Spring 2024

CMPE 258-01 Deep Learning

Dr. Kaikai Liu, Ph.D. Associate Professor

Department of Computer Engineering

San Jose State University

Email: kaikai.liu@sjsu.edu

Website: https://www.sjsu.edu/cmpe/faculty/tenure-line/kaikai-liu.php

Project Key Components

- Build a full pipeline of Deep Learning application with model training
  - Question/Problem Formulation: **propose** your own application and formulate the problem
  - Data Acquisition and Processing:
    - Identify a Suitable Dataset for Model Training and Evaluation
  - **State-of-the-Art Models**
    - Analyze the data, assess and compare various state-of-the-art open source models
  - **Model architecture change, Training, Evaluation, Fine-tunning**
    - Change the model architecture, tune parameters, and proceed with model training/fine-tunning, and evaluation. Gain the insights of the performance impact from the model architecture and parameters.
  - Inference, Optimization, and Real-time test
    - Create a comprehensive end-to-end deep learning application to enable inference using the trained model.
    - Depending on the chosen hardware platform, optimize model inference to enhance speed or reduce computational costs.
    - Perform evaluations and visualizations using real test data.

Common Datasets

- **NLP: Question-Answering, Semantic Search, Text Translation**
  - **SQuAD (Stanford Question Answering Dataset):** A dataset for reading comprehension tasks, where models answer questions based on a given passage.
  - Machine Translation (WMT19).
- **Audio: Speech Recognition, Translation**
  - **LibriSpeech:** A dataset of spoken words and sentences collected from audiobooks.
  - **Mozilla Common Voice:** A crowdsourced dataset of speech data for various languages.
- **Vision: Autonomous Driving: 2D/3D object detection, tracking, scene understanding**
  - **KITTI Vision Benchmark Suite** (https://www.cvlibs.net/datasets/kitti/)**:** A dataset for tasks like object detection, tracking, and scene understanding in autonomous driving scenarios.
  - Waymo open dataset: https://waymo.com/open/
  - Argo open dataset: https://www.argoverse.org/
  - NuScenes: https://www.nuscenes.org/

PyTorch

- PyTorch Introduction in https://github.com/lkk688/DeepDataMiningLearning
  - Pytorch Introductions, Tensors, and Autograd: colablink
  - Pytorch Regression and Logistic Regression: colablink
  - Pytorch Simple Neural Networks: colab

PyTorch

- Dynamic Neural Networks: Tape-Based Autograd
  - PyTorch has a unique way of building neural networks: using and replaying a tape recorder.
  - Most frameworks such as TensorFlow, Theano, Caffe, and CNTK have a static view of the world. One has to build a neural network and reuse the same structure again and again. Changing the way the network behaves means that one has to start from scratch.
  - With PyTorch, we use a technique called reverse-mode auto-differentiation, which allows you to change the way your network behaves arbitrarily with zero lag or overhead.

A graph is created on the fly


```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

While this technique is not unique to PyTorch, it's one of the fastest implementations of it to date.

Deep Learning Frameworks

- Tensorflow: https://www.tensorflow.org
  - It was developed by the Google Brain team for Google's internal use in research and production
  - As TensorFlow's market share among research papers was declining to the advantage of PyTorch, the TensorFlow Team announced a release of a new major version of the library in September 2019. TensorFlow 2.0 introduced many changes, the most significant being TensorFlow eager, which changed the automatic differentiation scheme from the static computational graph, to the "Define-by-Run" scheme originally made popular by Chainer and later PyTorch
  - Keras abstractions.
- Jax: Composable transformations of Python+NumPy programs: differentiate, vectorize, JIT to GPU/TPU, and more
  - https://github.com/google/jax
  - use Flax (https://github.com/google/flax) on top of JAX, which is a neural network library developed by Google. It contains many ready-to-use deep learning modules, layers, functions, and operations
  - Flax is a neural network library for JAX that is designed for flexibility.
- Apple MLX: An array framework for Apple silicon
  - https://github.com/ml-explore/mlx
  - MLX has a Python API that closely follows NumPy. MLX also has a fully featured C++ API, which closely mirrors the Python API. MLX has higher-level packages like mlx.nn and mlx.optimizers with APIs that closely follow PyTorch to simplify building more complex models.

TORCH.AUTOGRAD

- torch.autograd is PyTorch's automatic differentiation engine that powers neural network training.
- Training a NN happens in two steps:
  - Forward Propagation: In forward prop, the NN makes its best guess about the correct output. It runs the input data through each of its functions to make this guess.
  - Backward Propagation: In backprop, the NN adjusts its parameters proportionate to the error in its guess. It does this by traversing backwards from the output, collecting the derivatives of the error with respect to the parameters of the functions (gradients), and optimizing the parameters using gradient descent.
  - We use the model's prediction and the corresponding label to calculate the error (loss). The next step is to backpropagate this error through the network. Backward propagation is kicked off when we call **.backward()** on the error tensor. Autograd then calculates and stores the gradients for each model parameter in the parameter's **.grad** attribute.

