Universidade Federal de Santa Catarina
Departamento de Informática e Estatística

Gabriel Baldessar

DETECÇÃO DE ARMAS DE FOGO EM VÍDEO ATRAVÉS DE REDES NEURAIS

Florianópolis
2021

Gabriel Baldessar

DETECÇÃO DE ARMAS DE FOGO EM VÍDEO ATRAVÉS DE REDES NEURAIS

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como parte dos requisitos necessários para a obtenção do Título de Cientista da Computação.
Orientador: Prof. Dr. Rafael de Santiago

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística

Florianópolis
2021

Gabriel Baldessar

DETECÇÃO DE ARMAS DE FOGO EM VÍDEO ATRAVÉS DE REDES NEURAIS

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Cientista da Computação.

Comissão Examinadora

Prof. Dr. Rafael de Santiago
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. Renato Fileto
Universidade Federal de Santa Catarina

Prof. Dr. Elder Rizzon Santos
Universidade Federal de Santa Catarina

Florianópolis, 17 de maio de 2021

Dedico este trabalho a minha família e a todos aqueles que, de alguma forma, auxiliaram para a concretização desta etapa.

Agradecimentos

Agradeço a minha família que me apoiou e guiou durante essa trilha chamada universidade, Principalmente meus pais Paulo e Alessandra e a minhã irmã Beatriz. Sem eles este trabalho não teria sido possível.
Agradeço também ao Dr. Rafael de Santiago pelo apoio e conhecimento no desenvolvimento deste projeto.
Agradeço aos amigos Pamps, Beavis, Dimitri, Mazza, Eiske, Leonardo, Caio e Salomão e tantos outros amigos feitos durante o curso que ajudaram a fazer a faculdade um lugar melhor para mim.

"Quem quer rir, tem que fazer rir."
(Major Rocha)

Resumo

O número de furtos e roubos vem crescendo nos últimos anos. Nas situações mais graves, o uso de armas de fogo pode agravar a situação. Para reduzir esse problema, diversos espaços como lojas, depósitos, aeroportos e outros mais contratam diversos serviços de segurança. Entre eles o de vigilância remota. Visando apoiar a vigilância remota, este trabalho trata do desenvolvimento de uma rede neural artificial com o objetivo de auxiliar operadores humanos de sistemas de vigilância em circuitos internos. Mais especificamente, a rede neural deve reconhecer armas de fogo em video sem que seja necessário o envolvimento de um operador humano. Para endereçar esse objetivo, fora realizada uma pesquisa bibliográfica em literatura sobre redes neurais e reconhecimento de imagens, incluindo uma busca por trabalhos similares na literatura científica, bem como uma especificação preliminar da arquitetura da rede neural. Para atingir o objetivo, foi realizado o levantamento da literatura relacionada, redes que efetuam a classificação foram encontradas, um modelo de rede neural foi obtido e experimentações foram realizadas variando configurações de entrada no treinamento. Diversos experimentos foram realizados sobre a rede YOLOv3. Os melhores resultados foram obtidos obtiveram acurácia de 82,3%, uma precisão de 96,5% e uma taxa de recall de 84,4%.

Palavras-Chave: Redes neurais; Vigilância remota autônoma; Reconhecimento de padrões em vídeos

Abstract

The number of thefts and robberies has beem growing in the past years. In the worst cases the use of guns can make the situation worse. Looking for reducing this problem, deposits, stores, gas stations, airports and several more locations hires several security services. Between those services are the remote surveillance. Looking to help the remote surveillance, this work deals with the development of a artificial neural netwotk in order to assist human operators of surveillance systems in camera internal circuits. Specifically, the neural network must recognize firearms on video without the need of interaction from an human operator. To address this goal, a bibliographic search had been carried out in literature on neyral networks and image recognition, including a search for similar works in the scientific literature, as well as a preliminary specification of the neural network architecture. Several experiments were carried out on the network. The best results achieved an accuracy of 82.3%, a precision of 96.5% and a recall rate of 84.4%. In order to achieve the objective, a survey of related literature was carried out, network that peform classification were found, a model of neural network was obteined and experiments were made varying the training configuration.

Keywords: Neural Network; Autonomous remote surveillance. Patterns Recognition in video

Lista de figuras

Lista de tabelas

Lista de Siglas e Abreviaturas

IA	*Inteligência artificial*
RNC	*Rede Neural Convolucional*
UFSC	*Universidade Federal de Santa Catarina*

Sumário

1 Introdução

De acordo com O Escritório das Nações Unidas sobre Drogas e Crime (UNODC), o número de crimes envolvendo armas de fogo é alto em diversos países, como no México, com a marca de 21,5 crimes para cada 100.000 habitantes (UNODC, 2013).

Uma forma de reduzir o uso indevido de armas de fogo e através da prevenção dos crimes, por meio de detecção prévia, para que então agentes de segurança e forças da lei sejam capazes de agir (OLMOS; TABIK; HERRERA, 2017). Nesse sentido, uma das iniciativas é o uso de câmeras de monitoramento. Geralmente, a ideia é a de que uma equipe fique responsável por monitorar câmeras para tentar identificar situações suspeitas e agir previamente. Essa iniciativa é comum na preservação em condomínios, empresas e prédios públicos (PEI, 2019).

Câmeras de monitoramento podem envolver um custo maior do que a aquisição e a manutenção de equipamentos. É necessário que as imagens sejam analisadas constantemente para tomar-se ações pró-ativas. Por isso, geralmente esses sistemas envolvem o custo de contratar profissionais humanos que devem acompanhar as imagens ao vivo. Isso faz com que o custo seja alto para que seja empregado em um amplo aspecto (LEPESKA, 2011).

Soluções autônomas envolvem reconhecimento de padrões em imagens. Alguns exemplos de trabalhos que abordam o tema na literatura são:

- (HIJAZI; KUMAR; ROWEN, 2015): Trabalho que cobre o básico em relação às RNCs e aponta as sua vantagens em relação a outras técnicas;
- (DEEPTHI et al., 2019): Trabalho que mostra como tecnologias relacionadas à vigilância podem se beneficiar da utilização conjunta com redes neurais;
- (OLMOS; TABIK; HERRERA, 2017): Este Trabalho mostra como utilizar redes neurais junto a câmeras de segurança com o propósito de controle e vigilância.

Desta maneira, uma solução possível e inovadora seria utilizar de forma conjunta, um sistema vigilância com uma rede neural capaz de reconhecer armas de fogo e alertar as forças da lei em tempo real, a fim de diminuir o tempo de resposta das autoridades competentes (DEEPTHI et al., 2019). Nesse contexto, a trabalho apresenta um estudo sobre método automático de pessoas portando armas de mão em vídeos. Para isso, as seguintes atividades foram realizadas: (i) procurar por trabalhos similares na literatura; (ii) especificar método computacional de aprendizagem supervisionada para realizar a tarefa de reconhecimento de padrões; (iii) coletar bases de dados com exemplos rotulados para treinar a técnica a ser utilizada; (iv) desenvolver método especificado; (v) experimentar e analisar parâmetros do método computacional desenvolvido; (vi) comparar resultados

obtidos com outros encontrados na literatura; (vii) dar publicidade aos resultados através de veículos de divulgação científica.

1.1 Objetivos

Esta seção visa apresentar os objetivos propostos pelo trabalho que será realizado.

1.1.1 Objetivo Geral

Este trabalho tem como objetivo treinar uma rede neural para que ela seja capaz de reconhecer armas de fogo pessoais portadas por indivíduos em imagens de vídeo.

1.1.2 Objetivos Específicos

- O1. Pesquisar sobre trabalhos relacionados a reconhecimento de imagens utilizando redes neurais;
- O2. Obter *datasets* para o treinamento da rede neural;
- O3. Desenvolver uma rede neural capaz de reconhecer armas de fogo expostas por indivíduos;
- O4. Testar a rede neural e analisar os resultados obtidos.

1.1.3 Metodologia

O presente trabalho se trata de uma pesquisa quantitativa, pois avalia a solução com base na contagem de Verdadeiros Positivos, Falsos Positivos, Verdadeiros Negativos e Falsos Negativos para um método de classificação.

Para se desenvolver a pesquisa, fez-se: (i) um levantamento da fundamentação teórica e de trabalhos relacionados; (ii) uma análise da metodologia e resultados dos trabalhos relacionados; (iii) a seleção de uma solução em redes neurais que atuava com detecção de padrões em imagens; (iv) especificação dos experimentos; (v) preparação do *dataset*; (vi) execução e coleta de dados dos experimentos; (vii) análise dos dados obtidos.

1.1.4 Estrutura do Texto

Este documento está estruturado em cinco capítulos.

O Capítulo 1, Introdução, apresentou uma visão geral do trabalho, incluído: problematização, objetivos do projeto e metodologia utilizada.

No Capítulo 2, Fundamentação Teórica, possuí uma revisão da literatura sobre detecção de armas de fogo, redes neurais, inteligencia artificial e alguns *frameworks* que podem ser utilizados para o desenvolvimento da aplicação.

No Capítulo 3, trabalhos relacionados, é feita a analise de outros trabalhos que abordam um tema similar ao que será apresentado aqui, detecção de armas de fogo em video.

No Capítulo 4, especificação e Métodos, são especificados alguns pontos do trabalho: como o sistema será testado, como os testes serão avaliados, aspectos da rede utilizada e o cronograma esperado da disciplina TCC II.

No Capítulo 5, são apresentadas as Conclusões e trabalhos futuros.

2 Revisão de Literatura

Neste capítulo serão abordados assuntos relativos a tecnologias utilizadas para tratamento de imagens utilizando inteligencia artificial com enfoque em redes neurais. Serão apresentados conceitos estabelecidos na literatura, dando assim um embasamento para a pesquisa.

2.1 Redes Neurais

Concebida em 1943 em um artigo escrito por Warren McCulloch e Walter Pitts, onde era descrito como um nerônio deveria funcionar, foi modelada a primeira rede neural. Sua composição era de forma simples com circuitos eletrônicos (MCCULLOCH; PITTS, 1943). A partir dai abriu-se caminho para dois ramos de pesquisa em redes neurais:

- Processos biológicos no cérebro;
- Aplicação de redes neurais em Inteligencia Artificial (IA)

Fukushima (1975) concebe a primeira ideia de redes neurais multicamadas tendo como principio a hipótese: *"A sinapse do neurônio X para o neurônio Y é reforçada quando X dispara e nenhum outro neurônio próximo a Y é ativado mais forte que Y "*.

A ideia original de uma abordagem utilizando rede neural era criar um sistema computacional que pudesse resolver problemas gerais da mesma forma que um cérebro humano, porém com o passar do tempo, após observar diversas dificuldades, como representação do problema e a própria capacidade do hardware da época, a técnica acabou sendo utilizada para resolver problemas específicos, sendo a rede neural treinada unicamente para este propósito (BASHEER; HAJMEER, 2000).

A primeira a camada de nerônios é a entrada da rede, onde os dados são inseridos na rede e geralmente não ocorre nem um tipo de processamento em cima dos mesmos. Supondo uma rede com N camadas, as camadas 2 ate $N - 1$ são chamadas de camadas intermediárias, ou camadas ocultas, nelas é onde o processamento dos dados acontece, afim de se chegar ao resultado desejado. E por fim, a ultima camada, conhecida como camada de saída, é a camada que dará o resultado da analise da rede sobre os dado. Na Figura 1, uma topologia de uma rede neural pode ser visualizada.

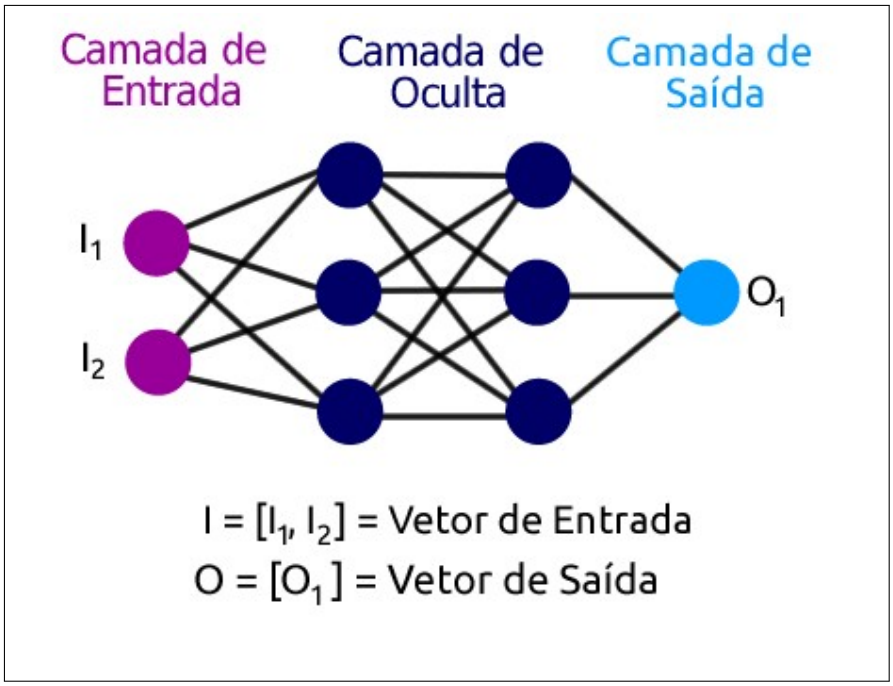


Figura 1 – Topologia de uma rede neural multicamada

Cada neurônio recebe todos os valores da camada anterior. Cada valor recebido é multiplicado pelos pesos das ligações entre os neurônios, representados na Figura 2 pelos vetores "w", e somado com uma constante chamada *bias*, representado na Figura 2 por "b". Essa constante possui o intuito de centralizar a curva da função de ativação em um valor conveniente.

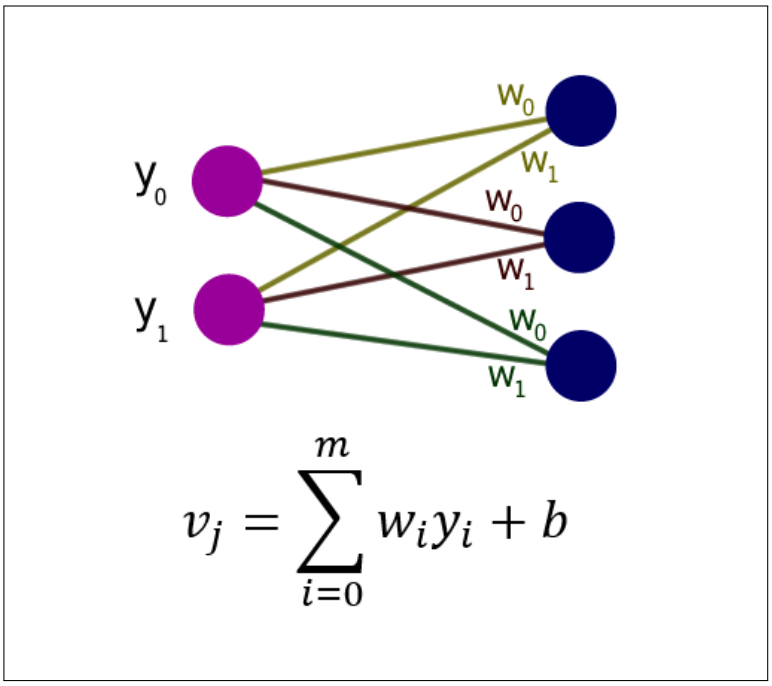


Figura 2 – Somatório ponderado das sinapses de cada neurônio

2.1.1 Função de ativação

Como citado anteriormente, as redes neurais artificiais são modelos matemáticos visando replicar o comportamento dos neurônios do cérebro, afim de alcançar algum objetivo predefinido. Porém, caso os dados passem adiante de neurônio em neurônio simplesmente, a rede neural não sera mais poderosa do que uma simples regressão linear (OLGAC; KARLIK, 2011). Para resolver este problema, foram adicionadas funções de ativação nas saídas de neurônios das camadas intermediárias.

Neurônios recebem como estimulo inicial uma carga elétrica(dados de entrada), e com base em sua importância para o momento, são ativados, repassando o estimulo ao próximo neurônio na cadeia. Abstraindo matematicamente o neurônio, podemos dividí-lo nas seguintes partes:

- x_i para uma amostra dados dados de entrada;
- w_i para um peso treinado paro os índices do dado de entrada;
- b para o valor do bias;
- f para a função de ativação.

Com essas variáveis abstraídas, é possível aplicar uma função de primeiro grau e realizar o somatório dos resultados para obter o valor de um neurônio individualmente.

Baseado no resultado do somatório de cada execução, será necessário aplicar uma função não linear sobre o resultado para decidir se o neurônio é considerado ativo, ou seja, que seu resultado é importante para alcançar a solução do problema (GLOROT; BENGIO, 2010). As funções não lineares são também conhecidas como funções de ativação. A Figura 3 mostra a estrutura do neurônio

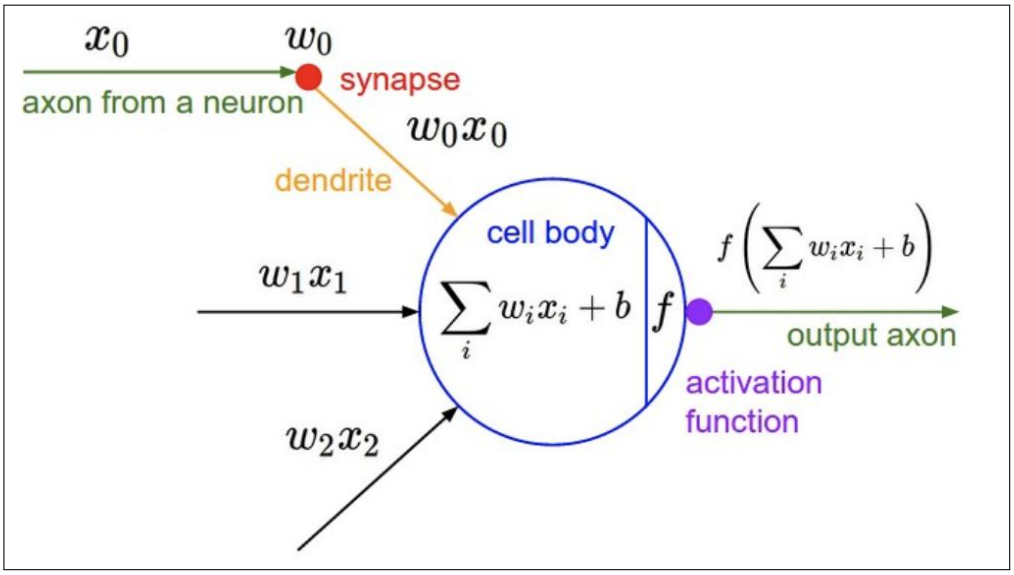


Figura 3 – Estrutura do neurônio, contendo variáveis da abstração

Diversas funções podem ser utilizadas como funções de ativação. A Figura 4 representar as principais encontradas na literatura.

