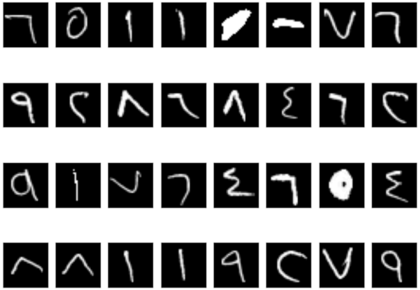
Some practical considerations

Using our own data

- suppose we want to build an arabic digit classification system



- data is give to us in, say, two spreadsheets
  - images spreadsheet with columns containing `uint8` data
  - labels spreadsheet containing a single column of `int` data

Inheriting from the `Dataset` class

```python
import pandas as pd
from torch.utils.data import Dataset

class AMNIST(Dataset):
    def __init__(self, imagesfile, labelsfile):
        super().__init__()
        df_x = pd.read_csv(imagesfile, header=None)
        df_y = pd.read_csv(labelsfile, header=None)
        self.X = df_x.iloc[:, :].to_numpy().reshape(-1,28,28) / 255.0
        self.y = df_y.iloc[:, 0].to_numpy()

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        image = self.X[idx]
        image = torch.tensor(np.expand_dims(image, axis=0))
        label = torch.tensor(self.y[idx])
        return image, label
```

Saving and loading models

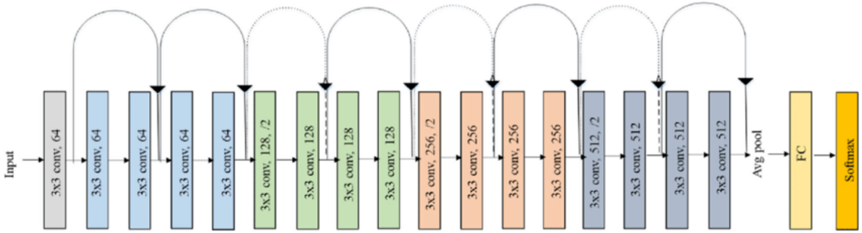
```
# saving/loading models
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

net = Net()
net.load_state_dict(torch.load(PATH))
```

Example of finetuning an existing network

```python
from torchvision import models

net = models.resnet18(pretrained=True)
for layer in net.parameters():
  layer.requires_grad=False
net.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
↪  padding=(3, 3), bias=False)
net.fc = nn.Linear (in_features=512, out_features=10, bias=True)
```



Data transformations

```python
from torchvision import transforms

transform_train = transforms.Compose([
                        transforms.RandomCrop(28, padding=4),
                        transforms.ToTensor(),
                    ])
# Training dataset
train_data = MNIST(root='./datasets', train=True, download=True,
↪   transform=transform_train)

train_loader = torch.utils.data.DataLoader(train_data,
↪   batch_size=batch_sz, shuffle=True, pin_memory=True)
```

Guess what this does:

```python
dataset = MyMNIST(image_dir, label_spreadsheet,
↪   transforms.Compose([transforms.ToTensor(),
↪   transforms.ColorJitter(...), transforms.RandomAffine(90)]))
train_loader=DataLoader(dataset, batch_size=10)

```

Standard deviation and data normalization

- $\mathrm{avg}(x) = \mathbf{1}^T x/n$
- de-meaned vector is $\tilde{x} = x - \mathrm{avg}(x)\mathbf{1}$ (so $\mathrm{avg}(\tilde{x}) = 0$)
- standard deviation of x

$$\mathrm{std}(x) = \mathrm{rms}(\tilde{x}) = \frac{\|x - (\mathbf{1}^T x/n)\mathbf{1}\|}{\sqrt{n}}$$

- $\mathrm{std}(x)$ gives "typical amount" x_i vary from $\mathrm{avg}(x)$
- $\mathrm{std}(x) = 0$ only if $x = \alpha\mathbf{1}$ for some α
- a basic formula

$$\mathrm{rms}(x)^2 = \mathrm{avg}(x)^2 + \mathrm{std}(x)^2$$

- standardization ($\mu = 0$, $\sigma = 1$, z-scores)

$$z = \frac{1}{\mathrm{std}(x)}(x - \mathrm{avg}(x)\mathbf{1})$$

Three sources of errors

- error between a trained neural network and a perfect classification function to learn can be viewed as having three components
    - optimization error
    - approximation error
    - generalization error

