1 Integer Representation

```
np.float32(6.5)/np.float64(6.5) # Cast to different precisions
np.spacing(1e25)
np.float64(2147483648.0)
np.nextafter(1e25, 1e26) - 1e25
bin(x) #Python function to get the binary
np.finfo(np.float64).minexp # minimum exponent value
np.finfo(np.float64).tiny # smallest possible positive value
(2.0**np.arange(-4, 4, 1)).sum() # Sums 2^-4, 2^-3, ..., 2^3
```

Unsigned Range (N bits): 0 to $2^N - 1$.

Signed Range (N bits): -2^{N-1} to $2^{N-1} - 1$ (using two's complement).

2 Floating-Point Numbers (IEEE 754)

2.1 Representation

- **Sign (S)**: 1 bit (0 for positive, 1 for negative).
- **Exponent (E)**: A biased exponent.
- **Mantissa (M)**: The fractional part of the number.

The value is: $(-1)^S \times (1 + M) \times 2^{E-\text{bias}}$

Single (32-bit): 1 sign, 8 exponent, 23 mantissa bits (bias 127).

Double (64-bit): 1 sign, 11 exponent, 52 mantissa bits (bias 1023).

2.2 Floating-Point Spacing (ULP)

The distance between two consecutive floating-point numbers is not constant (ULP - Unit in the Last Place). Staircase like. Gap between two representable floats. It's the value of the LSB of the mantissa.

Spacing: Increases with the magnitude of the number. For a number $x = M \times 2^E$, the spacing is $2^E \times \epsilon$.

`np.spacing(x)`: NumPy function that returns the distance between x and the next representable float.

As magnitude (↑), exponent (↑), spacing (↑), and relative precision (↓).

2.3 Numerical Stability and Order of Operations

Floating-point arithmetic is not associative. The order of operations matters and can lead to different results due to rounding errors. .e.g., `(0.1 + 0.2) + 0.3` is not always equal to `0.1 + (0.2 + 0.3)`. .This can affect the stability of complex calculations in ML algorithms. .If the added value is much smaller than the current number's ULP, it rounds back to the original number (e.g., `1e50 + 1e5 == 1e50` is True). .Order of operations matters: `0.7 + 0.7 + 1e16 == 1e16` is False, but `1e16 + 0.7 + 0.7 == 1e16` is True, because `1e16 + 0.7` rounds to `1e16` if `0.7` is less than half the ULP.

2.4 Example: Decimal to Float (Single Precision)-Remove

Let's convert **-0.75** to single precision.

1. **Sign (S)**: The number is negative, so $S = 1$.
2. **Binary Conversion**: $0.75 = 0.5 + 0.25 = 2^{-1} + 2^{-2} = 0.11_2$.
3. **Normalization**: Move the decimal point to get a number of the form $1.M$. $0.11_2 = 1.1_2 \times 2^{-1}$.
4. **Exponent (E)**: The exponent is -1. The bias is 127. So, the biased exponent is $E = -1 + 127 = 126$. In binary, $126 = 01111110_2$.
5. **Mantissa (M)**: The fractional part from normalization is 1. For single precision, we need 23 bits, so we pad with zeros: 10000000000000000000000.

2.5 Example: Float to Decimal (Single Precision)-Remove

Let's convert the float: `0 10000001 10100000000000000000000`

1. **Sign (S)**: $S = 0$, so the number is positive.
2. **Exponent (E)**: $E = 10000001_2 = 129$. The unbiased exponent is $129 - 127 = 2$.
3. **Mantissa (M)**: $M = 1010\ldots$. The implicit leading bit is 1, so the significand is $1.M = 1.101_2$.
4. **Calculation**: The value is $(1.101)_2 \times 2^2$. Shift the decimal point by 2 places: 110.1_2. Convert to decimal: $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} = 4 + 2 + 0 + 0.5 = 6.5$.

So, the number is **6.5**.

2.6 Relevance in ML/DL

The choice of floating-point precision is critical in Machine and Deep Learning.

- **Large-Scale Computation**: Deep learning models involve millions of parameters and vast amounts of numerical computations.
- **Error Accumulation**: Small rounding errors in floating-point math can accumulate through many layers, potentially leading to suboptimal model convergence.
- **Efficiency & Stability**: Lower precision (e.g., 16-bit floats) can speed up training and reduce memory usage, but may affect numerical stability. Higher precision is more accurate but slower. This trade-off is crucial in optimization.
- 32-bit floats (single precision) are faster and use less memory (Advantage), but have lower precision/accuracy (Disadvantage) compared to 64-bit (double precision).
- More exponent bits (e.g., 52-11 scheme) result in a larger range of representable numbers (closer to 0 and infinity) than more mantissa bits (e.g., 57-6 scheme).

3 Gradient Descent

3.1 Stopping Criteria for Gradient Descent

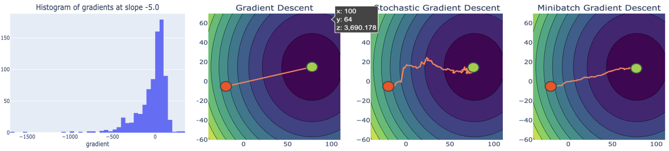
Gradient descent typically stops when one of the following criteria is met: .The step size (i.e., the change in parameters) falls below a predefined threshold. . A maximum number of iterations (steps) is completed.