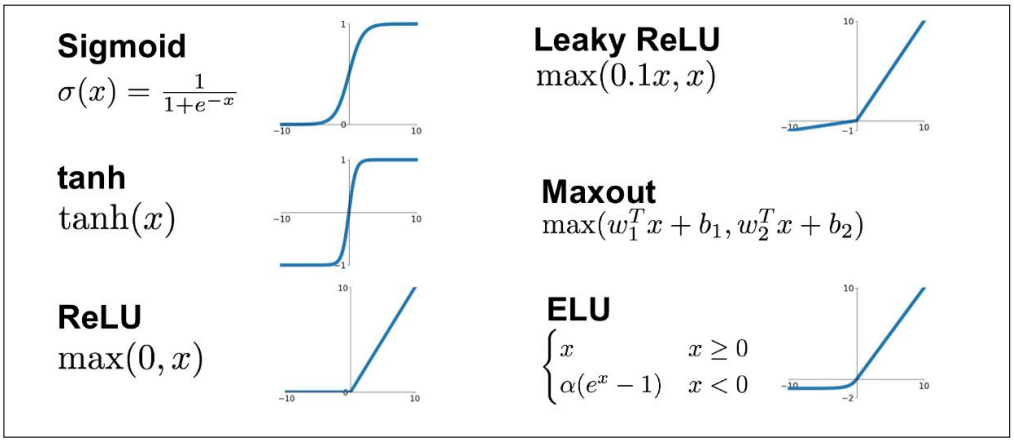


Figura 4 – Principais funções de ativação utilizadas

2.1.2 Função de custo

Para que a rede neural possa ser treinada e fazer com que exista a conversão dos pesos, de forma que consigam satisfazer os problemas de e detecção, é necessário que os pesos possam se adaptar dinamicamente. Esse processo é chamado de treinamento da rede neural, dessa forma tornando os pesos e bias melhores para a entrada do problema.(DATA SCIENCE ACADEMY, 2019)

Para que um algoritmo de otimização possa ser utilizado, é necessário primeiro que seja definida uma função de custo, função esta responsável por medir o quão bem o modelo proposto resulta nos dados esperados. A Figura 5 mostra as funções de custo mais utilizadas na literatura

symbol	name	equation
$\mathcal{L}_1$	L_1 loss	$\|\mathbf{y} - \mathbf{o}\|_1$
$\mathcal{L}_2$	L_2 loss	$\|\mathbf{y} - \mathbf{o}\|_2^2$
$\mathcal{L}_1 \circ \sigma$	expectation loss	$\|\mathbf{y} - \sigma(\mathbf{o})\|_1$
$\mathcal{L}_2 \circ \sigma$	regularised expectation loss[1]	$\|\mathbf{y} - \sigma(\mathbf{o})\|_2^2$
$\mathcal{L}_\infty \circ \sigma$	Chebyshev loss	$\max_j \lvert\sigma(\mathbf{o})^{(j)} - \mathbf{y}^{(j)}\rvert$
hinge	hinge [13] (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})$
hinge^2	squared hinge (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^2$
hinge^3	cubed hinge (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^3$
log	log (cross entropy) loss	$-\sum_j \mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}$
log^2	squared log loss	$-\sum_j [\mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}]^2$

Figura 5 – Principais funções de custo utilizadas (JANOCHA; CZARNECKI, 2017)

2.1.3 Otimização

A função de custo ajudará a identificar o quão bem o modelo treinado está predizendo os valores. Com isso, já será possível executar os algoritmos de otimização sobre a função de custo escolhida, tornando o resultado gerado pela rede cada vez mais próximo da "predição perfeita"(DATA SCIENCE ACADEMY, 2019).

O problema de otimização pode ser caracterizado na matemática como um problema de minimização, onde a função de custo sera minimizada através da troca dos parâmetros.

Para que o mínimo global seja alcançado, vários métodos podem ser utilizados, o mais usado na literatura consiste no cálculo de derivadas. A derivada de um determinado ponto em uma função unidimensional expressa o vetor de mudança da dada função naquele ponto. Com essa informação se torna possível calcular qual a direção e intensidade do movimento.

$$\frac{dt}{dy} = \lim_{h \to 0} \frac{f(t+h) - f(t)}{h} \tag{2.1}$$

A fórmula acima é a representação matemática da derivada em uma função unidimensional. No caso de redes neurais, temos múltiplas entradas da rede, então as derivadas de cada entrada são chamadas de derivadas parciais, o vetor de direções resultante desse calculo de derivada é chamado de gradiente. Utilizando o gradiente encontrado é possível encontrar a melhor direção que o modelo deve seguir para convergir para o melhor resultado possível.

Uma vez que foram realizados os cálculos da derivada, é necessário utilizar este valor para atualizar os parâmetros(pesos) do modelo. Um método muito utilizado para essa atualização é o método de gradiente descendente (GRUS, 2015).

A ideia do gradiente descendente é atualizar os pesos e bias a cada iteração sobre os dados. O primeiro passo da etapa de treinamento consiste em definir uma variável chamada de taxa de aprendizado, que representa a intensidade a qual os pesos serão atualizados, em outras palavras, o quão rápido o modelo aprende. Embora seja contra intuitivo, uma taxa de aprendizado muito grande pode não ser o melhor valor, podendo levar o modelo a problemas de convergência.

O algorítimo do gradiente descendente funciona da seguinte forma:

- Calcula-se o gradiente

$$gradiente = derivada(j(x), i, w)$$

  onde:

  - *j(x)* é a função de custo
  - i é o vetor de entrada

– x é o vetor de pesos

- e depois atualiza-se os pesos, utilizando o gradiente calculado anteriormente

$$peso_k = peso_k - (learning\_rate * gradiente)$$

Um ponto importante a ser mencionado é o sinal negativo na formula de atualização dos pesos, uma vez que os sinais positivos e negativos devem ser invertidos para que os pesos sejam atualizados na direção correta.

2.1.3.1 Backpropagation

A técnica de Backpropagation ocupa um espaço muito importante na fase de otimização de uma rede neural, uma vez que ela realiza o aprendizado na rede. A técnica é aplicada no final de cada etapa do treinamento. Após a predição ser realizada pela rede, o erro entre o valor esperado e o obtido é utilizado com o gradiente da saída gerada pela camada final. Então, através da regra da cadeia, os pesos e bias das camadas anteriores são atualizados. Essa calibração dos pesos e bias materializam o aprendizado na rede (DATA SCIENCE ACADEMY, 2019).

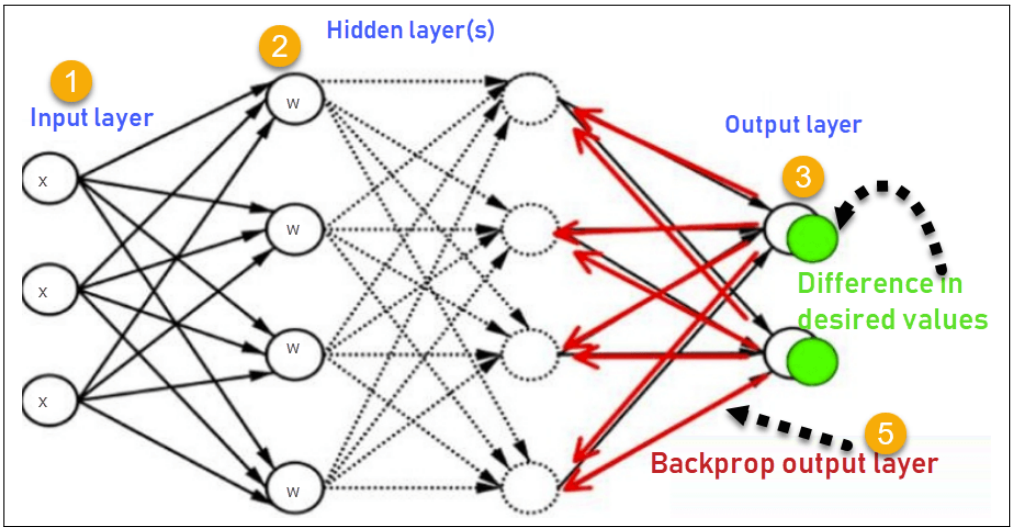


Figura 6 – Funcionamento do backpropagation

Na Figura 6, as setas vermelhas mostram o início do caminho das atualizações dos pesos e bias, seguindo da mesma forma nas camadas seguintes.

2.2 Deep Learning

Redes neurais são compostas geralmente por três camadas: uma camada de entrada, uma camada de saída e uma camada intermediária, responsável pelo processamento dos

dados. O cérebro humano, por outro lado, funciona como se possuísse diversas camadas, como no exemplo dado por Marr (2020): *"Partes diferentes do cérebro humano são responsáveis por processar diferentes partes dos dados, e essas partes são ordenadas hierarquicamente, ou em camadas. Dessa forma, enquanto a informação vai entrando no cérebro, cada camada de neurônio processa os dados e gera informação, e subsequentemente passa essa informação para a próxima camada. Por exemplo, quando você sente o cheiro de pizza de uma pizzaria no outro lado da rua, seu cérebro processa o cheiro em múltiplos estágios: 'sinto cheiro de pizza'(Dados de entrada)... 'Eu adoro pizza'(Pensamento)...'Eu vou comprar um pedaço de pizza'(Tomando uma decisão)...'Eu prometi que não comeria mais besteiras'(Memória)...'Um pedaço não mata ninguém né?'(raciocinando)...'Vou comprar um pedaço de pizza!'(ação)"*

Logo uma rede neural com camadas profundas(deep learning) possui múltiplas camadas intermediárias. O termo "deep" geralmente se refere ao número de camadas da rede, uma vez que tradicionalmente redes neurais comuns tem de uma a três camadas intermediárias, com uma rede no estilo deep learning podendo chegar a mais de 150 camadas (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.2.1 Redes Neurais Convolucionais

Redes neurais convolucionais (RNCs) são um subgrupo das redes neurais que tem se provado muito eficazes em áreas relacionadas a reconhecimentos de imagem e classificação.

O nome "Redes Neurais Convolucionais" vem do fato que a rede emprega uma operação matemática chamada convolução. Uma rede Convolucional é uma rede neural que utiliza a convolução no lugar da multiplicação de matriz em pelo menos uma de suas camadas (GOODFELLOW; BENGIO; COURVILLE, 2016).

Uma rede neural convolucional conta, como citado anteriormente, com algumas camadas com funções especiais, sendo elas:

- Camada Convolucional;
- Camada de Pooling;
- Camada totalmente conectada.

A camada convolucional funciona da seguinte forma: uma imagem não e nada além de uma matriz de *pixels*, e os *pixels* são nada mais do que números representando cores.

O algoritmo passa diversos filtros previamente treinados pela rede(também conhecidos como kernels), esses filtros são responsáveis por captar características da imagem, e cada filtro gera uma nova matriz, que ressalta as características procuradas por cada filtro, como curvas, pontas, cores e outros padrões (Lawrence et al., 1997).

A camada de Pooling é a camada responsável por diminuir o tamanho das matrizes geradas pelo algoritmo da camada convolucional, porém essa compressão da matriz deve

ser feita ressaltando ainda as características capturadas pelo filtro. Outra característica importante encontrada na camada de Pooling é a capacidade de suprimir ruído da imagem, tornando mais evidente as características que são procuradas.

Como apresentado na Figura 7, podem haver mais de uma camada Convolucional e de Pooling na rede

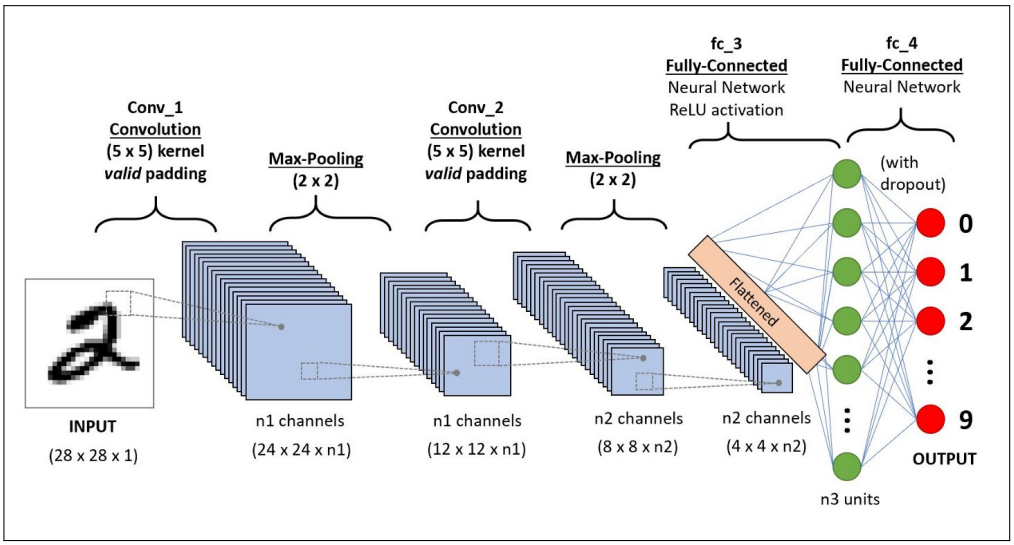


Figura 7 – Sequência de camadas em uma RNC que reconhece números escritos a mão

Na Figura 7 o reconhecimento de imagem passa por todas as etapas citadas anteriormente. Nas etapas marcadas como *convolution*(convolução) a rede esta buscando por traços da imagem, como curvas, retas, áreas sombreadas, e quaisquer outras características que a rede achou relevante para o reconhecimento de dígitos durante seu treinamento. E também pode-se notar que nas etapas de *pooling* representadas na imagem, a imagem do digito vai sendo diminuída, porem de forma que mantenha suas características. E ao final, a imagem é achatada(*flattened*) servindo de entrada para a camada totalmente conectada

A camada totalmente conectada atribui um peso para cada característica observada, de acordo com o seu valor para ajudar a reconhecer a imagem, com isso feito, a rede pode nos dizer a probabilidade da imagem conter o objeto desejado.(GOODFELLOW; BENGIO; COURVILLE, 2016)

2.2.2 Redes Neurais Recorrentes

Redes Neurais Recorrentes (RNN) podem ser interpretadas como redes neurais com uma espécie de memória, não dependendo apenas da entrada atual, mas também do resultado anterior para gerar um novo resultado (JAIN; MEDSKER, 1999).

Entre as aplicações que utilizam redes neurais recorrentes estão:

- Reconhecimento de fala;
- Predição do mercado financeiro;
- Predição de próxima palavra (teclados de celular).

A Figura 8 mostra uma abstração do comportamento de um RNN simples, onde a rede recebe como entrada, não só os dados externos, como o resultado gerado anteriormente.

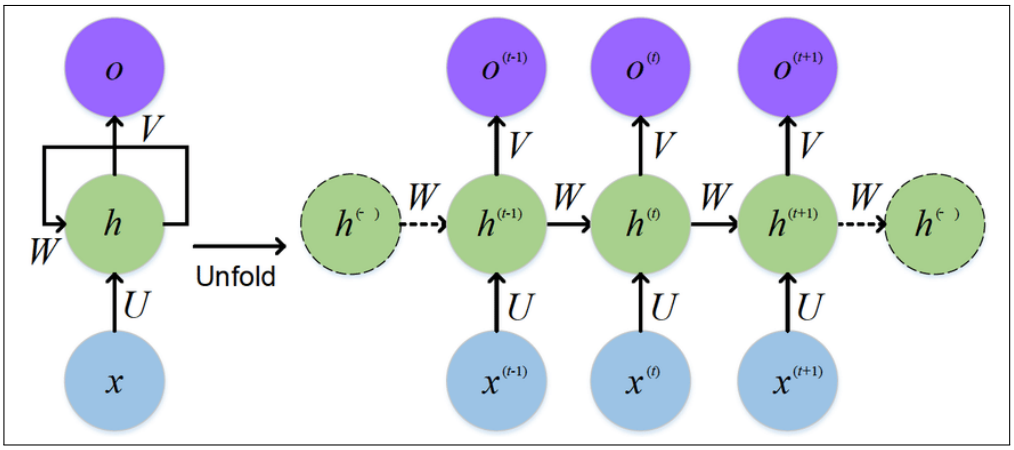


Figura 8 – Desdobramento de uma Rede Neural Recorrente

2.3 Métricas de Análise de resultados

Após o treinamento de redes neurais, é preciso de métricas para que a rede possa ser analisada.

As redes são analisados da seguinte forma:

- Acurácia: Quantidade total de segmentos avaliados corretamente, tanto Verdadeiros positivos quanto Verdadeiros negativos, sobre o numero total de testes, de modo que

$$Acurácia = \frac{VP + VN}{VP + VN + FP + FN}$$

  Acurácia deve ser avaliada com um certo cuidado, pois não captura visão se a rede tem o comportamento esperado.

- Precisão: Quantidade de testes classificados como positivos que realmente são positivos, de modo que

$$Precisão = \frac{VP}{VP + FP}$$

- Recall: quantidade de segmentos positivos que foram classificados corretamente. É importante para garantir que a classe positiva que é minoritária também terá uma boa taxa de acerto. Para esse trabalho essa é uma métrica muito importante.

$$Recall = \frac{VP}{VP + FN}$$

2.4 Frameworks e Bibliotecas

Nesta seção, são apresentadas as principais ferramentas que serão utilizadas no desenvolvimento do trabalho.