```
loss = (prediction - labels).sum()
loss.backward() # backward pass
```

TORCH.AUTOGRAD

- Next, we load an optimizer, in this case SGD with a learning rate of 0.01 and momentum of 0.9. We register all the parameters of the model in the optimizer.

```
optim = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)
```

- Finally, we call .step() to initiate gradient descent. The optimizer adjusts each parameter by its gradient stored in .grad.

```
optim.step() #gradient descent
```

- If we create two tensors a and b with **requires_grad=True**. This signals to autograd that every operation on them should be tracked.

Computational Graph

- Conceptually, autograd keeps a record of data (tensors) & all executed operations (along with the resulting new tensors) in a directed acyclic graph (DAG) consisting of Function objects. In this DAG, **leaves are the input tensors**, **roots are the output tensors**. By tracing this graph **from roots to leaves**, you can automatically compute the gradients using the chain rule.
- In a forward pass, autograd does two things simultaneously:
  - run the requested operation to compute a resulting tensor, and
  - maintain the operation's gradient function in the DAG.
- The backward pass kicks off when .backward() is called on the DAG root. autograd then:
  - computes the gradients from each .grad_fn,
  - accumulates them in the respective tensor's .grad attribute, and
  - using the chain rule, propagates all the way to the leaf tensors.
- **DAGs are dynamic in PyTorch:** the graph is recreated from scratch; after each .backward() call, autograd starts populating a new graph. This is exactly what allows you to use control flow statements in your model; you can change the shape, size and operations at every iteration if needed.

Computational Graph

- torch.autograd tracks operations on all tensors which have their requires_grad flag set to True. For tensors that don't require gradients, setting this attribute to False excludes it from the gradient computation DAG.
- The output tensor of an operation will require gradients even if only a single input tensor has **requires_grad=True**.
- In a NN, parameters that don't compute gradients are usually called frozen parameters. It is useful to "freeze" part of your model if you know in advance that you won't need the gradients of those parameters (this offers some performance benefits by reducing autograd computations).
- In finetuning, we freeze most of the model and typically only modify the classifier layers to make predictions on new labels.
- The same exclusionary functionality is available as a context manager in **torch.no_grad()**

TORCH.AUTOGRAD

- TORCH.TENSOR.BACKWARD
  - Tensor.backward(gradient=None, retain_graph=None, create_graph=False, inputs=None)