Practical training considerations

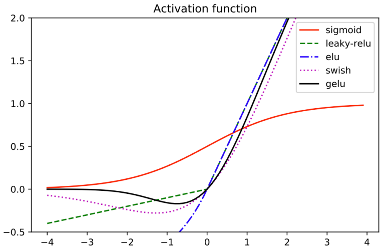
- the exploding gradient problem—gradient clipping solution

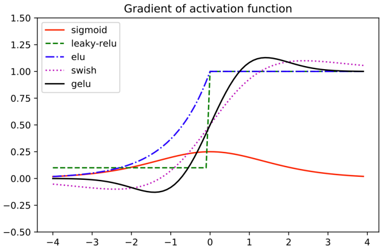
$$g' = \min(1, \frac{c}{\|g\|})g$$

- the vanishing gradient problem
  - non-saturating activation functions
  - residual networks (ResNets)
  - parameter initialization
  - standardize activations at each layer

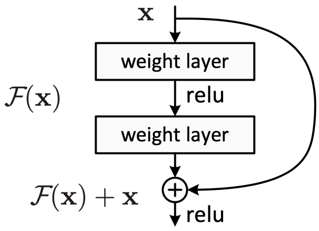
Non-saturating activation functions

Name	Definition
Sigmoid	$\boldsymbol{\sigma}(a) = \frac{1}{1+e^{-a}}$
Hyperbolic tangent	$\tanh(a) = 2\boldsymbol{\sigma}(2a) - 1$
Softplus	$\sigma_+(a) = \log(1 + e^a)$
Rectified linear unit	$\mathrm{ReLU}(a) = \max(a, 0)$
Leaky ReLU	$\max(a, 0) + \alpha \min(a, 0)$
Exponential linear unit	$\max(a, 0) + \min(\alpha(e^a - 1), 0)$
Swish	$a\boldsymbol{\sigma}(a)$
GELU	$a\Phi(a)$

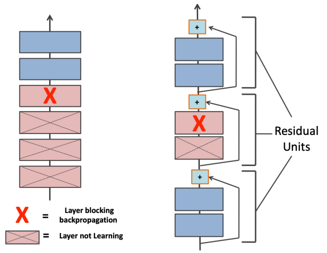




ResNets



$\mathbf{x}$
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu



- intuition is that it is easier to learn to generate a small perturbation to the input than to directly predict the output
- gradient at layer l depends on the gradient of layer L in a way that is independent of the depth of the network

Parameter initialization

- initialization of θ parameters need to be done somewhat carefully
- sampling parameters from a standard normal with fixed variance can result in exploding activations or gradients
- pytorch has a (good) Glorot initialization, $\sigma^2 = 2/(n_{in} + n_{out})$
- data-driven initializations, which compute variances of the activations across a minibatch, may be used

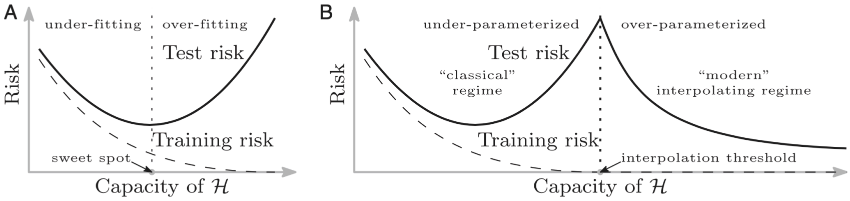
Regularization

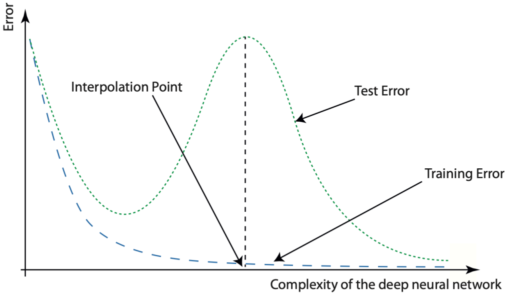
- it is common to add a regularization term to the NLL loss objective

