0.1 Linear classification and kNN

Nearest neighbor: Train, O(1): learn the training dataset; Predict, O(n): for each test image, find closest train image, predict label of nearest image; Hyperparameters: K neighbors and distance metric (L1, L2)

0.1.1 Linear classification

Linear classification ($y = WX + b$) defined by the loss function used to select W
SVM loss ("hinge loss"): Hyperparameters: Δ and whether to square the hinge loss. Hinge loss is not unique to scaling: In a zero loss case, W and λW result in same loss. Note if we regularize, using Δ as a hyperparameter is redundant and we can just set to 1.

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + \Delta & \text{else} \end{cases} = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta) \qquad L = \frac{1}{n} L_i$$

Softmax loss ("cross-entropy loss"): At initialization, all the s_i's will be approximately equal, so loss will be $\approx -log(1/C) = log(C)$. Numerical stability is a concern when taking exponents. "Normalization trick" multiplies the top and bottom by $\exp - \max_j f_j$

$$P(Y = k \mid X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \qquad L_i = -\log P(Y = k \mid X = x_i) = -\log \frac{e^{s_k}}{\sum_j e^{s_j}}$$

0.2 Optimization

```
# Stochastic Gradient Descent (SGD)
data_batch = sample_training_data(data, 256) # sample 256 examples
weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
weights += -step_size * weights_grad # perform parameter updates
# Problem: move in one direction more than another, local min/max, noisy; solution: build up "velocity" as running mean of gradients
# SGD+Momentum
vx = rho * vx + weights_grad # initialize vx to 0
weights += -step_size * vx
# Nesterov momentum: more advanced, uses gradient of where we're headed
vx = rho * vx - alpha * evaluate_gradient(loss_fun, data_batch, weights + rho * vx)
weights += -step_size * vx
# AdaGrad: add element-wise scaling of gradient; progress along steep directions is damped and along flat directions is accelerated
grad_squared += weights_grad * weights_grad
weights += -step_size * weights_grad / (np.sqrt(grad_squared) + 1e-7)
# RMSProp: addresses problem that step size decays to zero, solution: make it leaky!
grad_squared = decay_rate * grad_squared + (1 - decay_rate) * weights_grad * weights_grad
weights += -step_size * weights_grad / (np.sqrt(grad_squared) + 1e-7)
# Adam: can we combine momentum with element-wise scaling? (basically Momentum + RMSProp)?
first_moment = beta1 * first_moment + (1 - beta1) * weights_grad # calculate momentum, first_moment initialized to 0
second_moment = beta2 * second_moment + (1 - beta2) * weights_grad * weights_grad # RMSProp, second_moment initialized to 0
first_unbias = first_moment / (1 - beta ** t) # bias correction, t is the number of the iterations
second_unbias = second_moment / (1 - beta2**t) # bias correction needed b/c moments initialized to 0
weights += -step_size * weights_grad / (np.sqrt(grad_squared) + 1e-7) # RMSProp
```

Learning rate (aka step-size) is a hyperparameter that can be learned. It can also change over time

- Step decay: reduce learning rate at a few fixed points over the training
- Cosine decay: $\alpha_t = 1/2 * \alpha_0 * (1 + \cos t\pi/T)$
- Linear decay: $\alpha_t = \alpha_0(1 - t/T)$

0.3 Backpropagation

If we can compute $\partial L/\partial W_i$ then we can learn the weights W_i through gradient descent
Matrix multiplication ($Y = WX$ for $W \in \mathbb{R}^{D\times M}, X \in \mathbb{R}^{N\times D}, Y \in \mathbb{R}^{N\times M}$)

- $\partial L/\partial X = (\partial L/\partial y)W^T$ for $\partial L/\partial Y \in \mathbb{R}^{N\times M}, \partial L/\partial X \in \mathbb{R}^{N\times D}$
- $\partial L/\partial W = X^T(\partial L/\partial Y)$ for $\partial L/\partial W \in \mathbb{R}^{D\times M}$

TODO: tricks to computing gradients. example

0.4 Neural Networks

Activation functions

- **Sigmoid:** $1/(1 + e^{-x})$. Two major drawbacks, i) kills gradients: gradients near either tail are almost zero, so it cannot pass through upstream gradients ii) not zero-centered: can produce zig-sagging dynamics in gradient updates since it will lead to all pos. or all neg. gradient updates
- **Tanh:** $tanh(x)$. Zero-centered, but still kills gradients
- **ReLU:** $max(0, x)$. Most popular for its simple compute. ReLU units can "die" during training. Large gradient can cause weights to update so that the neuron will not activate again.
- **Leaky ReLU:** $max(\gamma x, x), \gamma$ small. Attempts to fix "dying ReLU". **Maxout** further generalizes leaky ReLU: $\max(w_1^T x + b_1, w_2^T x + b_2)$

- **ELU:** $\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

Initialization: Problem with initializing weights to 0: if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. Solution is to initialize to random small inputs.