  - https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html
  - Computes the gradient of current tensor wrt graph leaves.
  - The graph is differentiated using the chain rule. **If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient**, the function additionally requires specifying gradient. It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. self.
  - This function accumulates gradients in the leaves - you might need to zero .grad attributes or set them to None before calling it.

TORCH.AUTOGRAD

- torch.autograd is an engine for computing vector-Jacobian product.

In many cases, we have a scalar loss function, and we need to compute the gradient with respect to some parameters. However, there are cases when the output function is an arbitrary tensor. In this case, PyTorch allows you to compute so-called **Jacobian product**, and not the actual gradient.

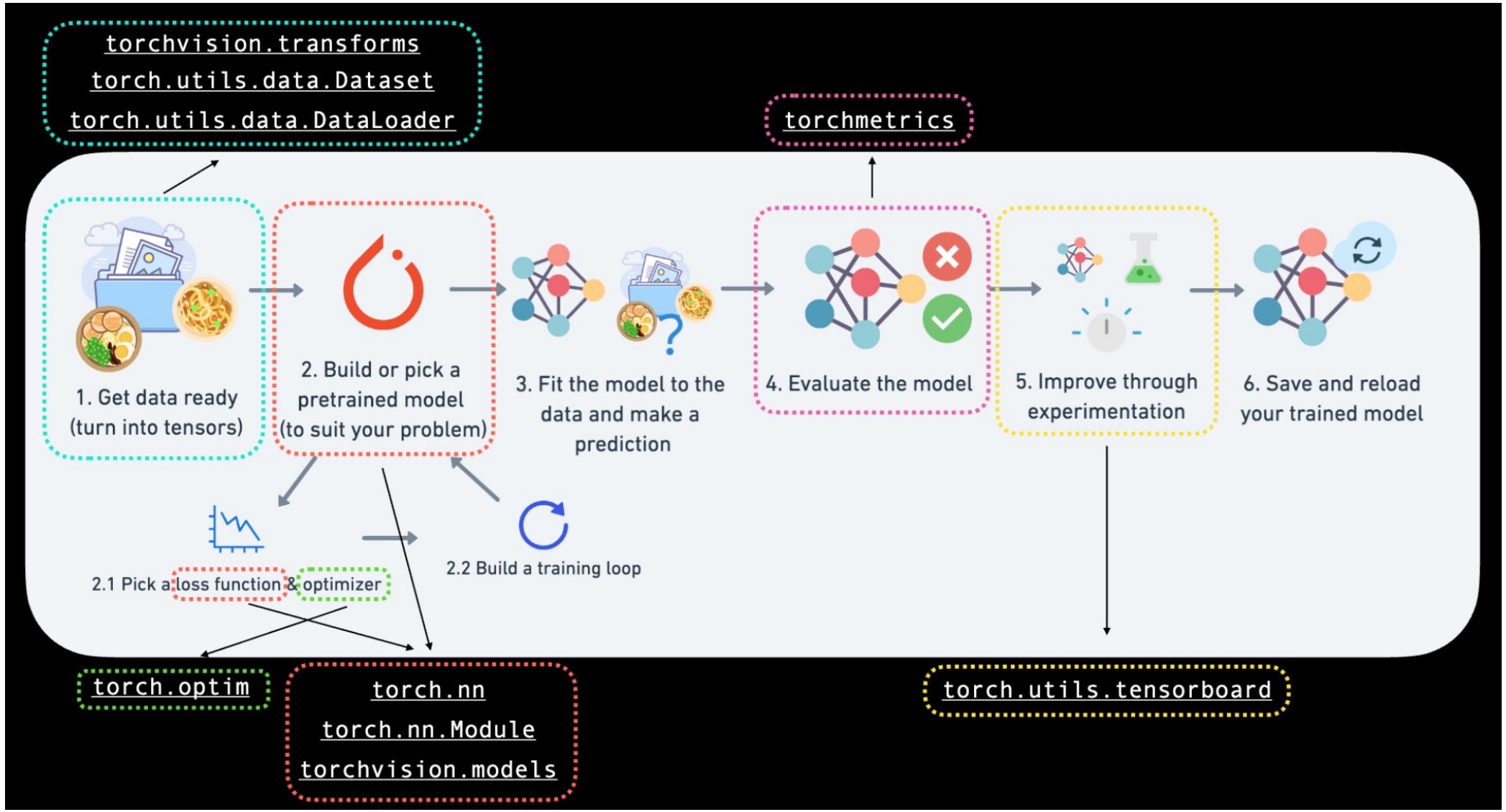
For a vector function $\vec{y} = f(\vec{x})$, where $\vec{x} = \langle x_1, \dots, x_n \rangle$ and $\vec{y} = \langle y_1, \dots, y_m \rangle$, a gradient of $\vec{y}$ with respect to $\vec{x}$ is given by **Jacobian matrix**:

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Instead of computing the Jacobian matrix itself, PyTorch allows you to compute **Jacobian Product** $v^T \cdot J$ for a given input vector $v = (v_1 \dots v_m)$. This is achieved by calling `backward` with v as an argument. The size of v should be the same as the size of the original tensor, with respect to which we want to compute the product:

if you pass v^T as the gradient argument, then y.backward(gradient) will give you not J but $v^T \cdot J$ as the result of x.grad

PyTorch Workflow



PyTorch models

- Architecture of a classification model
  - torch.nn.linear: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*)

Applies a linear transformation to the incoming data: $y = xA^T + b$.

Hyperparameter	Binary Classification	Multiclass classification
Input layer shape (`in_features`)	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
Neurons per hidden layer	Problem specific, generally 10 to 512	Same as binary classification
Output layer shape (`out_features`)	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
Hidden layer activation	Usually ReLU (rectified linear unit) but can be many others	Same as binary classification
Output activation	Sigmoid (`torch.sigmoid` in PyTorch)	Softmax (`torch.softmax` in PyTorch)
Loss function	Binary crossentropy (`torch.nn.BCELoss` in PyTorch)	Cross entropy (`torch.nn.CrossEntropyLoss` in PyTorch)
Optimizer	SGD (stochastic gradient descent), Adam (see `torch.optim` for more options)	Same as binary classification

PyTorch models

- Architecture of a classification model
  - torch.nn.linear:

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

