularization techniques (like weight decay or weight clipping)

The difference between the different normalization layers lies in how they compute the mean and variance used to normalize the data and the dimensions over which they are applied:

- Batch Normalization: Computes the mean and variance for normalization over the mini-batch. It works well for medium-sized datasets but can behave inconsistently for small batch sizes or with recurrent neural networks. Normalization is done for each feature channel independently.
- Instance Normalization: Computes the mean and variance for each individual instance (image) in the batch independently. It normalizes each sample separately rather than across a batch, making it useful for style transfer tasks in computer vision. It treats each instance of data as if it were its own batch.
- Group Normalization: It divides the channels into groups and computes the mean and variance for normalization within each group. It doesn't depend on the batch size, which makes it suitable for tasks with small batch sizes and recurrent neural networks. It is more stable than batch normalization when training with small batch sizes. Layer normalization is a particular case where there is only one group. Instance Normalization is when the number of groups equals the number of channels.

In color!

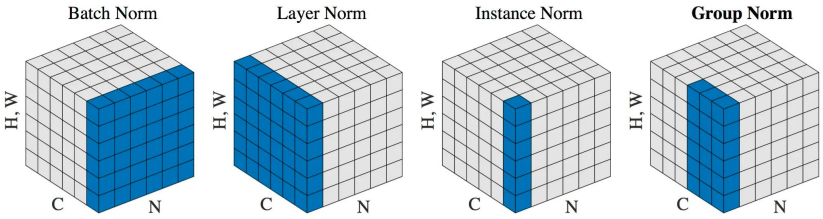

Figure: Illustrations of classic normalization techniques

In this example, we will build a GAN (inspired by DCGAN, Deep Convolutional GAN)
What we need in terms of networks are:

- A generator, i.e., a network that maps a latent vector to the data of interest
- A discriminator, i.e., a network that maps data of interest to the probability of being in the training set.

The big "problem" is the training algorithm...

The Generator

We will use transposed convolutions to do the upsampling here.

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            #...
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh() # clamps the output in [-1,1]
            # state size. ``(nc) x 64 x 64``
        )

    def forward(self, input):
        return self.main(input)
```

Discriminator

In most GAN architecture, the discriminator architecture mirrors the one from the generator.

```
class Discriminator(nn.Module):
    def __init__(self,):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # ...
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid() # To output between 0 and 1
        )
    def forward(self, input):
        return self.main(input)
```

Hacky loss function

The objective function for a GAN, often referred to as the min-max loss, is given by the following equation:

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

This looks a lot like the Binary Cross-Entropy Loss (`nn.BCELoss`)
And this is how it is defined in PyTorch:

$$\ell(x, y) = L = \{l_1, , \cdots, l_N\}^T, \quad l_n = -w_n[y_n \log x_n + (1 - y_n) \log(1 - x_n)]$$

Gradient descent is always made to minimize, so we must be clever.

Hacky loss cont.

Notice how the Binary Cross-Entropy provides the calculation of both log components in the objective function (i.e., $\log(D(x))$ and $log(1 - D(G(z)))$. We can specify what part of the BCE equation to use with the y input. This will be accomplished in the training loop soon, but it is important to understand how to choose which component we wish to calculate by changing y (i.e., the ground truth labels).
Next, we define our real label as 1 and the fake label as 0. These labels will be used when calculating the losses of D and G, which is also the convention used in the original GAN paper. Finally, we set up two separate optimizers, one for D and one for G, since we have two optimization problems (a minimization and a maximization in the original loss).
We will generate a fixed batch of latent vectors drawn from a Gaussian distribution to keep track of the generator's learning progression. In the training loop, we will periodically input this fixed noise into G, and over the iterations, we will see images form out of the noise. This is a good practice to identify convergence problems.
As specified in the DCGAN paper, both are Adam optimizers with a learning rate of 0.0002 and $\beta_1 = 0.5$.

Training the Discriminator

Training the discriminator aims to maximize the probability of correctly classifying a given input as real or fake.
Practically, we want to maximize $\log(D(x)) + \log(1 - D(G(z)))$.
First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss, and then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss, and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

In code

```
## Train with all-real batch
netD.zero_grad()
# Format batch
real_cpu = data[0].to(device)
b_size = real_cpu.size(0)
label = torch.full((b_size,), real_label, dtype=torch.float, device=device) # =(1,1,1,1)
# Forward pass real batch through D
output = netD(real_cpu).view(-1)
# Calculate loss on all-real batch
errD_real = criterion(output, label) # Because the label is real_label=1, it is equal to -log(D(x))
# Calculate gradients for D in the backward pass
errD_real.backward()
D_x = output.mean().item()
## Train with all-fake batch
# Generate a batch of latent vectors
noise = torch.randn(b_size, nz, 1, 1, device=device)
# Generate fake image batch with G
fake = netG(noise)
label.fill_(fake_label)
# Classify all fake batches with D
output = netD(fake.detach()).view(-1)
# Calculate D's loss on the all-fake batch
errD_fake = criterion(output, label)
# Calculate the gradients for this batch, accumulated (summed) with previous gradients
errD_fake.backward()
D_G_z1 = output.mean().item()
# Compute the error of D as the sum over the fake and the real batches
errD = errD_real + errD_fake
# Update D
optimizerD.step()
```

Generator Training

We will use a similar "hack". If we set the labels to 1, the criterion will be $-\log(D(G(z))$, which is precisely what we want to minimize.