2.4.1 TensorFlow

Lançado ao público em fevereiro de 2017, TensorFlow é uma biblioteca de código aberto criada pela equipe Brain da Google utilizada para computação numérica e *machine learning*. A biblioteca agrupa uma grande quantidade de modelos e algoritmos para *machine learning* e *deep learning* e o disponibiliza de um modo abstraído, permitindo maior facilidade para o desenvolvimento. TensorFlow possui uma API front-end em python, porem o framework é executado em C++ para que seja possível uma melhor performance de execução (TENSORFLOW, 2020).

2.4.2 Keras

O framework TensorFlow adotou Keras como API de alto nível desde a sua versão 2.0 Keras fora criada para ser *user friendly*, modular e fácil de escalar em tamanho. Ela foi pensada para ser de fácil compreensão para seres humanos.

Keras por si só não possui uma *engine* para realizar suas operações em baixo nível, como produto de matrizes, tensores e convoluções. para isso ele conta com um back-end de terceiros. Embora Keras suporte diversas engines, a sua engine principal e padrão é o TensorFlow.(KERAS, 2020)

2.4.3 Spark

Pensado para ser um framework com o proposito de um processamento eficiente de *big data*, o spark foi criado em 2009 pelo AMPLab da universidade da Califórnia e logo em 2010 seu código foi aberto como um projeto da fundação apache.

A principio, o framework spark possui uma API de fácil compreensão para o gerenciamento de *big data* de diversas naturezas, como textos, grafos, imagens, entre muitos outros, além de também suportar dados de diferentes origens como dados em batch ou streaming, ou seja, dados em tempo real.(SPARK, 2020)

2.4.4 PyTorch

PyTorch é uma biblioteca criada em 2016 e desenvolvida pela FAIR(*Facebook's AI Research*), foi escrita utilizando-se python, C++ e CUDA, sendo pensado seu uso em conjunto com a linguagem de desenvolvimento python. Assim como o TensorFlow, PyTorch tem objetivos similares, facilitar o desenvolvimento de redes neural. Embora não

tao popular quanto o seu concorrente TensoFlow, o PyTorch vem ganhando bastante espaço, principalmente pelo seu suporte ao uso de placas gráficas para treinamento e uso das redes neurais.(PYTORCH, 2020)

Este framework foi utilizado no trabalho, pelo seu bom desempenho e pelo grande suporte encontrado na internet.

3 Trabalhos Relacionados

Os trabalhos aqui apresentados foram escolhidos após uma pesquisa por artigos relacionados a detecção de armas de fogo no Google Schoolar. Vale a pena mencionar que apenas três trabalhos foram apresentados aqui por perceber que o reconhecimento de armas se dava em duas situações. A primeira se tratava da detecção de armas de fogo em câmeras de raio-X, o que para o propósito deste trabalho não se aplica. A segunda situação, é a detecção de armas de fogo em câmeras de vigilância. Então, optou-se por trabalhos mais relevantes relacionados a segunda situação.

3.1 A Computer Vision based Framework for Visual Gun Detection using Harris Interest Point Detector

Os autores (TIWARI; VERMA, 2015) apresentam a baixa eficiência e os desafios enfrentados por operadores de redes de câmeras de segurança, afirmando que 95% de eficiência é perdida após apenas 22 minutos de monitoramento contínuo, fazendo que a vigilância fique severamente prejudicada.

Os autores sugerem uma abordagem utilizando segmentação baseada em cores para eliminar cores e objetos que não são de interesse para identificação, *Harris Interest Point Detector*, que consiste em detectar certos ângulos importantes na imagem, e um descritor de pontos chave FREAK (Fast Retina Keypoint).

Embora estes ângulos detectados representem uma porcentagem muito pequena da imagem, eles contem as características mais importantes na detecção, em conjunto do descritor de pontos chave FREAK (Fast Retina Keypoint), que interpreta os pontos encontrados pelo método anterior.

Para o teste foi utilizado um *dataset* criado pelos próprios autores, composto por 65 imagens positivas, onde armas estão presentes na imagem e 24 imagens negativas, nas quais armas não estão presentes na imagem. O *dataset* passou por um pré-processamento responsável por remover ruídos da imagem.

A fase de testes foi dividida em seis etapas, cada uma delas abordando diferentes características das imagens.

1. Reconhecimento de armas de fogo em diferentes *backgrounds*.
2. Reconhecimento de armas de fogo em diferentes graus de iluminação.

3. Reconhecimento de armas de fogo com variação das armas reconhecidas.
4. Reconhecimento de armas de fogo em diferentes graus de oclusão da arma.
5. Reconhecimento de armas de fogo com variação na escala e rotação da arma.
6. Reconhecimento de múltiplas armas de fogo ao mesmo tempo

A métrica utilizada para medir a eficiência do *framework* é dado pela fórmula

$$Accuracy = \frac{I_{PD} + I_{NU}}{I_{PD} + I_{PU} + I_{NU} + I_{ND}}, \tag{3.1}$$

na qual I_{PD} é o numero de imagens positivas nas quais armas foram detectadas, I_{PU} é o numero de imagens positivas nas quais armas não foram detectadas, I_{ND} é o numero de imagens negativas nas quais armas foram detectadas e I_{NU} é o numero de imagens negativas nas quais armas não foram detectadas.

Etapas	**Numero de imagens positivas**	**Numero de imagens corretamente classificadas**	**Taxa de Positivos verdadeiros**
Etapa 1	12	11	91,66%
Etapa 2	9	7	77,77%
Etapa 3	11	9	81.81%
Etapa 4	17	14	82.35%
Etapa 5	10	8	80%
Etapa 6	6	5	83.33%

Tabela 1 – Resulados obtidos com diferentes experimentos (TIWARI; VERMA, 2015)

O Resultado do trabalho foi um sistema de detecção com eficácia geral de 84.26%, porém com certos problemas em imagens com mudanças na iluminação, onde os autores notaram que houve uma queda significativa na acurácia.

3.2 Gun Detection in Surveillance Videos using Deep Neural Networks

Os autores (LIM et al., 2019) apontam o quão suscetível a falhas humanas os sistemas de vigilância por câmeras são, sendo relatados até mesmo jogos de esconde esconde entre os operadores das câmera e os seguranças. Visando resolver este problema, foi proposto a utilização de uma rede neural profunda, capaz de alertar o operador das câmeras de vigilância para possíveis perigos, no caso deste trabalho em especifico, armas de fogo. Buscando a melhor opção para a rede neural, diversas redes foram cogitadas, levando em consideração os seguintes parâmetros:

- Ser capaz de realizar o reconhecimento em tempo real;
- Uso de memória reduzido;
- Uso de poder computacional reduzido.

A seguir algumas análises das topologias realizadas pelos autores:

- Mobilenet-SSD: Uma rede que usa convoluções separáveis em profundidade para detecção em várias escalas, permitindo que a Mobilenet-SSD detecte objetos com precisão e em tempo real
- Faster R-CNN: Embora alcance uma precisão maior na detecção do que a M2Det, é computacionalmente muito pesada devido ao processo de reconhecimento em dois estágios.
- RefineDet: Também uma arquitetura de detecção em dois estágios, porém utilizando-se da técnica de regressão em cascata, com isso alcançando uma precisão maior do que a rede Faster R-CNN, porem mantendo a eficiência da Mobilenet-SSD
- M2Det: Detector de objetos em um estagio baseado em uma rede em piramide multi-nível, capaz de superar limitações de outras redes de um e dois estágios, demonstrando uma maior precisão e eficiência no uso de recursos computacionais em relação as outras redes aqui citadas, sendo esta a rede escolhida pelos autores.

Após pesquisarem diversos *datasets* existentes os quais continham imagens da perspectiva de câmeras de segurança ficou claro para os mesmos que não havia informação o suficiente para o devido treinamento da rede.

O dataset UCF Crime consiste em vídeos de vigilância retirados do YouTube e LiveLeak, mostrando prisões, ataques, roubos, brigas, tiroteios e diversas outras situações. Os vídeos deste *dataset* foram todos reduzidos para 240 $\times$ 320 *pixels* com 30 *fps*. Entretanto, nem todos os vídeos do *dataset* possuíam a presença de uma arma e apenas 26 vídeos tinham uma representação clara de uma arma de fogo, com todos os outros vídeos possuindo apenas representações fracas de uma arma de fogo, causada principalmente pela baixa qualidade dos vídeos, devido a dupla compressão ao gravar e ao realizar o *upload* para o site. Sendo apenas o UCF Crime *dataset* insuficiente para o treinar a rede com precisão.

Com esse fim a Universidade de Granada construiu um *dataset* que consiste em 3000 imagens ricas em contexto, porem, embora seja um *dataset* bem variado, apenas 48 imagens de todas as 3000 são da perspectiva de uma câmera de segurança, o que torna este *dataset* também insuficiente para o treinamento eficiente da rede para o proposito dos autores.

A solução encontrada pelos autores foi construir seu próprio dataset com imagens de perspectiva de câmeras de vigilância. Utilizando de câmeras de segurança em diversos ambientes(indoor e outdoor) com diversos tipos de iluminação, com cenas gravadas durante o dia, tarde e noite. Com isso os autores simularam situações empunhando uma réplica de pistola na cor preta.

Dataset	# de Frames	# de Frames emcontexto CCTV	Visão da arma
Universidade de Granada	3000	48	Visão frontal e lateral da arma
UCF Crime	7247	419	Imagem embaçadade CCTV
Dataset dos autores	5500	5500	Visão clara daarma de diversospontos de vista

Tabela 2 – Comparação entre os datasets apresentados no artigo (LIM et al., 2019)

Utilizando 45 frames do dataset UCF Crimes como baseline dos testes para determinar a precisão do sistema, comparando com o treinamento realizado com o dataset dos proprios autores com o dataset da Universidade de Granada.

Dataset	Precisão média IoU 0.5:0.95 - 0.5 - 0.75	Precisão média, Área S - M - L
Universidade de Granada	0.114 - 0.281 - 0.053	0 - 0.110 - 0.800
Dataset dos autores	0.223 - 0.442 - 0.202	0.180 - 0.224 - 0.717

Tabela 3 – Resultados obtidos comparando os 2 datasets para treinamento (LIM et al., 2019)

Como é possível observar, tanto a media de IoU (*intersection over union*) quanto a media de área se mostraram superiores quando a rede era treinada utilizando-se o *dataset* dos autores, com exceção da precisão da área em imagens de alta qualidade, que se mostrou ligeiramente inferior.

Após todo o trabalho desenvolvido os autores criarem um detector de objetos utilizando o modelo M2Det como detector de objetos, tendo como diferencial o *dataset* desenvolvido pelos mesmos, visando o caso especifico de câmeras de vigilância e com isso tornando a rede muito mais eficiente.

3.3 A Handheld Gun Detection using Faster R-CNN Deep Learning

Afim de implementar a Faster-CNN, uma arquitetura com o objetivo de detecção de objetos, os autores (VERMA; DHILLON, 2017) adotaram o *framework* MatConvNet. Ele auxilia no desenvolvimento de redes neurais convolucionais.

Os autores optaram pela rede VGG-16, uma rede neural convolucional com 16 camadas de profundidade. Além das facilidades do *framework*, ele também oferece uma rede pré-treinada capaz de classificar 1000 categorias de objetos, como teclados, mouses, canetas e diversos animais.

Para treinamento da rede, os autores optaram por utilizar o dataset Internet Movie Firearms DataBase (IMFDB), que é composto por imagens de armas de fogo retiradas de filmes, video games, programas de TV e animações, sendo as imagens dividas por categorias como: pistolas, revolveres, fuzis de assalto, etc. Os autores optaram por treinar

o sistema utilizando apenas imagens revolveres, fuzis e espingardas, enquanto para as imagens negativas foram utilizadas imagens aleatórias de diversas categorias como flores e animais.

A Figura 9 mostra exemplos de imagens positivas enquanto a Figura 10 mostra exemplo de imagens negativas.

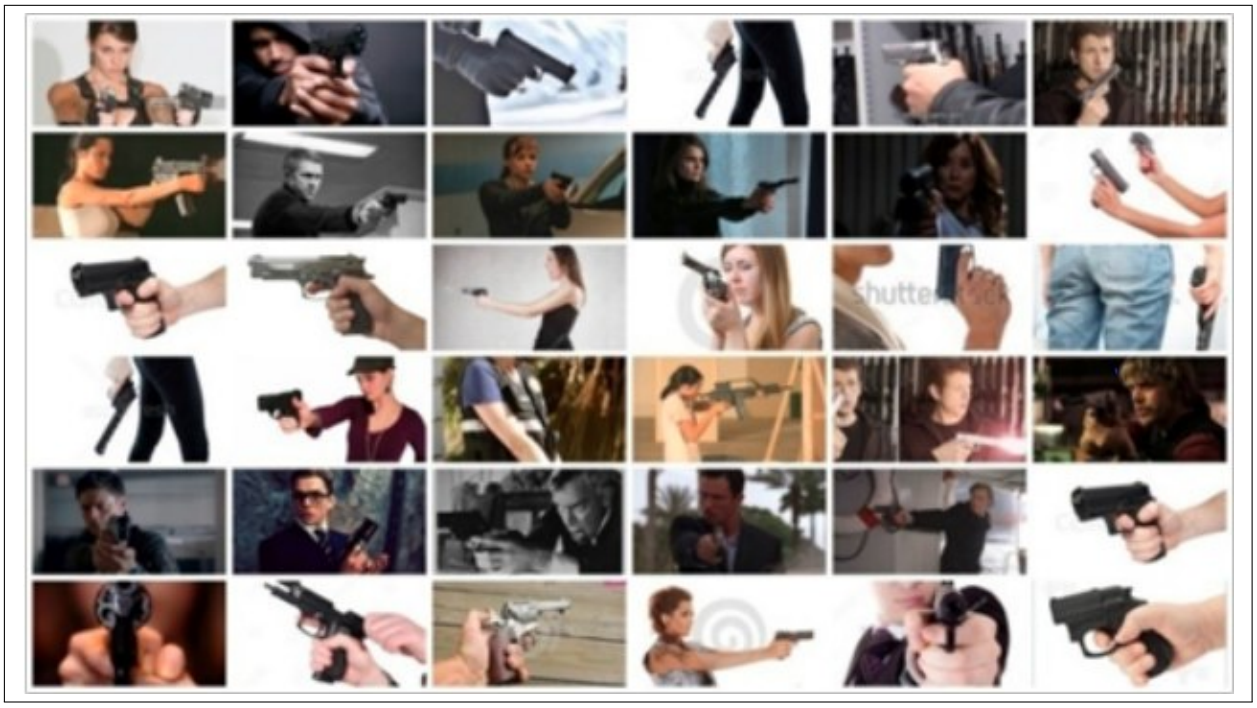


Figura 9 – Imagens Positivas (VERMA; DHILLON, 2017)

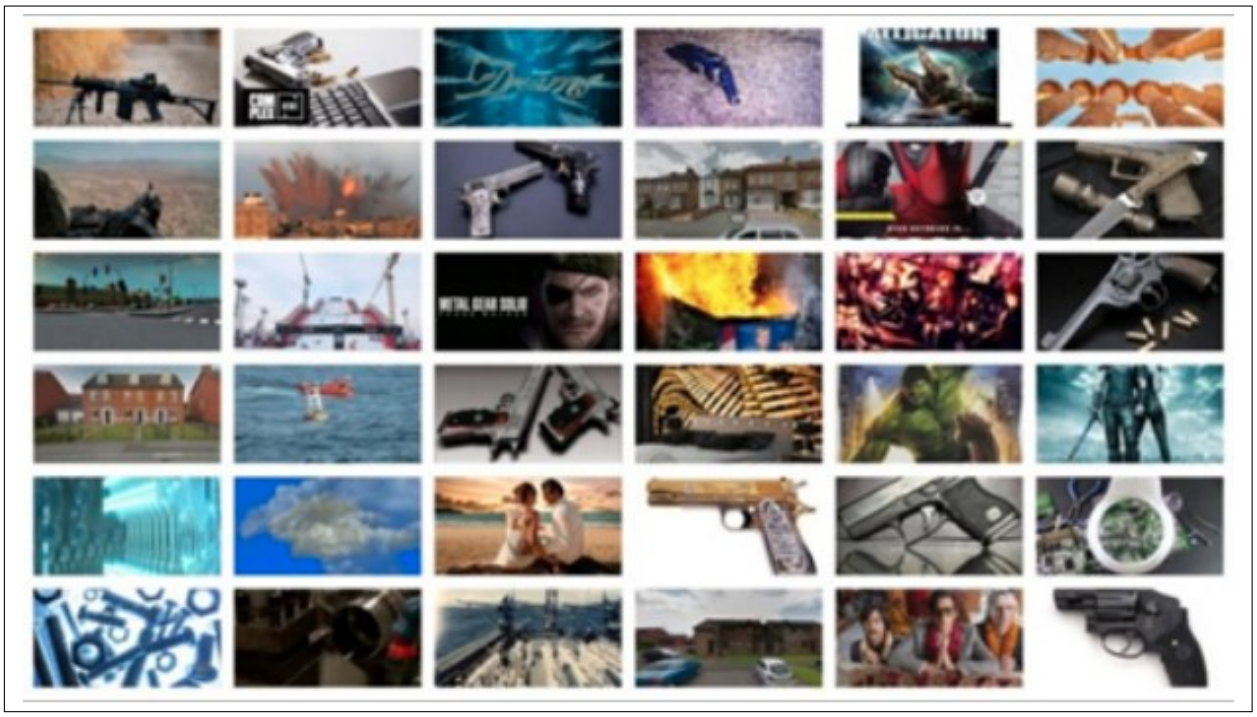


Figura 10 – Imagens Negativas (VERMA; DHILLON, 2017)

É possível notar pelas Figuras 9 e 10 que os autores estão interessados em detectar apenas pessoas portando armas de fogo, uma vez que imagens de armas de fogo isoladas sem qualquer contexto estão no grupo de imagens negativas. Para avaliar a performance do sistema, os autores utilizaram de três métricas, sendo estas: taxa de verdadeiro positivo, taxa de falso positivo e acurácia. A taxa de verdadeiro positivo é dada pelo percentual de imagens positivas as quais foram corretamente analisadas pelo sistema, sendo expressada na equação abaixo:

$$TRP = \frac{TP}{TP + FN}. \tag{3.2}$$

No qual, TRP é a taxa de casos verdadeiro positivos, TP é o numero de verdadeiros positivos e FN é o numero de falso negativos no sistema.