3.2 Gradient Descent Pseudocode

- Gradients are essential for minimizing loss functions; they indicate direction and magnitude for parameter updates to decrease loss.
- The number of iterations for convergence depends on initial parameter values; closer to the minimum means fewer iterations.
- SGD loss can increase if the learning rate is too large or if the batch gradient does not accurately represent the true gradient.
- Doubling the learning rate can lead to longer/shorter optimization or divergence (inf/nan). The loss landscape itself does not vary.
- Different batch sizes result in different gradients and trajectories, so 1 epoch doesn't lead to the same point in the loss landscape.
- SGD's path is noisy because mini-batch gradients are approximations, causing random deviations from the true gradient.
- To avoid local minima, try initializing parameters with different values or using SGD with different batch sizes.
- Real-life loss functions of neural networks are rarely smooth-looking
- Recommended to scale features when using gradient descent.
- Feature scaling makes the loss function's contours more spherical rather than elongated ellipses. This disproportionate contribution of different features to the loss can result in slower convergence for gradient descent.
- Gradient Descent is only guaranteed to find the global minimum for convex functions. For non-convex functions (like those in deep learning), it can easily get stuck in a local minimum or a saddle point.

3.3 Stochastic Gradient Descent Pseudocode

```
• Initialize parameters $w$ with some arbitrary values.
• While stopping criteria not met:
    – Shuffle the training dataset to ensure random order.
    – For each training example $(\mathbf{x}_i, y_i)$ in the dataset:
        ∗ Calculate the gradient of the loss for that specific example.
        ∗ Update $w$ immediately using the rule: $w = w - \alpha \nabla \mathcal{L}_i(w)$.
• Repeat until stopping criteria is met.
```



The histogram is left-skewed, indicating that more often than not, our gradient is negative. This means that we need to increase W to decrease our loss. Notice we bounce around a bit. We get this "noise" because we are only basing our adjustment on one data point.f we were to traverse the overall loss landscape (the contour plot above), we should have smoothly progressed towards the minimum because the direction of the steepest descent does not change in the above case. However, the SGD method actually traverses a different loss landscape at each iteration, which is why the path towards the minimum looks noisy.

3.4 minibatch stochastic gradient descent

- Its between GD and SGD. Rather than calculating the gradient from just one random data point, calculate it based on a batch of data points.
- The larger the batch, the closer we are to the gradient calculated using the whole dataset
- But also the bigger the batch, the more computations will be needed
- **Approach 1:** Shuffle the dataset and pre-divide it into batches, like cross-validation. This is "without replacement", every example is used once.
- **Approach 2:** Pick a batch of certain size randomly from the dataset without replacement. In this case, you won't have the same example occurring more that once in a batch, but you might have the same example in both batch 1 and batch 2. Every example may not be used in this approach.

3.5 Batch, Epoch and Learning Rates

Assume a dataset of N observations (rows, samples, examples, data points, or points).

- **Iteration**: Each time you update model weights.
- **Batch**: A subset of data used in an iteration.
- **Epoch**: One full pass through the dataset to look at all N examples.
- **In GD**: 1 Iteration = 1 epoch, **In SGD**: n Iteration = 1 epoch, **In MGD**: (n/batch_size) Iteration = 1 epoch
- Use an "adaptive" learning rate ADAM optimizer- take big steps when far from the minimum of loss function, and smaller steps when we close to it;
- Use "momentum" learning rate - using how our weights have changed in past iterations to influence how it changes in future iterations

```
#for X_batch, y_batch in dataloader:
# optimizer.zero_grad()
# logits = model(X_batch)
# loss = criterion(logits, y_batch)
# loss.backward()
# optimizer.step()
```

1.Pytorch Training structure. 2.Get a mini-batch from the DataLoader 3.Clear old gradients 4.Forward pass: compute logits (raw scores) 5.Compute the loss 6.Backward pass: compute gradients 7.Update parameters

4 Neural Networks

- **Activation Functions Purpose:** Introduce non-linearity to model complex relationships in data.
- Activation functions should be non-linear and tend to be monotonic and continuously differentiable (smooth)
- **Output Layer (Regression):** Typically no activation is applied, as they limit the range of values and prevent continuous outputs.
- Number of weights in the first hidden layer depends on the input feature dimension, not batch size.
- **Parameter (weights and biases) calculation:** $(InputFeatures \times HiddenBiases + (HiddenNeurons \times OutputFeatures) + OutputBiases$.
- Example: 6 input, 2 output, 1 hidden layer with 5 neurons: $(6 \times 5) + 5 + (5 \times 2) + 2 = 47$ parameters.
- The 'forward()' method defines how input data flows through the network's layers (defined in '__init__') to produce an output.
- **Automatic Differentiation :** Can be used for both deep and shallow neural networks. PyTorch provides a .backward() method on every tensor that takes part in gradient computation. This enables us to do gradient descent without worrying about the structure of our model, since we no longer need to compute the derivative ourselves. Computes gradients only for tensors with 'requires_grad=True'.

4.1 Loss Functions

Now to train this, we would need to have a loss function, and try to minimize it, right? Rules of thumb regarding the training:

- **Predict a real number**: 1 output (usually the case, but not necessarily) and we use MSE for loss.
- **Binary classification**: 1 logit and we use BCEWithLogitsLoss for loss.
- **Multi-class (k classes)**: k logits + CrossEntropyLoss (PyTorch expects raw logits).

Other Common Loss Functions:

- **Mean Absolute Error (L1 Loss)**: $|y - \hat{y}|$. Less sensitive to outliers than MSE.
- **Huber Loss**: A combination of MSE and MAE. It's quadratic for small errors and linear for large errors.