```
netG.zero_grad()
label.fill_(real_label)  # fake labels are real for generator cost
# Since we just updated D, perform another forward pass of the all-fake batch through D
output = netD(fake).view(-1)
# Calculate G's loss based on this output
errG = criterion(output, label)
# Calculate gradients for G
errG.backward()
D_G_z2 = output.mean().item()
# Update G
optimizerG.step()
```

(Soon To Be?) Dr. Lavault Magic Tricks

- I *personnaly* don't like activation functions at the end of my networks. To mitigate this, I would prefer to offload weird calculations to the loss functions, not the network.
- In my experience, adding a penalty term to any overshoot by the discriminator (in other words, a penalty if the values are not within specification) greatly helps with the stability and the dynamic range. For waveforms, since my GAN tended to output values that were too big, it made it go poof.
- Always have a form of "reference latent" to check if your network has converged and to check any mode collapse (i.e., outputting a subset of the training data only).
- In terms of control, if you use features from some descriptor, try to reimplement the descriptor to make it fit in the training. This will give more meaningful gradients to the network! And in general, if you know things should have a particular characteristic, try to use penalties to constrain it softly.

LOL

Figure: You will understand by the end of the section

Weight Initialization

During tutorials, we have seen that random parameters are "better" for initializing layers, but we need to choose the parameters wisely.
All in all, you should go with:

- Sigmoid-like activation: Xavier Initialization
- ReLU-like activation: He/Kaiming Initialization

Choosing between uniform and normal is not generally the question. Unless you have some insights on your application or the paper you are trying to reproduce uses a particular distribution.

Contents

Meet the data!

Yeah! Numbers....

Practice time!

Create a network that solves the MNIST problem in less than 30 minutes.
Students whose model has the best accuracy on the test set will have the right to brag to the others.
Notebook time!

Figure: Who has the reference?

Meet the data!

Challenges

The CIFAR-100 dataset is considered harder to work with than the MNIST dataset for several reasons beyond just resolution:

- Complexity of Images: CIFAR-100 images are RGB color images with more complex structures and patterns compared to MNIST's grayscale images of handwritten digits. Color images generally require learning more features to distinguish between different objects.
- Variety of Classes: CIFAR-100 has 100 different classes, compared to 10 for MNIST. More chances to get it wrong!
- Intra-class Variation: Within each class, CIFAR-100 has a larger variation in the appearance of objects due to different angles, lighting conditions, and occlusions. MNIST digits are relatively normalized in size and centered in the frame.
- Inter-class Similarity: Some classes in CIFAR-100 are very similar to each other, requiring the model to learn fine-grained differences to tell them apart, unlike the more distinct digits in MNIST.
- Image Quality: Although both datasets have low-resolution images, the details required to differentiate between CIFAR-100's classes are more subtle and can be easily lost due to the low resolution.
- Background Noise: CIFAR-100 images often contain background details or noise not present in MNIST images, which are clean and contain only the digit to be classified.
- Real-world Complexity: The objects in CIFAR-100 are real-world objects with a higher degree of variability compared to the relatively simple and consistent shapes of MNIST digits.

But... What does it mean?

We. Need. To. Go. Deeper.
Enter ResNet (well, the original was trained on ImageNet... Oops).

Or trying to fit a square peg into a round hole. But it's an interesting thing nonetheless.

Definition (Transfer Learning)

Transfer learning is a machine learning technique where a model developed for a particular task is reused as the starting point for a model on a second task. It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks given the vast computing and time resources required to develop neural network models on these problems and from the huge jumps in skill they provide on related problems.

Example 1 - Reusing the architectures minus a few details

Here, we change the network's last layer to match the new number of classes.

```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, n_class)
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)
```

Example 2 - Using the pre-trained model as a feature extractor

```
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')
# Making the network "untrainable"
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default,
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, n_class)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()
```

Time to get your hands dirty!

Notebook time!

Figure: You expected a meme, but it's me: Pelvoux!

Outline

Meet the data!

ENST-Drums (Gillet and Richard 2006):

- Most of the dataset is not interesting: we want one-shot samples, not loops.
- Augmentation will be necessary.

Result: Lavault, Roebel, and Voiry 2022

Figure: Short Version

Long Version

The idea of StyleWaveGAN was to make a waveform generator, but waveforms are inherently 1D.
We could have used spectral representation (STFT) to allow us to reuse the original network, but there is a catch: STFT is inefficient in terms of output size...

- GANSynth (Engel et al. 2019): final resolution 1024x512, for 4s at 16kHz = 64000 samples
- SWG (Lavault, Roebel, and Voiry 2022): final resolution is final output = 65536

So less is more.
In addition, there was AutoFade and batch balancing (first for a generation task?) and differentiable perceptual controls.
Exercise: Read Tero Karras et al. "Analyzing and improving the image quality of stylegan". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 8110–8119 and Antoine Lavault, Axel Roebel, and Matthieu Voiry. "StyleWaveGAN: Style-based Synthesis of Drum Sounds with Extensive Controls using Generative Adversarial Networks". In: *19th Sound and Music Computing Conference (SMC 2022)*. Saint-Etienne, France, June 2022. URL: https://hal.archives-ouvertes.fr/hal-03693950. What are the key differences, if any?
Exercise: reimplement AutoFade in Torch.

Live at Wembley!

Demo Time! But in Tensorflow...

Figure: How I feel when I give a class. Minus the mustache.