Enquanto a taxa de falso positivo é determinado pelo percentual de imagens negativas avaliadas incorretamente pelo sistema.

E a acurácia é a proporção do total de imagens analisadas corretamente pelo sistema em relação a todas as imagens analisadas, dando uma visão geral do funcionamento da rede.

Os autores testaram a performance do sistema utilizando-o em diversos cenários, e diferentes ângulos e com oclusão da arma de fogo, sendo avaliado através de acurácia, taxa de verdadeiro positivo e taxa de falso negativo, predição de valor positivo e predição de valor negativo. Após testarem o sistema com diferentes classificadores, sendo estes Support Vector Machine (SVM), K-Nearest Neighbor (KNN) e Ensemble Tree, verificou-

se que o classificador Ensemble Tree foi o que demonstrou melhores resultados tendo chegado a 93.1% de precisão do sistema.

3.4 You Only Look Once: Unified, Real-Time Object Detection

You Only Look Once (REDMON; FARHADI, 2018) também conhecido como YOLO é um algoritmo de detecção de objetos em tempo real mais rápido em comparação com a família R-CNN (R-CNN, Fast R-CNN, Faster R-CNN, etc .), algoritmos já apresentados em outros trabalhos relacionados.

O algoritmo YOLO pode ser dividido em duas partes principais: O extrator de características e o detector de objetos, ambos multi-escala. Quando o sistema recebe de entrada uma imagem, ela primeiro passa pelo extrator, para que características sejam obtidas em escalas variadas e após a obtenção das características, as mesmas alimentam diferentes ramos do detector para detectar os objetos e suas respectivas classes.

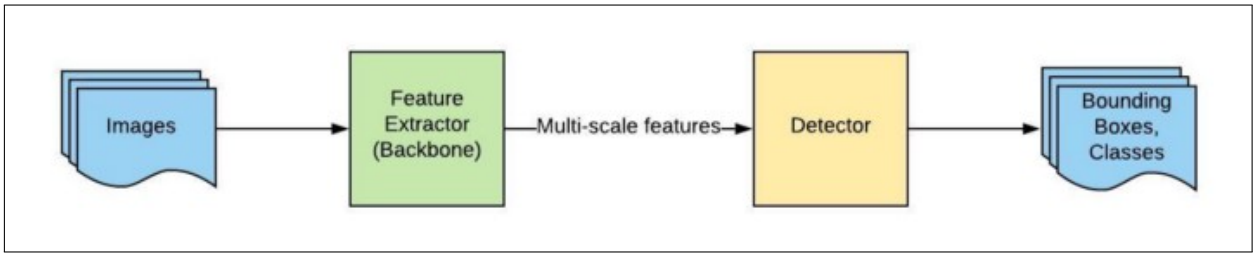


Figura 11 – Abstração da arquitetura YOLO (SCIENCE, 2020)

O Algoritmo YOLO divide a imagem de entrada em uma matriz SxS, cada campo da matriz faz N predições e calcula o score P, probabilidade que indica se existe ou não um objeto ali. Então cada campo da matriz também prediz a probabilidade condicional do objeto pertencer a determinada classe, ou seja, definir a que classe o objeto pertence, como mostra a Figura 12

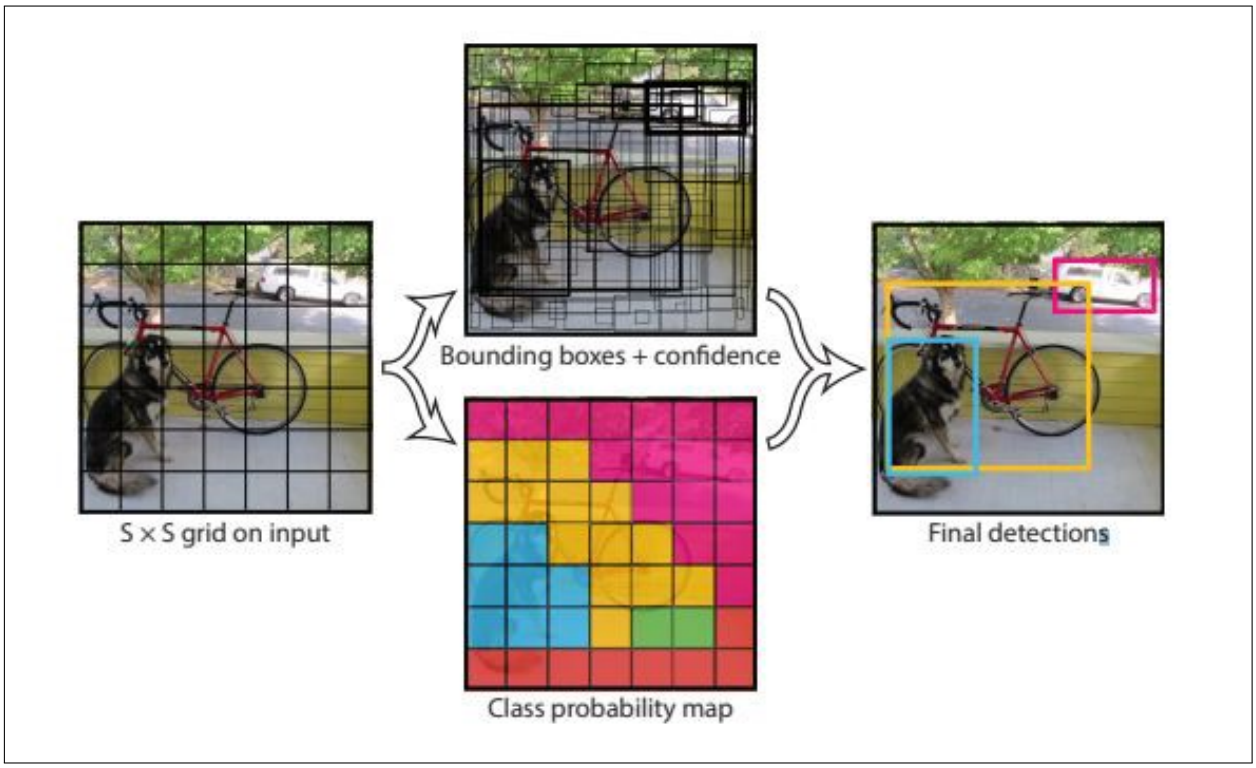


Figura 12 – Conceito YOLO (REDMON et al., 2015)

A rede neural convolucional do YOLO possui 24 camadas convolucionais que são responsáveis por extrair características da imagem de entrada, sendo imediatamente seguidas por duas camadas totalmente conectadas, responsáveis pela classificação e a detecção. A saída da rede é um tensor no formato 7x7x30 como mostrado na Figura 13

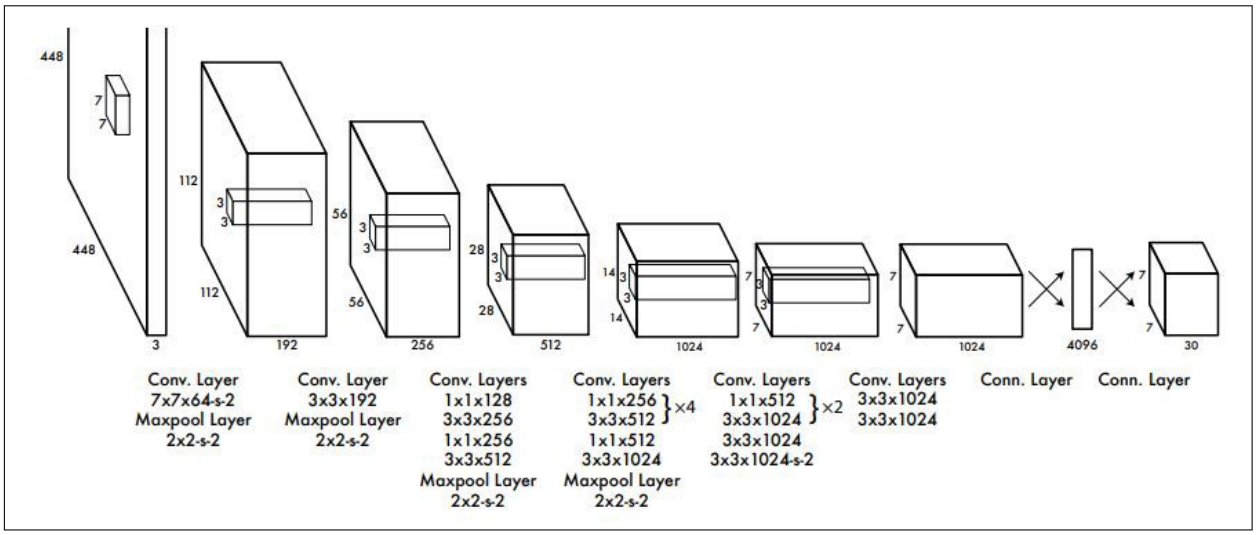


Figura 13 – RNC YOLO (REDMON et al., 2015)

4 Especificação e Métodos

Neste capítulo, são especificados alguns pontos do trabalho como: o tipo de rede que será utilizada, os experimentos utilizados para testar as redes treinadas.

4.1 Datasets

O dataset da Universidade de Granada, citado anteriormente em um trabalho relacionado, foi escolhido para compor o treinamento da rede, pois contem imagens de armas de fogos em diferentes contextos, como sobre uma mesa, no coldre ou sendo segurada por uma pessoa.

Para complementar o treinamento, são utilizadas algumas imagens escolhidas aleatoriamente de outros datasets, fornecendo ao treinamento exemplos de imagens onde armas de fogo não estão presentes.

4.2 Especificação da rede que será utilizada

Inspirando-se nos trabalhos relacionados, a rede inicialmente escolhida para a realização do projeto foi a Rede Neural Convolucional YOLOv3, um detector de objetos em um estágio feito já pensando em reconhecimento de objetos em tempo real, uma melhoria do já citado YOLO, quem vem demonstrando desempenho superior a outras redes já citadas anteriormente.

4.3 Roteiro de experimentos

Os experimentos se resumem em testar a rede com uma base de testes, em que previamente se tem conhecimento se a imagem possui ou não uma arma de fogo. Após a realização dos testes, se obtém os números de Verdadeiros Positivos, Falsos Positivos, Verdadeiros Negativos e Falsos Negativos, para avaliação descrita na próxima seção do trabalho.

4.4 Análise de resultados: como será realizada a análise

Após o treinamento da rede desenvolvida com imagens contendo armas de fogo em diferentes contextos e a realização de testes, para obtenção de métricas citadas anterior-

mente, com *datasets* também contendo armas de fogo em diferentes contextos porém, que não participaram do treinamento da rede.

Para comparar os resultados serão utilizadas as métricas descritas na seção 2.3

4.5 Treinamento e Resultados

Nesta seção, são apresentadas informações relativa a coleta de dados, treinamento e resultados obtidos durante a realização do trabalho.

4.5.1 Dados de treinamento

O treinamento consistiu em diferentes combinações das imagens de indivíduos portando armas de fogo em diferentes ângulos e também armas soltas. As imagens foram retiradas do dataset de armas de fogo da Universidade de Granada e foram classificadas utilizando o software de código aberto chamado CVAT.

4.5.2 Dados de teste

As imagens de treinamento e teste foram divididas em dois conjuntos, para posterior verificação dos resultados obtidos através de métricas como precisão e acurácia. O conjunto de teste contentando aproximadamente 3645 imagens, e o conjunto de teste 200 imagens.

4.5.3 Treinamento

Um total de oito experimentos foram realizados, alterando as imagens, alterando a combinação de imagens e o número de épocas (numero de vezes que o treinamento utilizada o conjunto de imagens) que foram utilizadas.

Todos os pesos gerados pelo treinamento com as imagens de arma de fogo utilizaram como base os pesos fornecidos pelo repositório original do projeto YOLOv3 que é obtido automaticamente ao executar o projeto pela primeira vez.

Os treinamentos foram realizados em uma máquina com as seguintes especificações.

- Processador: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
- Memória RAM: 32 GB
- GPU: NVIDIA GeForce GTX 1660 Ti 6 GB

4.5.4 Data Augmentation

Data Augmentation é uma técnica utilizada quando se há poucos dados para treinamento e também para evitar um erro chamado overfitting, que ocorre quando o modelo se adaptou muito bem aos dados com os quais está sendo treinado; porém, não generaliza bem para novos dados.

A técnica consiste em alterar diversas vezes uma imagem, utilizando a original e todas as alterações geradas no treinamento dos pesos da rede. (DYK; MENG, 2001)

A Figura 14 mostra um exemplo do resultado da técnica utilizada.

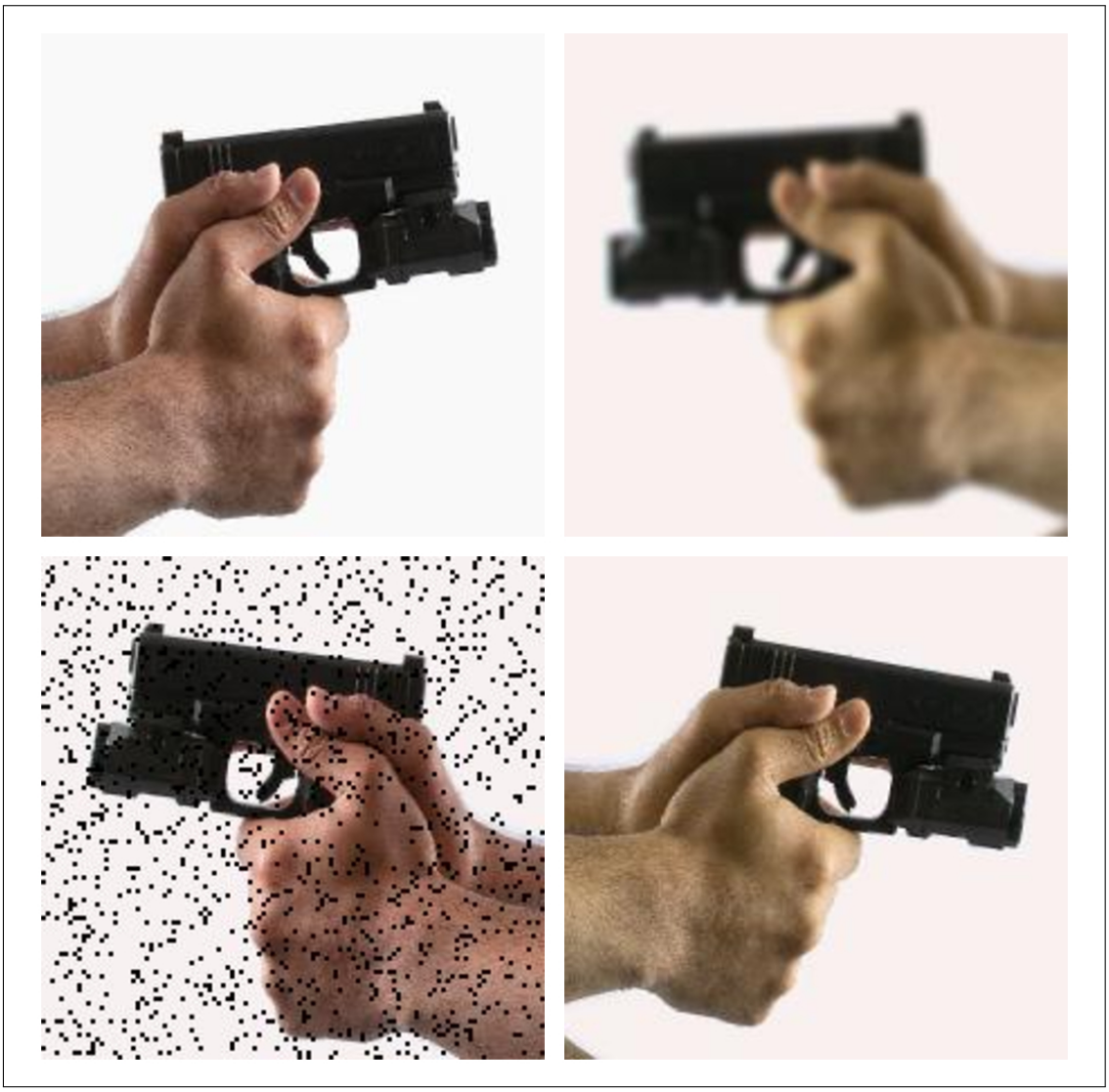


Figura 14 – Imagens geradas pela técnica de Data Augmentation

4.5.5 Resultados do treinamento

Oito conjuntos de treinamentos foram experimentados para a rede YOLO. Dos conjuntos analisados, apenas três ensaios são destacados, por terem obtido melhores resultados.

Eles estão divididos em Treinamento 1, 2 e 3.

Os parâmetros em si também são experimentações, uma vez que o treinamento possui um ponto ótimo e não necessariamente mais imagens de treinamento geram uma rede melhor.

4.5.5.1 Treinamento 1

O Treinamento 1 operou sobre os seguintes parâmetros:

- épocas: 200
- Batch: 5
- Numero de imagens: 2187
- Tempo de treinamento: 23 horas

Neste treinamento foram utilizadas, além de imagens de armas de fogo, imagens não relacionadas, como pessoas, paisagens e figuras afim de permitir a classificação de imagens sem armas. O conjunto de treinamento contou com imagens adaptadas usando técnicas de Data Augmentation como: posições diferentes, borradas e com ruído na imagem.