0.4.1 Regularization

- L1, L2, elastic net
- Max norm constraints: clamp the weight vector of each neuron to satisfy $\|W\|_2 < c$
- Dropout: keep each neuron active with probability p. During testing, no dropout is applied, but all weights are scaled by p. In practice "inverted dropout" is implemented to improve test-time prediction (i.e., divide by p in ReLU step during training)

0.5 Convolutions

Feature representation: color histogram • histogram of oriented gradients • bag of words
Convolutional neural nets (CNN) learn these features so they don't need to be defined ahead of time. Further, CNNs take advantage of data structure.

- Convolution filters are of dimension C2xC1xHFxWF: C2 out-channels, C1 in-channels (same as num channels in input layer), HF filter height, WF filter width. They also include a C2-dimensional bias vector.
- Output activation layer size, assuming square input, HxH, square filter, FxF, pad, P, and stride, S: $(H + 2P - F)/S + 1$
- Successive convolution adds $F - 1$ to the receptive field size; with L layers, the receptive field size is $1 + L * (K - 1)$
- Number of parameters: $F^2 * C1 * C2 + C2$

0.5.1 Pooling

Makes representations smaller and more manageable • Operates on each activation map independently • No learnable parameters • introduces spatial invariance • Filter size, FxF, stride, S, produces output $H_2 = (H_1 - F)/S + 1$, $W_2 = (W_1 - F)/S + 1$

0.5.2 Batch normalization

Usually inserted after fully connected or convolutional layers • allows higher learning rates and faster convergence • more robust to initialization • as with all normalization techniques, normalization parameters are determined from the training data only and applied to train/test/val • Fully connected nets, $x \in \mathbb{R}^{N\times D} \implies (\mu, \sigma, \gamma, \beta) \in \mathbb{R}^{1\times D}$ • Convolutional nets $x \in \mathbb{R}^{N\times C\times H\times W} \implies (\mu, \sigma, \gamma, \beta) \in \mathbb{R}^{1\times C\times 1\times 1}$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \ (\gamma, \beta \text{ learned}), \quad \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}, \quad \mu_j = \frac{1}{n}\sum_i x_{i,j}, \quad \sigma_j^2 = \frac{1}{n}\sum_i (x_{i,j} - \mu_j)^2$$

Test time: Mean and variance estimates depend on each minibatch, so we need to use the running average of each (calculated during training) to use when testing the model.
Layer normalization: Fully connected nets, $x \in \mathbb{R}^{N\times D} \implies (\mu, \sigma) \in \mathbb{R}^{N\times 1}, (\gamma, \beta) \in \mathbb{R}^{1\times D}$ • Convolutional nets $x \in \mathbb{R}^{N\times C\times H\times W} \implies (\mu, \sigma) \in \mathbb{R}^{N\times 1\times 1\times 1}, (\gamma, \beta) \in \mathbb{R}^{1\times C\times 1\times 1}$
Instance normalization: Convolutional nets $x \in \mathbb{R}^{N\times C\times H\times W} \implies (\mu, \sigma) \in \mathbb{R}^{N\times C\times 1\times 1}, (\gamma, \beta) \in \mathbb{R}^{1\times C\times 1\times 1}$
Group normalization: Convolutional nets $x \in \mathbb{R}^{N\times C\times H\times W}$ and $C/g = G$ groups $\implies (\mu, \sigma) \in \mathbb{R}^{N\times G\times 1\times 1}, (\gamma, \beta) \in \mathbb{R}^{1\times C\times 1\times 1}$

0.5.3 Architectures

VGGNet: Smaller and more convolution layers, leading to same receptive field, more activation layers, and fewer learned parameters.
GoogLeNet: Apply parallel filter op.s (1x1, 3x3, 5x5) and pooling op., concatenate together channel-wise. Reduce depth with 1x1x64 convolutional layer first. Many fewer params, avoids expensive FC layers
ResNet: Deeper models should work better, but are harder to optimize. In ResNet, update $x \to F(x) \to y = ReLU(F(x)) \to H(y) \to z = ReLU(H(y))$, to $x \to F(x) \to y = ReLU(F(x)) \to H(y) \to z = ReLU(H(y) + x)$

0.5.4 Transfer learning

Freeze most layers of a trained CNN, reinitialize and train last layer, or last several layers • These last several layers can be used as inputs to other models, like what is used in object detection, and other more complex deep learning problems

0.6 TODO

Data augmentation RNNs / Attention / Transformers Image captioning Object detection and segmentation Style transfer