```python
# Create a linear regression model in PyTorch
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()

        # Initialize model parameters
        self.weights = nn.Parameter(torch.randn(1,
            requires_grad=True,
            dtype=torch.float
        ))

        self.bias = nn.Parameter(torch.randn(1,
            requires_grad=True,
            dtype=torch.float
        ))

    # forward() defines the computation in the model
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.weights * x + self.bias
```

Linear regression model with `nn.Parameter`

```python
# Create a linear regression model in PyTorch with nn.Linear
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        # Use nn.Linear() for creating the model parameters
        self.linear_layer = nn.Linear(in_features=1,
                                      out_features=1)

    # forward() defines the computation in the model
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.linear_layer(x)
```

Linear regression model with `nn.Linear`

PyTorch Training Loop

```python
# Pass the data through the model for a number of epochs (e.g. 100)
for epoch in range(epochs):

    # Put model in training mode (this is the default state of a model)
    model.train()

    # 1. Forward pass on train data using the forward() method inside
    y_pred = model(X_train)

    # 2. Calculate the loss (how different are the model's predictions to the true values)
    loss = loss_fn(y_pred, y_true)

    # 3. Zero the gradients of the optimizer (they accumulate by default)
    optimizer.zero_grad()

    # 4. Perform backpropagation on the loss
    loss.backward()

    # 5. Progress/step the optimizer (gradient descent)
    optimizer.step()
```

Note: all of this can be turned into a function

Pass the data through the model for a number of epochs (e.g. 100 for 100 passes of the data)

Pass the data through the model, this will perform the `forward()` method located within the model object

Calculate the loss value (how wrong the model's predictions are)

Zero the optimizer gradients (they accumulate every epoch, zero them to start fresh each forward pass)

Perform backpropagation on the loss function (compute the gradient of every parameter with `requires_grad=True`)

Step the optimizer to update the model's parameters with respect to the gradients calculated by `loss.backward()`

PyTorch Testing Loop

```python
# Setup empty lists to keep track of model progress
epoch_count = []
train_loss_values = []
test_loss_values = []

# Pass the data through the model for a number of epochs (e.g. 100) pochs):
for epoch in range(epochs):

    ### Training loop code here ###

    ### Testing starts ###

    # Put the model in evaluation mode
    model.eval()

    # Turn on inference mode context manager
    with torch.inference_mode():
      # 1. Forward pass on test data
      test_pred = model(X_test)