Este dataset foi composto por 300 imagens tratados pela técnica de data augmentation(14%), 250 imagens que não contem armas de fogo(11,5%) e 1637 imagens com ocorrências de arma de fogo(75,5%).

As imagens das Figuras 15, 16 e 17 mostram os resultados do conjunto de teste utilizado durante o treinamento. Observando as imagens dos testes feitos durante o treinamento da rede, é possível notar que os pesos, embora reconheçam alguns das armas de fogo mostradas na imagem, possuem um nível de confiança muito baixo, deixando claro q ainda existe espaço para melhoria no treinamento dos pesos da rede. É possível notar que os ruídos na imagem impactam negativamente na detecção.

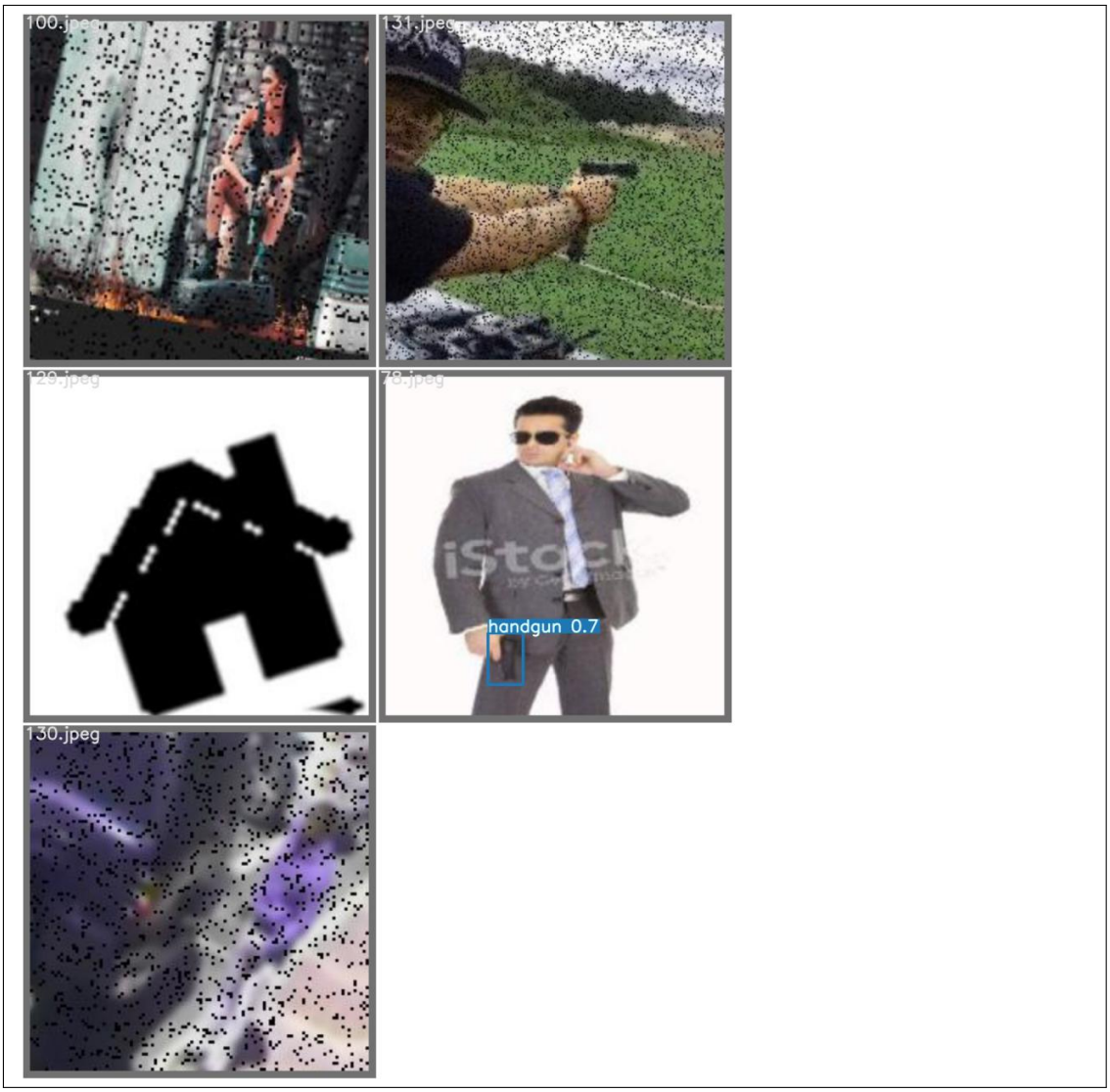


Figura 15 – Resultados do teste utilizando os pesos do treinamento 1

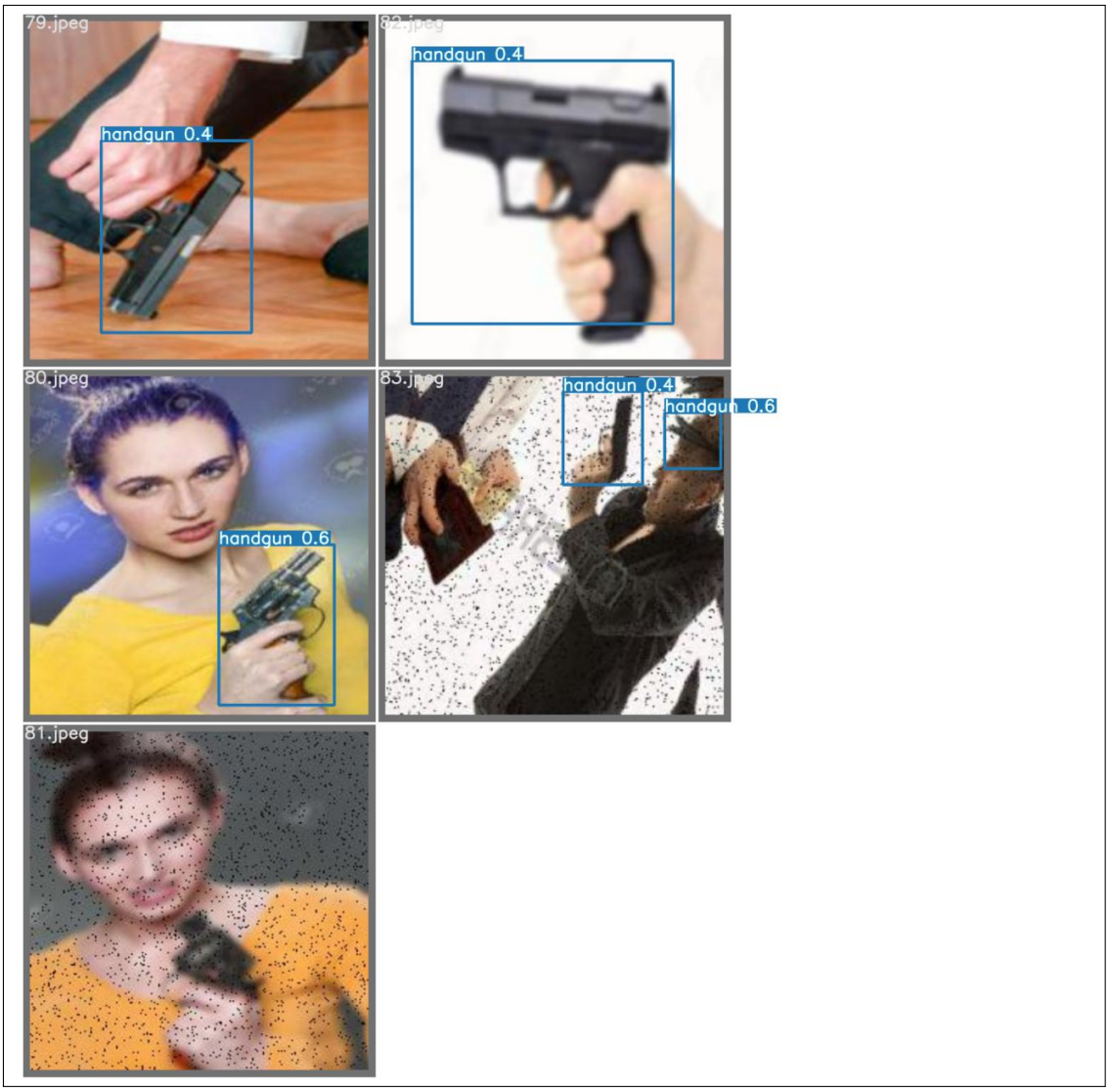


Figura 16 – Resultados do teste utilizando os pesos do treinamento 1

Figura 17 – Resultados do teste utilizando os pesos do treinamento 1

4.5.5.2 Treinamento 2

O Treinamento 2 operou sobre os seguintes parâmetros:

- épocas: 100
- Batch: 5
- Numero de imagens: 1800
- Tempo de treinamento: 18 horas

Neste treinamento, não foram utilizadas imagens beneficiadas pela técnica de Data Augmentation durante o treinamento.

Este dataset foi composto por 350 imagens que não contem armas de fogo(20%) e 1450 imagens com ocorrências de arma de fogo(80%).

As imagens das Figuras 18, 19 e 20 mostram os resultados do conjunto de testes utilizados durante o treinamento. Observando as imagens dos testes feitos durante o treinamento da rede, é possível notar uma melhora considerável na detecção e o aumento da taxa da confiança das predições. Porém, é possível notar também uma maior frequência de falsos positivos.

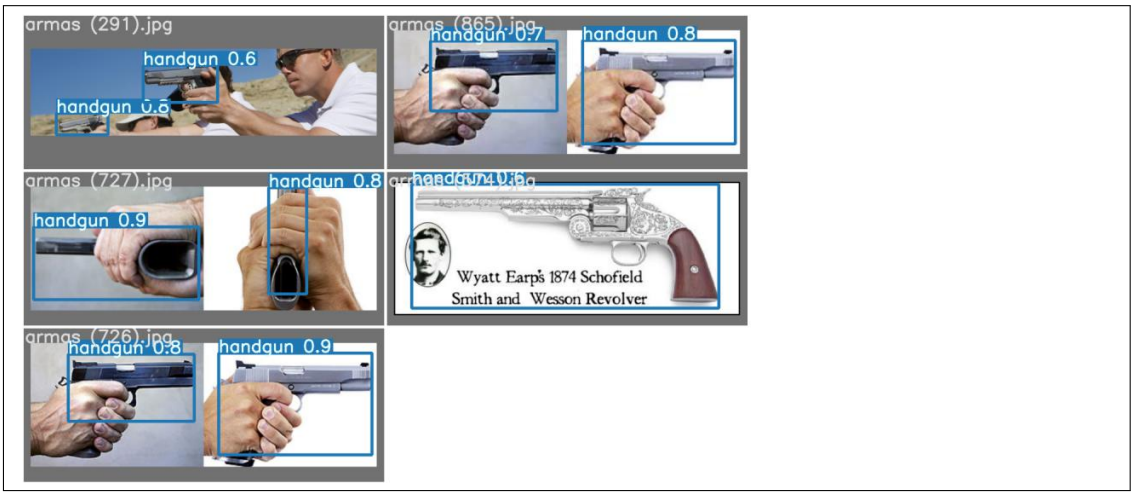


Figura 18 – Resultados do teste utilizando os pesos do treinamento 2

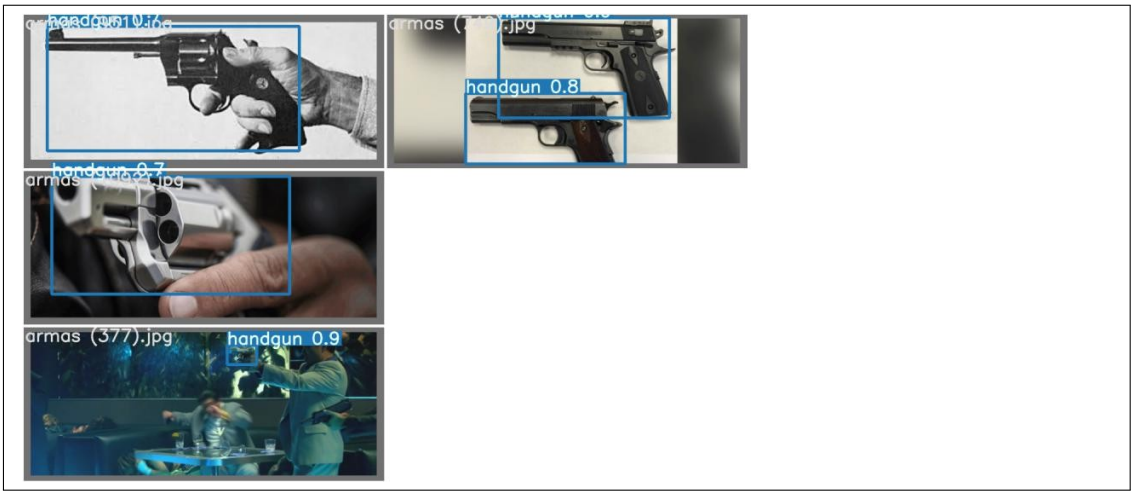


Figura 19 – Resultados do teste utilizando os pesos do treinamento 2

Figura 20 – Resultados do teste utilizando os pesos do treinamento 2

4.5.5.3 Treinamento 3

O Treinamento 3 operou sobre os seguintes parâmetros:

- épocas: 100
- Batch: 5
- Numero de imagens: 1312
- Tempo de treinamento: 20 horas

Neste treinamento, foram utilizadas apenas imagens de armas de fogo durante o treinamento.

As imagens das Figuras 21, 22 e 23 seguir mostram os resultados do conjunto de testes utilizados durante o treinamento. Embora as detecções tenham ficado melhores do que o primeiro treinamento, o segundo treinamento parece ter obtido melhores resultados.

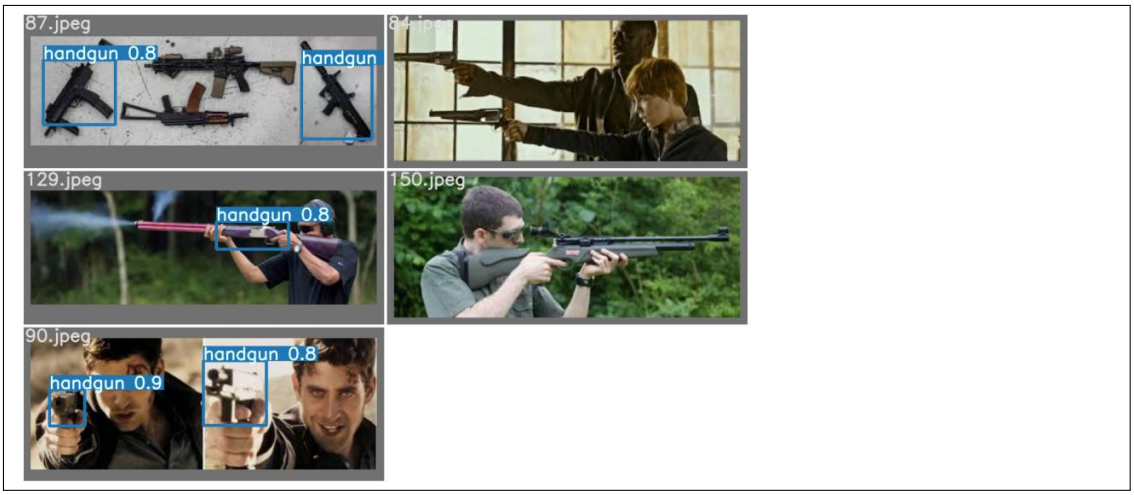


Figura 21 – Resultados do teste utilizando os pesos do treinamento 3

Figura 22 – Resultados do teste utilizando os pesos do treinamento 3

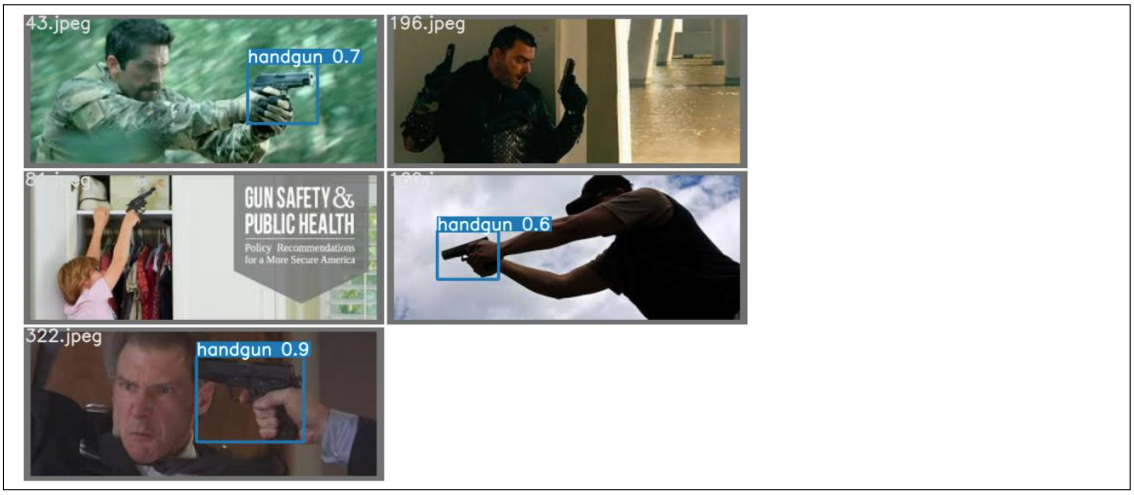


Figura 23 – Resultados do teste utilizando os pesos do treinamento 3

4.6 Teste dos pesos

Embora o treinamento tenha seu teste integrado, é difícil comparar as redes entre si, uma vez que utilizam imagens diferentes.

Visando resolver este problema, as três calibrações da rede geradas pelos experimentos apresentados anteriormente serão usados para reconhecer armas de fogo sobre um mesmo conjunto de imagens.

O conjunto consiste em 245 imagens onde sua grande maioria contendo armas de fogo, que não foram utilizadas em nem um treinamento dos pesos citados anteriormente.