$$\mathcal{L}(\theta) + \text{regularization}$$

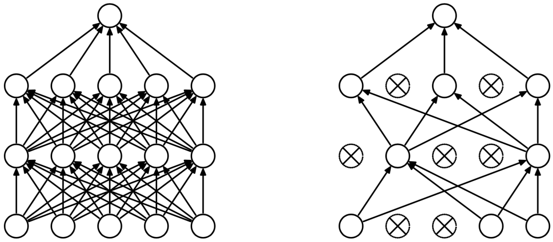
- $\lambda\|\theta\|^2$ often used for regularization
  - encourage small weights, and therefore simpler models
  - referred to as weight decay
- stopping when the error on validation starts to increase is a regularization strategy that prevents overfitting
  - referred to as early stopping
  - but... double descent curves have been often observed
- dropout can also prevent overfitting
- encourage flat minima
- and many others....

Double descent in overparameterized networks



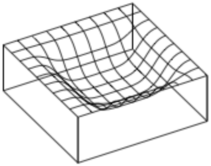


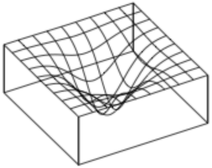
Dropout



- turn off (randomly, on a per-example, or per-batch basis) all outgoing connections from each neuron with probability p
- intuitively, each neuron must learn to perform well even if others units are randomly missing—prevents complex and fragile dependencies from being learned

Local minima are not all created equal





- flat minima correspond to regions in parameters space where there is a lot of posterior uncertainty
- hence solutions from this region are unlikely to memorize irrelevant details from the training set
- noise in SGD may prevent algorithm from entering narrow regions of landscape
- sharpness aware terms can be added the objective
- estimates of variance of minibatch gradients (a measure of stability, and hence generalization ability) may be added

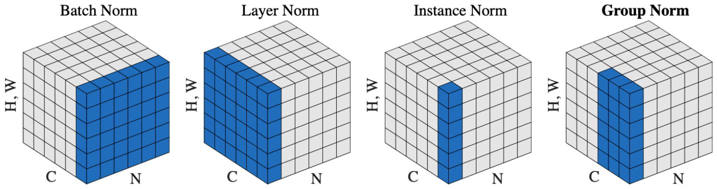
Batch normalization layer

- in order to reduce vanishing or exploding gradient problems with SGD, it is often useful to normalize data in every layer
- similar to standardizing input data
- batch normalization: replace activation vector z_n by $\tilde{z}_n$

$$\begin{aligned}
\mu_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{z \in \mathcal{B}} z \\
\sigma_{\mathcal{B}}^2 &= \frac{1}{|\mathcal{B}|} \sum_{z \in \mathcal{B}} (z - \mu_{\mathcal{B}})^2 \\
\hat{z}_n &= \frac{(z - \mu_{\mathcal{B}})}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
\tilde{z}_n &= \gamma \hat{z}_n + \beta
\end{aligned}$$

- done dynamically in every layer with every batch
  - trainable parameters: one γ and one β per feature map per layer
  - note: must be held fixed at inference time (post training)

Other kinds of normalizations



- can be conveniently added as layers in the neural network definition

Optimization for training NN

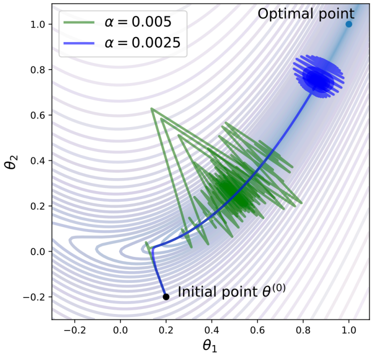
- learn a function $f(x_n;\theta) \approx y_n$
- loss $\mathcal{L}(\theta) = \frac{1}{N}\sum_{n=1}^{N} \ell(y, f(x_n;\theta))$
  - "loss" incurred for not properly aligning our prediction $f(x)$ with y
- regularization to avoid overfitting

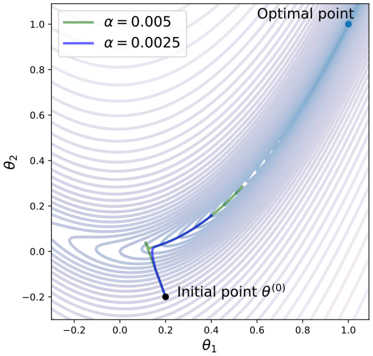
$$\theta^* = \underset{\theta}{\operatorname{argmin}}\ \mathcal{L}(\theta) + \lambda\,\mathrm{pen}(\theta)$$