      # 2. Caculate loss on test data
      test_loss = loss_fn(test_pred, y_test)

    # Print out what's happening every 10 epochs
    if epoch % 10 == 0:
        epoch_count.append(epoch)
        train_loss_values.append(loss)
        test_loss_values.append(test_loss)
        print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss} ")
```

Note: all of this can be turned into a function

Create empty lists for storing useful values (helpful for tracking model progress)

Tell the model we want to evaluate rather than train (this turns off functionality used for training but not evaluation)

Turn on `torch.inference_mode()` context manager to disable functionality such as gradient tracking for inference (gradient tracking not needed for inference) (faster performance!)

Pass the test data through the model (this will call the model's implemented `forward()` method)

Calculate the test loss value (how wrong the model's predictions are on the test dataset, lower is better)

Display information outputs for how the model is doing during training/testing every ~10 epochs (note: what gets printed out here can be adjusted for specific problems)

Evaluation Methods

- Classification evaluation methods

Metric Name	Metric Forumla	Code	When to use
Accuracy	$\textbf{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$	`torchmetrics.Accuracy()` or `sklearn.metrics.accuracy_score()`	Default metric for classification problems. Not the best for imbalanced classes.
Precision	$\textbf{Precision} = \frac{tp}{tp + fp}$	`torchmetrics.Precision()` or `sklearn.metrics.precision_score()`	Higher precision leads to less false positives.
Recall	$\textbf{Recall} = \frac{tp}{tp + fn}$	`torchmetrics.Recall()` or `sklearn.metrics.recall_score()`	Higher recall leads to less false negatives.
F1-score	$\textbf{F1-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$	`torchmetrics.F1Score()` or `sklearn.metrics.f1_score()`	Combination of precision and recall, usually a good overall metric for a classification model.
Confusion matrix	NA	`torchmetrics.ConfusionMatrix()`	When comparing predictions to truth labels to see where model gets confused. Can be hard to use with large numbers of classes.