4.6.1 Comparação dos treinamentos

Os dados Verdadeiro Positivo(VP), Falso Positivo(FP), Verdadeiro Negativo(VN), Falso Negativo(FN) obtidos dos pesos podem ser visualizados na Tabela 4.

	VP	**FP**	**VN**	**FN**
Treinamento 1	199	8	8	93
Treinamento 2	245	9	6	45
Treinamento 3	215	18	8	74

Tabela 4 – Resultados obtidos para a matriz de confusão em cada treinamento.

4.6.2 Acurácia

A acurácia demonstra se as armas de fogo foram classificadas corretamente e também se outros objetos não foram classificados erroneamente como armas de fogo.

Abaixo a acurácia dos treinamentos citados na seção anterior

- Treinamento 1: Acurácia $= \frac{VP+VN}{VP+VN+FP+FN} = \frac{199+8}{199+8+8+93} = 0.672 = 67{,}2\%$
- Treinamento 2: Acurácia $= \frac{VP+VN}{VP+VN+FP+FN} = \frac{245+6}{245+6+9+45} = 0.823 = 82{,}3\%$
- Treinamento 3: Acurácia $= \frac{VP+VN}{VP+VN+FP+FN} = \frac{215+8}{215+8+18+74} = 0.708 = 70{,}8\%$

Essa primeira métrica ajuda a avaliar o desempenho dos pesos calibrados durante os experimentos. Os pesos obtidos com o treinamento 2 figuraram na melhor opção. Porém a discussão dos resultados deve considerar o uso de outras métricas como Precisão e Recall (próximas seções).

4.6.3 Precisão

Precisão é uma métrica que avalia a porcentagem de predições Verdadeiros Positivos em relação a todas as predições Positivas. Abaixo a precisão dos treinamentos citados na seção anterior.

- Treinamento 1: Precisão = $\frac{VP}{VP+FP} = \frac{199}{199+8} = 0.9613 = 96{,}1\%$
- Treinamento 2: Precisão = $\frac{VP}{VP+FP} = \frac{245}{245+9} = 0.965 = 96{,}5\%$
- Treinamento 3: Precisão = $\frac{VP}{VP+FP} = \frac{215}{215+18} = 0.923 = 92{,}3\%$

Os resultados mostram que o treinamento 2 é também mais preciso do que os outros dois treinamentos, embora o treinamento 1 tenha obtido resultado semelhante.

4.6.4 Recall

Recall tem como objetivo determinar quanto das imagens que se esperavam ser Positivas foram realmente classificadas como Positivas.

Abaixo o Recall dos treinamentos citados na seção anterior.

- Treinamento 1: Recall = $\frac{VP}{VP+FN} = \frac{199}{199+93} = 0.744 = 74{,}4\%$
- Treinamento 2: Recall = $\frac{VP}{VP+FN} = \frac{245}{245+45} = 0{,}884 = 84{,}4\%$
- Treinamento 3: Recall = $\frac{VP}{VP+FN} = \frac{215}{215+74} = 0.682 = 68{,}2\%$

Novamente o treinamento 2 se destaca no valor da métrica, alcançando uma taxa de recall superior a 10% do segundo lugar.

4.6.5 Resultados

Observando as métricas de acurácia, precisão e recall, pode-se chegar à conclusão que dos três pesos apresentados, o que obteve melhor desempenho em todas as métricas foi o treinamento 2, não deixando duvida do desempenho superior em relação as outras.

5

Para reduzir crimes com armas de fogo, uma abordagem que pode ser adotada é a prevenção por meio da detecção de indivíduos portando-as. Utilizando-se de técnicas de detecções de padrões, como redes neurais artificiais, os problemas de custo e eficiência nessa aplicação podem ser reduzidos. Por exemplo, a tarefa de detecção fica para a inteligencia artificial, que avisaria um operador humano caso algo seja detectado, deixando a cargo deste tomar alguma atitude.

Este trabalho especifica um procedimento automático para detecção de indivíduos portando armas de fogo utilizando rede neural artificial profunda. Para cumprir essa tarefa, um levantamento de trabalhos relacionados foi realizado na literatura acadêmica, um conjunto de *datasets* foi identificado, e a especificação da rede neural foi realizada.

Com os experimentos realizados foi possível criar uma rede capaz de realizar detecção de armas de fogo com uma acurácia de 82,3% , uma precisão de 96,5% e uma taxa de recall de 84,5% dentro das condições apresentadas durante o treinamento da rede.

embora ainda falte refinamento tando sobre o *dataset* e sobre o treinamento da rede, a rede obtida mostra-se uma prova de conceito bem sucedida, mostrando que a detecção de objetos perigosos por sistemas de vigilância é possível

Como trabalhos futuros podemos listar:

- a otimização do *dataset* abrangendo mais casos próximos do real afim de evitar qualquer problemas causados pelo *dataset* atual.
- Expandir o numero de armas detectadas pela rede, como armas brancas e armas longas.
- Expandir o *dataset* com imagens de próprias de circuito interno de vídeos e também imagens noturnas.
- Comparar o desempenho da rede YOLOv3 com outras redes do estado da arte.

Referências Bibliográficas

BASHEER, I.; HAJMEER, M. Artificial neural networks: fundamentals, computing, design, and application. **Journal of Microbiological Methods**, v. 43, n. 1, p. 3 – 31, 2000. ISSN 0167-7012. Neural Computting in Micrbiology. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167701200002013>. 16

DATA SCIENCE ACADEMY. **Deep Learning Book**. [S.l.: s.n.], 2019. <http://deeplearningbook.com.br/uma-breve-historia-das-redes-neurais-artificiais>. Accessed: 2020-11-25. 19, 20, 21

DEEPTHI, T. et al. Firearm recognition using convolutional neural network. **International Journal of Scientific Research in Computer Science, Engineering and Information Technology**, p. 136–141, 03 2019. 13

DYK, D. A. van; MENG, X.-L. The art of data augmentation. **Journal of Computational and Graphical Statistics**, Taylor & Francis, v. 10, n. 1, p. 1–50, 2001. Disponível em: <https://doi.org/10.1198/10618600152418584>. 37

FUKUSHIMA, K. Cognitron: A self-organizing multilayered neural network. **Biological Cybernetics**, v. 20, n. 3, p. 121–136, Sep 1975. ISSN 1432-0770. Disponível em: <https://doi.org/10.1007/BF00342633>. 16

GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. **Journal of Machine Learning Research - Proceedings Track**, v. 9, p. 249–256, 01 2010. 18

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. 22, 23

GRUS, J. **Data Science from Scratch: First Principles with Python**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2015. 93-100 p. ISBN 149190142X. 20

HIJAZI, S.; KUMAR, R.; ROWEN, C. Using convolutional neural networks for image recognition by. In: . [S.l.: s.n.], 2015. 13

JAIN, L. C.; MEDSKER, L. R. **Recurrent Neural Networks: Design and Applications**. 1st. ed. USA: CRC Press, Inc., 1999. ISBN 0849371813. 23

JANOCHA, K.; CZARNECKI, W. M. **On Loss Functions for Deep Neural Networks in Classification**. 2017. 8, 19

KERAS. **Keras about**. 2020. <https://keras.io/about/>. 25

Lawrence, S. et al. Face recognition: a convolutional neural-network approach. **IEEE Transactions on Neural Networks**, v. 8, n. 1, p. 98–113, 1997. 22

LEPESKA. 2011. <https://www.bloomberg.com/news/articles/2011-12-12/are-crime-cameras-really-worth-the-money>. Accessed: 2020-11-25. 13

LIM, J. et al. Gun detection in surveillance videos using deep neural networks. In: **2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)**. [S.l.: s.n.], 2019. p. 1998–2002. 9, 28, 30

MARR, B. **What Are Artificial Neural Networks: a simple explanation for absolutely anyone**. 2020. <https://bernardmarr.com/default.asp?contentID=1568>. Accessed: 2020-11-26. 22

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **Bulletin of Mathematical Biophysics**, v. 5, p. 115–133, 1943. 16

OLGAC, A.; KARLIK, B. Performance analysis of various activation functions in generalized mlp architectures of neural networks. **International Journal of Artificial Intelligence And Expert Systems**, v. 1, p. 111–122, 02 2011. 18

OLMOS, R.; TABIK, S.; HERRERA, F. Automatic handgun detection alarm in videos using deep learning. **Neurocomputing**, v. 275, 02 2017. 13

PEI. 2019. <https://correio.rac.com.br/_conteudo/2019/08/campinas_e_rmc/858995-condominios-aderem-a-tecnologia-em-portarias.html>. Accessed: 2020-11-25. 13

PYTORCH. **PyTorch features**. 2020. <https://pytorch.org/features/>. 26

REDMON, J. et al. **You Only Look Once: Unified, Real-Time Object Detection**. 2015. Cite arxiv:1506.02640. Disponível em: <http://arxiv.org/abs/1506.02640>. 8, 34

REDMON, J.; FARHADI, A. **YOLOv3: An Incremental Improvement**. 2018. Cite arxiv:1804.02767Comment: Tech Report. Disponível em: <http://arxiv.org/abs/1804.02767>. 33

SCIENCE, T. D. **Towards Data Science**. 2020. <https://towardsdatascience.com/dive-really-deep-into-yolo-v3-a-beginners-guide-9e3d2666280e>. 8, 33

SPARK. **Spark Guide**. 2020. <https://spark.apache.org/docs/latest/ml-guide.html>. 25

TENSORFLOW. **TensorFlow Federated**. 2020. <https://www.tensorflow.org/federated>. 25

TIWARI, R. K.; VERMA, G. K. A computer vision based framework for visual gun detection using harris interest point detector. **Procedia Computer Science**, v. 54, p. 703 – 712, 2015. ISSN 1877-0509. Eleventh International Conference on Communication Networks, ICCN 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Data Mining and Warehousing, ICDMW 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Image and Signal Processing, ICISP 2015, August 21-23, 2015, Bangalore, India. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050915014076>. 9, 27, 28

UNODC. 2013. <https://www.unodc.org/documents/data-and-analysis/statistics/GSH2013/2014_GLOBAL_HOMICIDE_BOOK_web.pdf>. Accessed: 2020-11-25. 13

VERMA, G. K.; DHILLON, A. A handheld gun detection using faster r-cnn deep learning. In: **Proceedings of the 7th International Conference on Computer and Communication Technology**. New York, NY, USA: Association for Computing Machinery, 2017. (ICCCT-2017), p. 84–88. ISBN 9781450353243. Disponível em: <https://doi.org/10.1145/3154979.3154988>. 8, 30, 31, 32

A SBC Paper

Detecção de armas de fogo em vídeo usando redes neurais

Gabriel Vieira Baldessar[1]**,Rafael de Santiago**[2]

[1]Departamento de Informatica e Estatıstica
Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 5040 – 88040-900
Florianopolis – SC – Brazil

Resumo. *O número de furtos e roubos vem crescendo nos últimos anos. Nas situações mais graves, o uso de armas de fogo pode agravar a situação. Para reduzir esse problema, diversos espaços como lojas, depósitos, aeroportos e outros mais contratam diversos serviços de segurança. Entre eles o de vigilância remota. Visando apoiar a vigilância remota, este trabalho trata do desenvolvimento de uma rede neural artificial com o objetivo de auxiliar operadores humanos de sistemas de vigilância em circuitos internos. Mais especificamente, a rede neural deve reconhecer armas de fogo em video sem que seja necessário o envolvimento de um operador humano. Para endereçar esse objetivo, fora realizada uma pesquisa bibliográfica em literatura sobre redes neurais e reconhecimento de imagens, incluindo uma busca por trabalhos similares na literatura científica, bem como uma especificação preliminar da arquitetura da rede neural. Para atingir o objetivo, foi realizado o levantamento da literatura relacionada, redes que efetuam a classificação foram encontradas, um modelo de rede neural foi obtido e experimentações foram realizadas variando configurações de entrada no treinamento. Diversos experimentos foram realizados sobre a rede YOLOv3. Os melhores resultados foram obtidos obtiveram acurácia de 82,3%, uma precisão de 96,5% e uma taxa de recall de 84,4%.*

Abstract. *The number of thefts and robberies has beem growing in the past years. In the worst cases the use of guns can make the situation worse. Looking for reducing this problem, deposits, stores, gas stations, airports and several more locations hires several security services. Between those services are the remote surveillance. Looking to help the remote surveillance, this work deals with the development of a artificial neural netwotk in order to assist human operators of surveillance systems in camera internal circuits. Specifically, the neural network must recognize firearms on video without the need of interaction from an human operator. To address this goal, a bibliographic search had been carried out in literature on neyral networks and image recognition, including a search for similar works in the scientific literature, as well as a preliminary specification of the neural network architecture. Several experiments were carried out on the network. The best results achieved an accuracy of 82.3%, a precision of 96.5% and a recall rate of 84.4%. In order to achieve the objective, a survey of related literature was carried out, network that peform classification were found, a model of neural network was obteined and experiments were made varying the training configuration.*

1. Introdução

De acordo com O Escritório das Nações Unidas sobre Drogas e Crime (UNODC), o número de crimes envolvendo armas de fogo é alto em diversos países, como no México, com a marca de 21,5 crimes para cada 100.000 habitantes [UNO 2013].

Uma forma de reduzir o uso indevido de armas de fogo e através da prevenção dos crimes, por meio de detecção prévia, para que então agentes de segurança e forças da lei sejam capazes de agir [Olmos et al. 2017]. Nesse sentido, uma das iniciativas é o uso de câmeras de monitoramento. Geralmente, a ideia é a de que uma equipe fique responsável por monitorar câmeras para tentar identificar situações suspeitas e agir previamente. Essa iniciativa é comum na preservação em condomínios, empresas e prédios públicos [por 2019].

Câmeras de monitoramento podem envolver um custo maior do que a aquisição e a manutenção de equipamentos. É necessário que as imagens sejam analisadas constantemente para tomar-se ações pró-ativas. Por isso, geralmente esses sistemas envolvem o custo de contratar profissionais humanos que devem acompanhar as imagens ao vivo. Isso faz com que o custo seja alto para que seja empregado em um amplo aspecto [Wor 2011].

Soluções autônomas envolvem reconhecimento de padrões em imagens. Alguns exemplos de trabalhos que abordam o tema na literatura são:

- [Hijazi et al. 2015]: Trabalho que cobre o básico em relação às RNCs e aponta as sua vantagens em relação a outras técnicas;
- [Deepthi et al. 2019]: Trabalho que mostra como tecnologias relacionadas à vigilância podem se beneficiar da utilização conjunta com redes neurais;
- [Olmos et al. 2017]: Este Trabalho mostra como utilizar redes neurais junto a câmeras de segurança com o propósito de controle e vigilância.

Desta maneira, uma solução possível e inovadora seria utilizar de forma conjunta, um sistema vigilância com uma rede neural capaz de reconhecer armas de fogo e alertar as forças da lei em tempo real, a fim de diminuir o tempo de resposta das autoridades competentes [Deepthi et al. 2019]. Nesse contexto, a trabalho apresenta um estudo sobre método automático de pessoas portando armas de mão em vídeos. Para isso, as seguintes atividades foram realizadas: (i) procurar por trabalhos similares na literatura; (ii) especificar método computacional de aprendizagem supervisionada para realizar a tarefa de reconhecimento de padrões; (iii) coletar bases de dados com exemplos rotulados para treinar a técnica a ser utilizada; (iv) desenvolver método especificado; (v) experimentar e analisar parâmetros do método computacional desenvolvido; (vi) comparar resultados obtidos com outros encontrados na literatura; (vii) dar publicidade aos resultados através de veículos de divulgação científica.

2. Redes Neurais

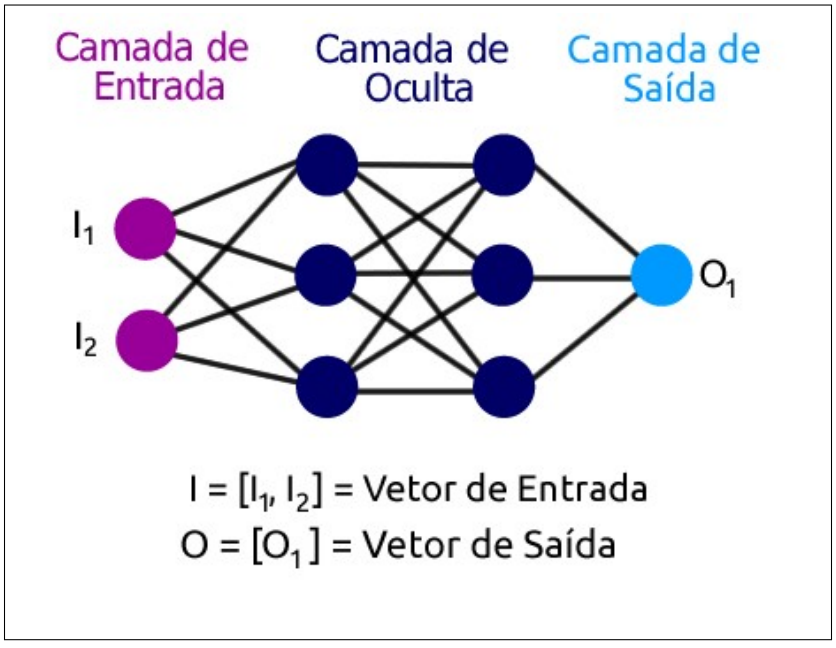
Concebida em 1943 em um artigo escrito por Warren McCulloch e Walter Pitts, onde era descrito como um nerônio deveria funcionar, foi modelada a primeira rede neural. Sua composição era de forma simples com circuitos eletrônicos [McCulloch and Pitts 1943]. A partir dai abriu-se caminho para dois ramos de pesquisa em redes neurais:

- Processos biológicos no cérebro;
- Aplicação de redes neurais em Inteligencia Artificial (IA)

Fukushima1975 concebe a primeira ideia de redes neurais multicamadas tendo como principio a hipótese: *"A sinapse do neurônio X para o neurônio Y é reforçada quando X dispara e nenhum outro neurônio próximo a Y é ativado mais forte que Y"*.