  - $\mathrm{pen}(\theta)$ takes lower values for parameters θ that yield functions f with lower complexity, λ is a regularization parameter that can be tuned
- local descent, compute next iterate as $\theta^{(k+1)} = \theta^{(k)} + \alpha^{(k)} d^{(k)}$
- choice of learning rate $\alpha^{(k)}$
  - fixed α called learning rate
  - decaying step factor

$$\alpha^{(k)} = \alpha^{(1)}\gamma^{k-1}$$

Decaying learning rate





- fixed learning rate $\alpha^{(k)} = \alpha$
- vs exponentially decaying learning rate $\alpha^{(k)} = \alpha\, 0.99^k$

Stochastic gradient descent

- search in direction of steepest descent (negative gradient direction)

$$d^{(k)} = -\nabla f(\theta^{(k)})$$

- in our use case of interest, our objective is a loss of the form:

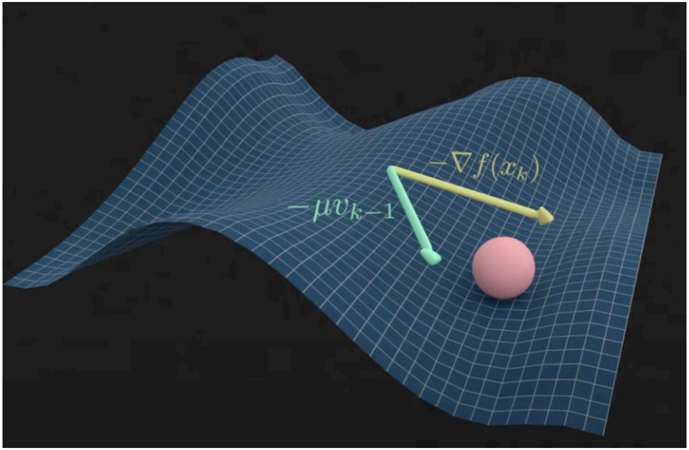
$$\frac{1}{N}\sum_{n=1}^{N} \ell(y_n, f(x_n; \theta))$$

- a noisy/randomized gradient may be obtained by choosing a subset of the training data (batch)

Termination conditions

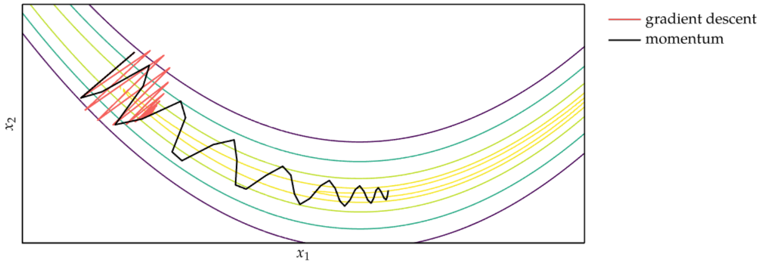
- maximum iterations
$$k > k_{\max}$$
- absolute improvement
$$f(\theta^{(k)}) - f(\theta^{(k+1)}) < \epsilon_a$$
- relative improvement
$$f(\theta^{(k)}) - f(\theta^{(k+1)}) < \epsilon_r |f(\theta^{(k)}|$$
- gradient magnitude
$$\|\nabla f(\theta^{(k+1)}\| \leq \epsilon_g$$

Momentum



Momentum

- some functions cause gradient descent to get stuck
- momentum overcomes these issues by replicating the effect of physical momentum



Further improvements

- Adagrad
  - instead of using the same learning rate for all components of x, Adaptive Subgradient method (Adagrad) adapts the learning rate for each component of x
  - dulls the influence of parameters with consistently high gradients
- RMSprop
  - extends Adagrad to avoid monotonically decreasing learning rate by maintaining a decaying average of squared gradients
- Adam
  - incorporates ideas from momentum and RMSProp
  - the adaptive moment estimation method (Adam), adapts the learning rate to each parameter