Optimizer algorithms

- Optimizer algorithms are optimization method that helps improve a deep learning model's performance.
- While training the deep learning optimizers model, modify each epoch's weights and minimize the loss function.
  - An **optimizer** is a function or an algorithm that adjusts the attributes of the neural network, such as weights and learning rates.
  - Thus, it helps in reducing the overall loss and improving accuracy.
  - **Gradient Descent Deep Learning Optimizer**
  - **Stochastic Gradient Descent Deep Learning Optimizer**
  - **Mini-batch Stochastic Gradient Descent**

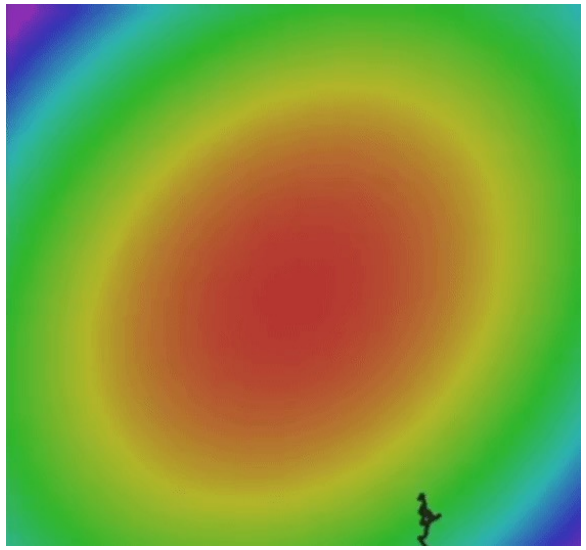
$$w = w - \texttt{learning\_rate} \cdot \nabla_w L(x_{(i:i+n)}, y_{(i:i+n)}, W)$$

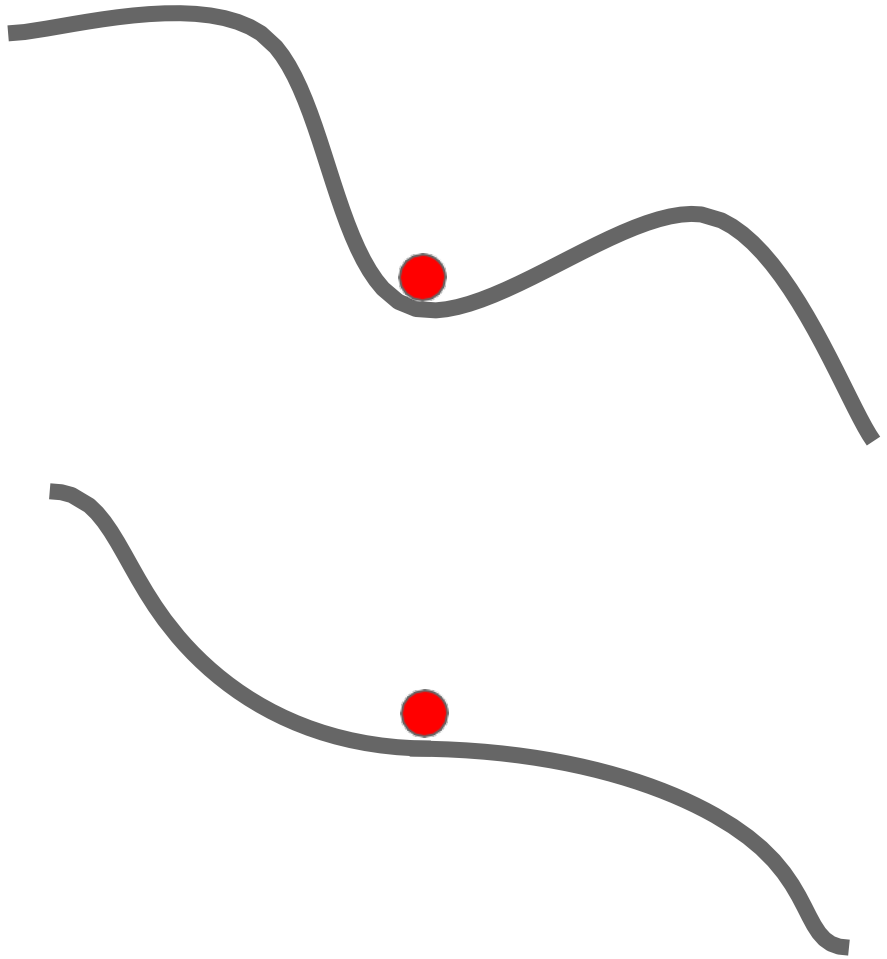
Limitations and Problems

- The gradients are still noisy because we estimate them based only on a small sample of our dataset. The noisy updates might not correlate well with the true direction of the loss function.
- If the loss function has a local minimum or a saddle point, it is very possible that SGD will be stuck there without being able to “jump out” and proceed in finding a better minimum. This happens because the gradient becomes zero so there is no update in the weight.
- Choosing a good loss function is tricky and requires time-consuming experimentation with different hyperparameters.
- The same learning rate is applied to all of our parameters, which can become problematic for features with different frequencies or significance.

Optimization: Problems with SGD

- What if the loss function has a **local minima** or **saddle point**?
- Saddle points much more common in high dimension
- Our gradients come from minibatches so they can be noisy!





A gradient close to zero in a saddle point or in a local minimum does not improve the weight parameters and prevents the whole learning process.

SGD + Momentum Optimizer

- The momentum term results in the individual gradients having less variance and thus less zig-zagging: more stable and faster convergence
  - Build up “velocity” as a running mean of gradients
  - Rho gives “friction”; typically rho=0.9 or 0.99