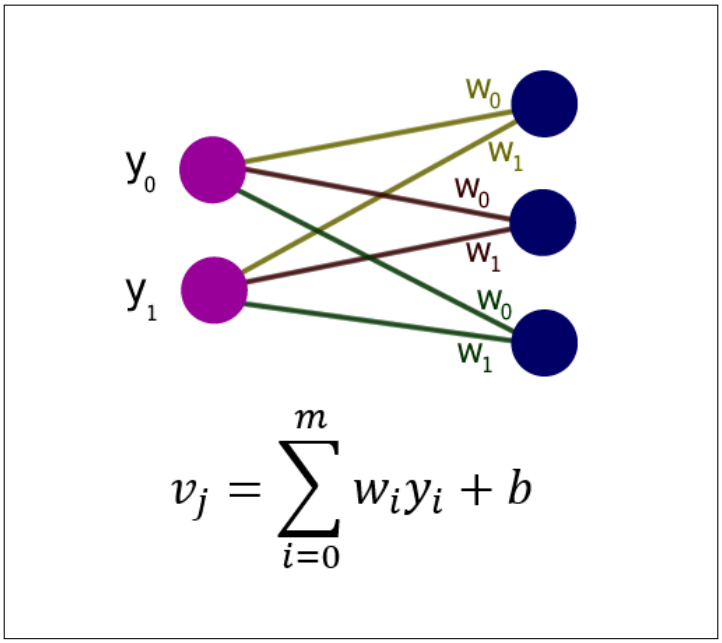
A ideia original de uma abordagem utilizando rede neural era criar um sistema computacional que pudesse resolver problemas gerais da mesma forma que um cérebro humano, porém com o passar do tempo, após observar diversas dificuldades, como representação do problema e a própria capacidade do hardware da época, a técnica acabou sendo utilizada para resolver problemas específicos, sendo a rede neural treinada unicamente para este propósito [Basheer and Hajmeer 2000].

A primeira a camada de nerônios é a entrada da rede, onde os dados são inseridos na rede e geralmente não ocorre nem um tipo de processamento em cima dos mesmos. Supondo uma rede com N camadas, as camadas 2 ate $N-1$ são chamadas de camadas intermediárias, ou camadas ocultas, nelas é onde o processamento dos dados acontece, afim de se chegar ao resultado desejado. E por fim, a ultima camada, conhecida como camada de saída, é a camada que dará o resultado da analise da rede sobre os dado. Na Figura 1, uma topologia de uma rede neural pode ser visualizada.



indent:

Cada neurônio recebe todos os valores da camada anterior. Cada valor recebido é multiplicado pelos pesos das ligações entre os neurônios, representados na Figura 2 pelos vetores "w", e somado com uma constante chamada *bias*, representado na Figura 2 por "b". Essa constante possui o intuito de centralizar a curva da função de ativação em um valor conveniente.



indent:

2.1. Deep Learning

Redes neurais são compostas geralmente por três camadas: uma camada de entrada, uma camada de saída e uma camada intermediária, responsável pelo processamento dos dados. O cérebro humano, por outro lado, funciona como se possuísse diversas camadas, como no exemplo dado por Bernard: *"Partes diferentes do cérebro humano são responsáveis por processar diferentes partes dos dados, e essas partes são ordenadas hierarquicamente, ou em camadas. Dessa forma, enquanto a informação vai entrando no cérebro, cada camada de neurônio processa os dados e gera informação, e subsequentemente passa essa informação para a próxima camada. Por exemplo, quando você sente o cheiro de pizza de uma pizzaria no outro lado da rua, seu cérebro processa o cheiro em múltiplos estágios: 'sinto cheiro de pizza'(Dados de entrada)... 'Eu adoro pizza'(Pensamento)...'Eu vou comprar um pedaço de pizza'(Tomando uma decisão)...'Eu prometi que não comeria mais besteiras'(Memória)...'Um pedaço não mata ninguém né?'(raciocinando)...'Vou comprar um pedaço de pizza!'(ação)"*

Logo uma rede neural com camadas profundas(deep learning) possui múltiplas camadas intermediárias. O termo "deep" geralmente se refere ao número de camadas da rede, uma vez que tradicionalmente redes neurais comuns tem de uma a três camadas intermediárias, com uma rede no estilo deep learning podendo chegar a mais de 150 camadas [Goodfellow et al. 2016].

2.2. Métricas de Análise de resultados

Após o treinamento de redes neurais, é preciso de métricas para que a rede possa ser analisada.

As redes são analisados da seguinte forma:

- Acurácia: Quantidade total de segmentos avaliados corretamente, tanto Verdadeiros positivos quanto Verdadeiros negativos, sobre o numero total de testes, de modo que

$$Acurácia = \frac{VP + VN}{VP + VN + FP + FN}$$

  Acurácia deve ser avaliada com um certo cuidado, pois não captura visão se a rede tem o comportamento esperado.
- Precisão: Quantidade de testes classificados como positivos que realmente são positivos, de modo que

$$Precisão = \frac{VP}{VP + FP}$$

- Recall: quantidade de segmentos positivos que foram classificados corretamente. É importante para garantir que a classe positiva que é minoritária também terá uma boa taxa de acerto. Para esse trabalho essa é uma métrica muito importante.

$$Recall = \frac{VP}{VP + FN}$$

3. Especificação e Métodos

3.1. Datasets

O dataset da Universidade de Granada, citado anteriormente em um trabalho relacionado, foi escolhido para compor o treinamento da rede, pois contem imagens de armas de fogos em diferentes contextos, como sobre uma mesa, no coldre ou sendo segurada por uma pessoa.

3.2. Especificação da rede que será utilizada

Inspirando-se nos trabalhos relacionados, a rede inicialmente escolhida para a realização do projeto foi a Rede Neural Convolucional YOLOv3, um detector de objetos em um estágio feito já pensando em reconhecimento de objetos em tempo real, uma melhoria do já citado YOLO, quem vem demonstrando desempenho superior a outras redes já citadas anteriormente.

3.3. Roteiro de experimentos

Os experimentos se resumem em testar a rede com uma base de testes, em que previamente se tem conhecimento se a imagem possui ou não uma arma de fogo. Após a realização dos testes, se obtém os números de Verdadeiros Positivos, Falsos Positivos, Verdadeiros Negativos e Falsos Negativos, para avaliação descrita na próxima seção do trabalho.

3.4. Dados de treinamento

O treinamento consistiu em diferentes combinações das imagens de indivíduos portando armas de fogo em diferentes ângulos e também armas soltas. As imagens foram retiradas do dataset de armas de fogo da Universidade de Granada e foram classificadas utilizando o software de código aberto chamado CVAT.

3.5. Treinamento

Um total de oito experimentos foram realizados, alterando as imagens, alterando a combinação de imagens e o número de épocas (numero de vezes que o treinamento utilizada o conjunto de imagens) que foram utilizadas.

Todos os pesos gerados pelo treinamento com as imagens de arma de fogo utilizaram como base os pesos fornecidos pelo repositório original do projeto YOLOv3 que é obtido automaticamente ao executar o projeto pela primeira vez.

Os treinamentos foram realizados em uma máquina com as seguintes especificações.

- Processador: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
- Memória RAM: 32 GB
- GPU: NVIDIA GeForce GTX 1660 Ti 6 GB

3.6. Resultados do treinamento

Oito conjuntos de treinamentos foram experimentados para a rede YOLO. Dos conjuntos analisados, apenas três ensaios são destacados, por terem obtido melhores resultados. Eles estão divididos em Treinamento 1, 2 e 3.

Os parâmetros em si também são experimentações, uma vez que o treinamento possui um ponto ótimo e não necessariamente mais imagens de treinamento geram uma rede melhor.

3.7. Comparação dos treinamentos

Os dados Verdadeiro Positivo(VP), Falso Positivo(FP), Verdadeiro Negativo(VN), Falso Negativo(FN) obtidos dos pesos podem ser visualizados na Tabela 1.

	VP	**FP**	**VN**	**FN**
Treinamento 1	199	8	8	93
Treinamento 2	245	9	6	45
Treinamento 3	215	18	8	74

indent:

3.7.1. Acurácia

A acurácia demonstra se as armas de fogo foram classificadas corretamente e também se outros objetos não foram classificados erroneamente como armas de fogo.

Abaixo a acurácia dos treinamentos citados na seção anterior

- Treinamento 1: Acurácia = $\frac{VP+VN}{VP+VN+FP+FN} = \frac{199+8}{199+8+8+93}$ = 0.672 = 67,2%
- Treinamento 2: Acurácia = $\frac{VP+VN}{VP+VN+FP+FN} = \frac{245+6}{245+6+9+45}$ = 0.823 = 82,3%
- Treinamento 3: Acurácia = $\frac{VP+VN}{VP+VN+FP+FN} = \frac{215+8}{215+8+18+74}$ = 0.708 = 70,8%

Essa primeira métrica ajuda a avaliar o desempenho dos pesos calibrados durante os experimentos. Os pesos obtidos com o treinamento 2 figuraram na melhor opção. Porém a discussão dos resultados deve considerar o uso de outras métricas como Precisão e Recall (próximas seções).

3.7.2. Precisão

Precisão é uma métrica que avalia a porcentagem de predições Verdadeiros Positivos em relação a todas as predições Positivas. Abaixo a precisão dos treinamentos citados na seção anterior.

- Treinamento 1: Precisão = $\frac{VP}{VP+FP} = \frac{199}{199+8}$ = 0.9613 = 96,1%
- Treinamento 2: Precisão = $\frac{VP}{VP+FP} = \frac{245}{245+9}$ = 0.965 = 96,5%
- Treinamento 3: Precisão = $\frac{VP}{VP+FP} = \frac{215}{215+18}$ = 0.923 = 92,3%

Os resultados mostram que o treinamento 2 é também mais preciso do que os outros dois treinamentos, embora o treinamento 1 tenha obtido resultado semelhante.

3.7.3.

Recall Recall tem como objetivo determinar quanto das imagens que se esperavam ser Positivas foram realmente classificadas como Positivas.

Abaixo o Recall dos treinamentos citados na seção anterior.

- Treinamento 1: Recall = $\frac{VP}{VP+FN} = \frac{199}{199+93}$ = 0.744 = 74,4%
- Treinamento 2: Recall = $\frac{VP}{VP+FN} = \frac{245}{245+45}$ = 0,884 = 84,4%
- Treinamento 3: Recall = $\frac{VP}{VP+FN} = \frac{215}{215+74}$ = 0.682 = 68,2%

Novamente o treinamento 2 se destaca no valor da métrica, alcançando uma taxa de recall superior a 10% do segundo lugar.

3.8. Resultados

Observando as métricas de acurácia, precisão e recall, pode-se chegar à conclusão que dos três pesos apresentados, o que obteve melhor desempenho em todas as métricas foi o treinamento 2, não deixando duvida do desempenho superior em relação as outras.

4. Conclusão e trabalhos futuros

Para reduzir crimes com armas de fogo, uma abordagem que pode ser adotada é a prevenção por meio da detecção de indivíduos portando-as. Utilizando-se de técnicas de detecções de padrões, como redes neurais artificiais, os problemas de custo e eficiência nessa aplicação podem ser reduzidos. Por exemplo, a tarefa de detecção fica para a inteligencia artificial, que avisaria um operador humano caso algo seja detectado, deixando a cargo deste tomar alguma atitude.

Este trabalho especifica um procedimento automático para detecção de indivíduos portando armas de fogo utilizando rede neural artificial profunda. Para cumprir essa tarefa, um levantamento de trabalhos relacionados foi realizado na literatura acadêmica, um conjunto de *datasets* foi identificado, e a especificação da rede neural foi realizada.

Com os experimentos realizados foi possível criar uma rede capaz de realizar detecção de armas de fogo com uma acurácia de 82,3% , uma precisão de 96,5% e uma taxa de recall de 84,5% dentro das condições apresentadas durante o treinamento da rede.

embora ainda falte refinamento tando sobre o *dataset* e sobre o treinamento da rede, a rede obtida mostra-se uma prova de conceito bem sucedida, mostrando que a detecção de objetos perigosos por sistemas de vigilância é possível

Como trabalhos futuros podemos listar:

- a otimização do *dataset* abrangendo mais casos próximos do real afim de evitar qualquer problemas causados pelo *dataset* atual.
- Expandir o numero de armas detectadas pela rede, como armas brancas e armas longas.
- Expandir o *dataset* com imagens de próprias de circuito interno de vídeos e também imagens noturnas.
- Comparar o desempenho da rede YOLOv3 com outras redes do estado da arte.

5. Exemplo de um Apêndice

Apêndices são iniciados com o comando appendix. Também é possível introduzi-los usando o environment appendix.

6. Exemplo de Outro Apêndice

Texto do Apêndice 6.

Referências

(2011). Lepeska. `https://www.bloomberg.com/news/articles/2011-12-12/are-crime-cameras-really-worth-the-money`. Accessed: 2020-11-25.

(2013). Unodc. `https://www.unodc.org/documents/data-and-analysis/statistics/GSH2013/2014_GLOBAL_HOMICIDE_BOOK_web.pdf`. Accessed: 2020-11-25.

(2019). Pei. `https://correio.rac.com.br/_conteudo/2019/08/campinas_e_rmc/858995-condominios-aderem-a-tecnologia-em-portarias.html`. Accessed: 2020-11-25.

Basheer, I. and Hajmeer, M. (2000). Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods*, 43(1):3 – 31. Neural Computting in Micrbiology.

Deepthi, T., Gaayathri, R., .S, S., Gebin, A., and Nithya, R. (2019). Firearm recognition using convolutional neural network. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 136–141.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Hijazi, S., Kumar, R., and Rowen, C. (2015). Using convolutional neural networks for image recognition by.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