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

Momentum

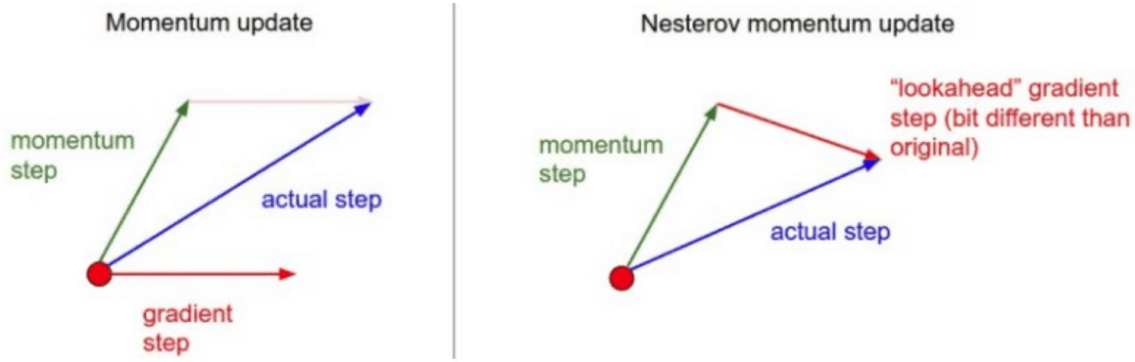
- Momentum
  - At every time step, we update our velocity by decaying the previous velocity on a factor of Friction ρ and we add the gradient of the weights on the current time. Then we update our weights in the direction of the velocity vector.
  - We can now escape local minimums or saddle points because we keep moving downwards even though the gradient of the mini-batch might be zero.
  - Momentum can also help us reduce the oscillation of the gradients because the velocity vectors can smooth out these highly changing landscapes.
  - Finally, it reduces the noise of the gradients (stochasticity) and follows a more direct walk down the landscape.

Nesterov momentum

- An alternative version of momentum, called Nesterov momentum, calculates the update direction in a slightly different way.
- Instead of combining the velocity vector and the gradients, we calculate where the velocity vector would take us and compute the gradient at this point.
- The most famous algorithm that make us of Nesterov momentum is called Nesterov accelerated gradient (NAG)

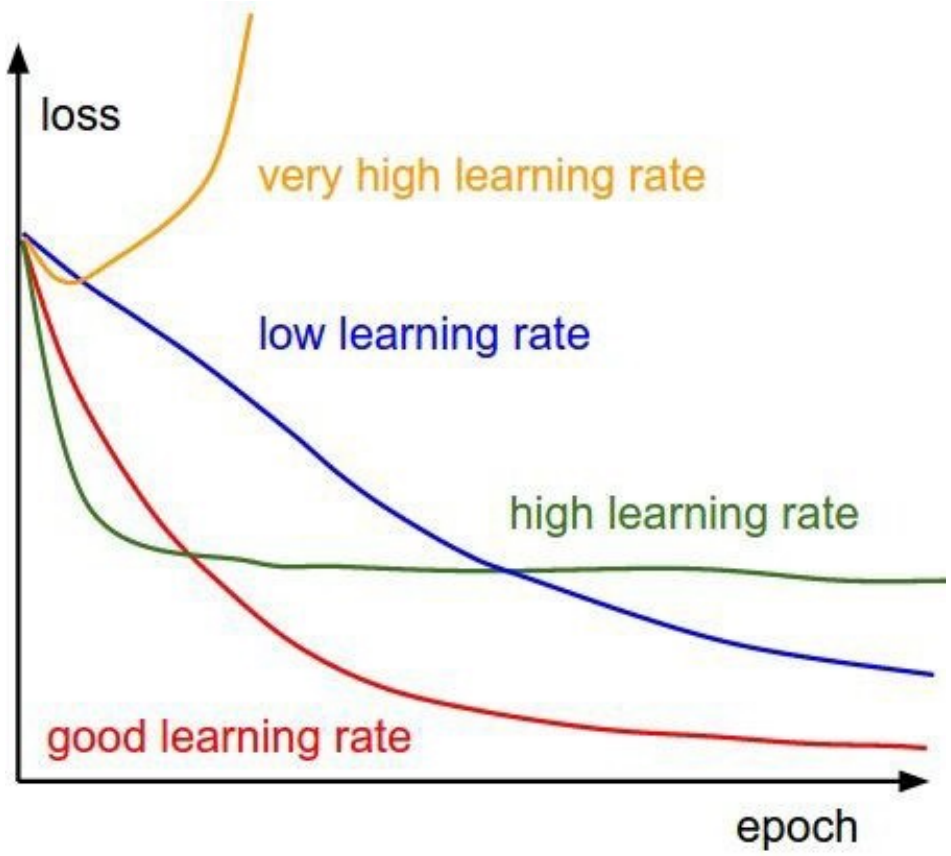
$$v_{t+1} = \rho v_t - \texttt{learning\_rate} \cdot \nabla_w L(w + \rho v_t)$$

$$w = w + v_{t+1}$$



Learning Rate

- **Learning rate** as a hyperparameter
  - Q: Which one of these learning rates is best to use?
  - A: All of them! Start with large learning rate and decay over time



Adaptive Learning Rate

- The other big idea of optimization algorithms is adaptive learning rate. The intuition is that we'd like to perform smaller updates for frequent features and bigger ones for infrequent ones.
- Adagrad
  - Uses different learning rates for each iteration
  - It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.
  - When the gradient is changing very fast, the learning rate will be smaller. When the gradient is changing slowly, the learning rate will be bigger.
  - Achieve a different learning rate for each parameter (or an adaptive learning rate).

Thank You

Address:
ENG257, SJSU

Email Address:
Kaikai.liu@sjsu.edu