Olmos, R., Tabik, S., and Herrera, F. (2017). Automatic handgun detection alarm in videos using deep learning. *Neurocomputing*, 275.

```
# pip install -r requirements.txt

# base ----------------------------------------
Cython
matplotlib>=3.2.2
numpy>=1.18.5
opencv-python>=4.1.2
Pillow
PyYAML>=5.3
scipy>=1.4.1
tensorboard>=2.2
torch>=1.7.0
torchvision>=0.8.1
tqdm>=4.41.0

# logging -------------------------------------
# wandb

# plotting ------------------------------------
seaborn>=0.11.0
pandas

# export --------------------------------------
# coremltools==4.0
# onnx>=1.8.0
# scikit-learn==0.19.2  # for coreml quantization

# extras --------------------------------------
thop  # FLOPS computation
pycocotools>=2.0  # COCO mAP


import argparse
import time
from pathlib import Path

import cv2
import torch
import torch.backends.cudnn as cudnn
from numpy import random

from models.experimental import attempt_load
from utils.datasets import LoadStreams, LoadImages
from utils.general import check_img_size, check_requirements,
    non_max_suppression, apply_classifier, scale_coords, \
     xyxy2xywh, strip_optimizer, set_logging, increment_path
from utils.plots import plot_one_box
```

```
47 from utils.torch_utils import select_device, load_classifier,
       time_synchronized
48
49
50 def detect(save_img=False):
51     source, weights, view_img, save_txt, imgsz = opt.source, opt.weights,
       opt.view_img, opt.save_txt, opt.img_size
52     webcam = source.isnumeric() or source.endswith('.txt') or source.lower
       ().startswith(
53         ('rtsp://', 'rtmp://', 'http://'))
54
55     # Directories
56     save_dir = Path(increment_path(Path(opt.project) / opt.name, exist_ok=
       opt.exist_ok)) # increment run
57     (save_dir / 'labels' if save_txt else save_dir).mkdir(parents=True,
       exist_ok=True) # make dir
58
59     # Initialize
60     set_logging()
61     device = select_device(opt.device)
62     half = device.type != 'cpu' # half precision only supported on CUDA
63
64     # Load model
65     model = attempt_load(weights, map_location=device) # load FP32 model
66     imgsz = check_img_size(imgsz, s=model.stride.max()) # check img_size
67     if half:
68         model.half() # to FP16
69
70     # Second-stage classifier
71     classify = False
72     if classify:
73         modelc = load_classifier(name='resnet101', n=2) # initialize
74         modelc.load_state_dict(torch.load('weights/resnet101.pt',
       map_location=device)['model']).to(device).eval()
75
76     # Set Dataloader
77     vid_path, vid_writer = None, None
78     if webcam:
79         view_img = True
80         cudnn.benchmark = True # set True to speed up constant image size
       inference
81         dataset = LoadStreams(source, img_size=imgsz)
82     else:
83         save_img = True
84         dataset = LoadImages(source, img_size=imgsz)
85
86     # Get names and colors
```

```
87      names = model.module.names if hasattr(model, 'module') else model.names
88      colors = [[random.randint(0, 255) for _ in range(3)] for _ in names]
89
90      # Run inference
91      t0 = time.time()
92      img = torch.zeros((1, 3, imgsz, imgsz), device=device)  # init img
93      _ = model(img.half() if half else img) if device.type != 'cpu' else
    None  # run once
94      for path, img, im0s, vid_cap in dataset:
95          img = torch.from_numpy(img).to(device)
96          img = img.half() if half else img.float()  # uint8 to fp16/32
97          img /= 255.0  # 0 - 255 to 0.0 - 1.0
98          if img.ndimension() == 3:
99              img = img.unsqueeze(0)
100
101         # Inference
102         t1 = time_synchronized()
103         pred = model(img, augment=opt.augment)[0]
104
105         # Apply NMS
106         pred = non_max_suppression(pred, opt.conf_thres, opt.iou_thres,
    classes=opt.classes, agnostic=opt.agnostic_nms)
107         t2 = time_synchronized()
108
109         # Apply Classifier
110         if classify:
111             pred = apply_classifier(pred, modelc, img, im0s)
112
113         # Process detections
114         for i, det in enumerate(pred):  # detections per image
115             if webcam:  # batch_size >= 1
116                 p, s, im0, frame = path[i], '%g: ' % i, im0s[i].copy(),
    dataset.count
117             else:
118                 p, s, im0, frame = path, '', im0s, getattr(dataset, 'frame
    ', 0)
119
120             p = Path(p)  # to Path
121             save_path = str(save_dir / p.name)  # img.jpg
122             txt_path = str(save_dir / 'labels' / p.stem) + ('' if dataset.
    mode == 'image' else f'_{frame}')  # img.txt
123             s += '%gx%g ' % img.shape[2:]  # print string
124             gn = torch.tensor(im0.shape)[[1, 0, 1, 0]]  # normalization
    gain whwh
125             if len(det):
126                 # Rescale boxes from img_size to im0 size
127                 det[:, :4] = scale_coords(img.shape[2:], det[:, :4], im0.
```

```
shape).round()
128
129             # Print results
130             for c in det[:, -1].unique():
131                 n = (det[:, -1] == c).sum()  # detections per class
132                 s += f'{n} {names[int(c)]}s, '  # add to string
133
134             # Write results
135             for *xyxy, conf, cls in reversed(det):
136                 if save_txt:  # Write to file
137                     xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) /
gn).view(-1).tolist()  # normalized xywh
138                     line = (cls, *xywh, conf) if opt.save_conf else (
cls, *xywh)  # label format
139                     with open(txt_path + '.txt', 'a') as f:
140                         f.write(('%g ' * len(line)).rstrip() % line +
'\n')
141
142                 if save_img or view_img:  # Add bbox to image
143                     label = f'{names[int(cls)]} {conf:.2f}'
144                     plot_one_box(xyxy, im0, label=label, color=colors[
int(cls)], line_thickness=3)
145
146         # Print time (inference + NMS)
147         print(f'{s}Done. ({t2 - t1:.3f}s)')
148
149         # Stream results
150         if view_img:
151             cv2.imshow(str(p), im0)
152
153         # Save results (image with detections)
154         if save_img:
155             if dataset.mode == 'image':
156                 cv2.imwrite(save_path, im0)
157             else:  # 'video'
158                 if vid_path != save_path:  # new video
159                     vid_path = save_path
160                     if isinstance(vid_writer, cv2.VideoWriter):
161                         vid_writer.release()  # release previous video
writer
162
163                     fourcc = 'mp4v'  # output video codec
164                     fps = vid_cap.get(cv2.CAP_PROP_FPS)
165                     w = int(vid_cap.get(cv2.CAP_PROP_FRAME_WIDTH))
166                     h = int(vid_cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
167                     vid_writer = cv2.VideoWriter(save_path, cv2.
VideoWriter_fourcc(*fourcc), fps, (w, h))
```

```
168                     vid_writer.write(im0)
169
170     if save_txt or save_img:
171         s = f"\n{len(list(save_dir.glob('labels/*.txt')))} labels saved to
    {save_dir / 'labels'}" if save_txt else ''
172         print(f"Results saved to {save_dir}{s}")
173
174     print(f'Done. ({time.time() - t0:.3f}s)')
175
176
177 if __name__ == '__main__':
178     parser = argparse.ArgumentParser()
179     parser.add_argument('--weights', nargs='+', type=str, default='yolov3.
    pt', help='model.pt path(s)')
180     parser.add_argument('--source', type=str, default='data/images', help='
    source')  # file/folder, 0 for webcam
181     parser.add_argument('--img-size', type=int, default=640, help='
    inference size (pixels)')
182     parser.add_argument('--conf-thres', type=float, default=0.25, help='
    object confidence threshold')
183     parser.add_argument('--iou-thres', type=float, default=0.45, help='IOU
    threshold for NMS')
184     parser.add_argument('--device', default='', help='cuda device, i.e. 0
    or 0,1,2,3 or cpu')
185     parser.add_argument('--view-img', action='store_true', help='display
    results')
186     parser.add_argument('--save-txt', action='store_true', help='save
    results to *.txt')
187     parser.add_argument('--save-conf', action='store_true', help='save
    confidences in --save-txt labels')
188     parser.add_argument('--classes', nargs='+', type=int, help='filter by
    class: --class 0, or --class 0 2 3')
189     parser.add_argument('--agnostic-nms', action='store_true', help='class-
    agnostic NMS')
190     parser.add_argument('--augment', action='store_true', help='augmented
    inference')
191     parser.add_argument('--update', action='store_true', help='update all
    models')
192     parser.add_argument('--project', default='runs/detect', help='save
    results to project/name')
193     parser.add_argument('--name', default='exp', help='save results to
    project/name')
194     parser.add_argument('--exist-ok', action='store_true', help='existing
    project/name ok, do not increment')
195     opt = parser.parse_args()
196     print(opt)
197     check_requirements()
```

```
198
199     with torch.no_grad():
200         if opt.update:  # update all models (to fix SourceChangeWarning)
201             for opt.weights in ['yolov3.pt', 'yolov3-spp.pt', 'yolov3-tiny.
    pt']:
202                 detect()
203                 strip_optimizer(opt.weights)
204         else:
205             detect()
206
207
208 import argparse
209 import json
210 import os
211 from pathlib import Path
212 from threading import Thread
213
214 import numpy as np
215 import torch
216 import yaml
217 from tqdm import tqdm
218
219 from models.experimental import attempt_load
220 from utils.datasets import create_dataloader
221 from utils.general import coco80_to_coco91_class, check_dataset, check_file
    , check_img_size, check_requirements, \
222     box_iou, non_max_suppression, scale_coords, xyxy2xywh, xywh2xyxy,
    set_logging, increment_path, colorstr
223 from utils.loss import compute_loss
224 from utils.metrics import ap_per_class, ConfusionMatrix
225 from utils.plots import plot_images, output_to_target, plot_study_txt
226 from utils.torch_utils import select_device, time_synchronized
227
228
229 def test(data,
230          weights=None,
231          batch_size=32,
232          imgsz=640,
233          conf_thres=0.001,
234          iou_thres=0.6,  # for NMS
235          save_json=False,
236          single_cls=False,
237          augment=False,
238          verbose=False,
239          model=None,
240          dataloader=None,
241          save_dir=Path(''),  # for saving images
```

```
242         save_txt=False,  # for auto-labelling
243         save_hybrid=False,  # for hybrid auto-labelling
244         save_conf=False,  # save auto-label confidences
245         plots=True,
246         log_imgs=0):  # number of logged images
247 
248     # Initialize/load model and set device
249     training = model is not None
250     if training:  # called by train.py
251         device = next(model.parameters()).device  # get model device
252 
253     else:  # called directly
254         set_logging()
255         device = select_device(opt.device, batch_size=batch_size)
256 
257         # Directories
258         save_dir = Path(increment_path(Path(opt.project) / opt.name,
    exist_ok=opt.exist_ok))  # increment run
259         (save_dir / 'labels' if save_txt else save_dir).mkdir(parents=True,
     exist_ok=True)  # make dir
260 
261         # Load model
262         model = attempt_load(weights, map_location=device)  # load FP32
    model
263         imgsz = check_img_size(imgsz, s=model.stride.max())  # check
    img_size
264 
265         # Multi-GPU disabled, incompatible with .half() https://github.com/
    ultralytics/yolov5/issues/99
266         # if device.type != 'cpu' and torch.cuda.device_count() > 1:
267         #     model = nn.DataParallel(model)
268 
269     # Half
270     half = device.type != 'cpu'  # half precision only supported on CUDA
271     if half:
272         model.half()
273 
274     # Configure
275     model.eval()
276     is_coco = data.endswith('coco.yaml')  # is COCO dataset
277     with open(data) as f:
278         data = yaml.load(f, Loader=yaml.FullLoader)  # model dict
279     check_dataset(data)  # check
280     nc = 1 if single_cls else int(data['nc'])  # number of classes
281     iouv = torch.linspace(0.5, 0.95, 10).to(device)  # iou vector for mAP@0
    .5:0.95
282     niou = iouv.numel()
```

```
283
284     # Logging
285     log_imgs, wandb = min(log_imgs, 100), None  # ceil
286     try:
287         import wandb  # Weights & Biases
288     except ImportError:
289         log_imgs = 0
290
291     # Dataloader
292     if not training:
293         img = torch.zeros((1, 3, imgsz, imgsz), device=device)  # init img
294         _ = model(img.half() if half else img) if device.type != 'cpu' else
     None  # run once
295         path = data['test'] if opt.task == 'test' else data['val']  # path
    to val/test images
296         dataloader = create_dataloader(path, imgsz, batch_size, model.
    stride.max(), opt, pad=0.5, rect=True,
297                                        prefix=colorstr('test: ' if opt.task
     == 'test' else 'val: '))[0]
298
299     seen = 0
300     confusion_matrix = ConfusionMatrix(nc=nc)
301     names = {k: v for k, v in enumerate(model.names if hasattr(model, '
    names') else model.module.names)}
302     coco91class = coco80_to_coco91_class()
303     s = ('%20s' + '%12s' * 6) % ('Class', 'Images', 'Targets', 'P', 'R', '
    mAP@.5', 'mAP@.5:.95')
304     p, r, f1, mp, mr, map50, map, t0, t1 = 0., 0., 0., 0., 0., 0., 0., 0.,
    0.
305     loss = torch.zeros(3, device=device)
306     jdict, stats, ap, ap_class, wandb_images = [], [], [], [], []
307     for batch_i, (img, targets, paths, shapes) in enumerate(tqdm(dataloader
    , desc=s)):
308         img = img.to(device, non_blocking=True)
309         img = img.half() if half else img.float()  # uint8 to fp16/32
310         img /= 255.0  # 0 - 255 to 0.0 - 1.0
311         targets = targets.to(device)
312         nb, _, height, width = img.shape  # batch size, channels, height,
    width
313
314         with torch.no_grad():
315             # Run model
316             t = time_synchronized()
317             inf_out, train_out = model(img, augment=augment)  # inference
    and training outputs
318             t0 += time_synchronized() - t
319
```

```
320             # Compute loss
321             if training:
322                 loss += compute_loss([x.float() for x in train_out],
    targets, model)[1][:3]  # box, obj, cls
323
324             # Run NMS
325             targets[:, 2:] *= torch.Tensor([width, height, width, height]).
    to(device)  # to pixels
326             lb = [targets[targets[:, 0] == i, 1:] for i in range(nb)] if
    save_hybrid else []  # for autolabelling
327             t = time_synchronized()
328             output = non_max_suppression(inf_out, conf_thres=conf_thres,
    iou_thres=iou_thres, labels=lb)
329             t1 += time_synchronized() - t
330
331         # Statistics per image
332         for si, pred in enumerate(output):
333             labels = targets[targets[:, 0] == si, 1:]
334             nl = len(labels)
335             tcls = labels[:, 0].tolist() if nl else []  # target class
336             path = Path(paths[si])
337             seen += 1
338
339             if len(pred) == 0:
340                 if nl:
341                     stats.append((torch.zeros(0, niou, dtype=torch.bool),
    torch.Tensor(), torch.Tensor(), tcls))
342                 continue
343
344             # Predictions
345             predn = pred.clone()
346             scale_coords(img[si].shape[1:], predn[:, :4], shapes[si][0],
    shapes[si][1])  # native-space pred
347
348             # Append to text file
349             if save_txt:
350                 gn = torch.tensor(shapes[si][0])[[1, 0, 1, 0]]  #
    normalization gain whwh
351                 for *xyxy, conf, cls in predn.tolist():
352                     xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).
    view(-1).tolist()  # normalized xywh
353                     line = (cls, *xywh, conf) if save_conf else (cls, *xywh
    )  # label format
354                     with open(save_dir / 'labels' / (path.stem + '.txt'), '
    a') as f:
355                         f.write(('%g ' * len(line)).rstrip() % line + '\n')
356
```

```
357             # W&B logging
358             if plots and len(wandb_images) < log_imgs:
359                 box_data = [{"position": {"minX": xyxy[0], "minY": xyxy[1],
     "maxX": xyxy[2], "maxY": xyxy[3]},
360                              "class_id": int(cls),
361                              "box_caption": "%s %.3f" % (names[cls], conf),
362                              "scores": {"class_score": conf},
363                              "domain": "pixel"} for *xyxy, conf, cls in
    pred.tolist()]
364                 boxes = {"predictions": {"box_data": box_data, "
    class_labels": names}}  # inference-space
365                 wandb_images.append(wandb.Image(img[si], boxes=boxes,
    caption=path.name))
366 
367             # Append to pycocotools JSON dictionary
368             if save_json:
369                 # [{"image_id": 42, "category_id": 18, "bbox": [258.15,
    41.29, 348.26, 243.78], "score": 0.236}, ...
370                 image_id = int(path.stem) if path.stem.isnumeric() else
    path.stem
371                 box = xyxy2xywh(predn[:, :4])  # xywh
372                 box[:, :2] -= box[:, 2:] / 2  # xy center to top-left
    corner
373                 for p, b in zip(pred.tolist(), box.tolist()):
374                     jdict.append({'image_id': image_id,
375                                   'category_id': coco91class[int(p[5])] if
    is_coco else int(p[5]),
376                                   'bbox': [round(x, 3) for x in b],
377                                   'score': round(p[4], 5)})
378 
379             # Assign all predictions as incorrect
380             correct = torch.zeros(pred.shape[0], niou, dtype=torch.bool,
    device=device)
381             if nl:
382                 detected = []  # target indices
383                 tcls_tensor = labels[:, 0]
384 
385                 # target boxes
386                 tbox = xywh2xyxy(labels[:, 1:5])
387                 scale_coords(img[si].shape[1:], tbox, shapes[si][0], shapes
    [si][1])  # native-space labels
388                 if plots:
389                     confusion_matrix.process_batch(pred, torch.cat((labels
    [:, 0:1], tbox), 1))
390 
391                 # Per target class
392                 for cls in torch.unique(tcls_tensor):
```

```
393                     ti = (cls == tcls_tensor).nonzero(as_tuple=False).view
    (-1)  # prediction indices
394                     pi = (cls == pred[:, 5]).nonzero(as_tuple=False).view
    (-1)  # target indices
395 
396                     # Search for detections
397                     if pi.shape[0]:
398                         # Prediction to target ious
399                         ious, i = box_iou(predn[pi, :4], tbox[ti]).max(1)
    # best ious, indices
400 
401                         # Append detections
402                         detected_set = set()
403                         for j in (ious > iouv[0]).nonzero(as_tuple=False):
404                             d = ti[i[j]]  # detected target
405                             if d.item() not in detected_set:
406                                 detected_set.add(d.item())
407                                 detected.append(d)
408                                 correct[pi[j]] = ious[j] > iouv  #
    iou_thres is 1xn
409                                 if len(detected) == nl:  # all targets
    already located in image
410                                     break
411 
412             # Append statistics (correct, conf, pcls, tcls)
413             stats.append((correct.cpu(), pred[:, 4].cpu(), pred[:, 5].cpu()
    , tcls))
414 
415         # Plot images
416         if plots and batch_i < 3:
417             f = save_dir / f'test_batch{batch_i}_labels.jpg'  # labels
418             Thread(target=plot_images, args=(img, targets, paths, f, names)
    , daemon=True).start()
419             f = save_dir / f'test_batch{batch_i}_pred.jpg'  # predictions
420             Thread(target=plot_images, args=(img, output_to_target(output),
     paths, f, names), daemon=True).start()
421 
422     # Compute statistics
423     stats = [np.concatenate(x, 0) for x in zip(*stats)]  # to numpy
424     if len(stats) and stats[0].any():
425         p, r, ap, f1, ap_class = ap_per_class(*stats, plot=plots, save_dir=
    save_dir, names=names)
426         p, r, ap50, ap = p[:, 0], r[:, 0], ap[:, 0], ap.mean(1)  # [P, R,
    AP@0.5, AP@0.5:0.95]
427         mp, mr, map50, map = p.mean(), r.mean(), ap50.mean(), ap.mean()
428         nt = np.bincount(stats[3].astype(np.int64), minlength=nc)  # number
     of targets per class
```

```
429     else:
430         nt = torch.zeros(1)
431 
432     # Print results
433     pf = '%20s' + '%12.3g' * 6  # print format
434     print(pf % ('all', seen, nt.sum(), mp, mr, map50, map))
435 
436     # Print results per class
437     if (verbose or (nc <= 20 and not training)) and nc > 1 and len(stats):
438         for i, c in enumerate(ap_class):
439             print(pf % (names[c], seen, nt[c], p[i], r[i], ap50[i], ap[i]))
440 
441     # Print speeds
442     t = tuple(x / seen * 1E3 for x in (t0, t1, t0 + t1)) + (imgsz, imgsz,
    batch_size)  # tuple
443     if not training:
444         print('Speed: %.1f/%.1f/%.1f ms inference/NMS/total per %gx%g image
     at batch-size %g' % t)
445 
446     # Plots
447     if plots:
448         confusion_matrix.plot(save_dir=save_dir, names=list(names.values())
    )
449         if wandb and wandb.run:
450             wandb.log({"Images": wandb_images})
451             wandb.log({"Validation": [wandb.Image(str(f), caption=f.name)
    for f in sorted(save_dir.glob('test*.jpg'))]})
452 
453     # Save JSON
454     if save_json and len(jdict):
455         w = Path(weights[0] if isinstance(weights, list) else weights).stem
     if weights is not None else ''  # weights
456         anno_json = '../coco/annotations/instances_val2017.json'  #
    annotations json
457         pred_json = str(save_dir / f"{w}_predictions.json")  # predictions
    json
458         print('\nEvaluating pycocotools mAP... saving %s...' % pred_json)
459         with open(pred_json, 'w') as f:
460             json.dump(jdict, f)
461 
462         try:  # https://github.com/cocodataset/cocoapi/blob/master/
    PythonAPI/pycocoEvalDemo.ipynb
463             from pycocotools.coco import COCO
464             from pycocotools.cocoeval import COCOeval
465 
466             anno = COCO(anno_json)  # init annotations api
467             pred = anno.loadRes(pred_json)  # init predictions api
```

```
468             eval = COCOeval(anno, pred, 'bbox')
469             if is_coco:
470                 eval.params.imgIds = [int(Path(x).stem) for x in dataloader
    .dataset.img_files]  # image IDs to evaluate
471             eval.evaluate()
472             eval.accumulate()
473             eval.summarize()
474             map, map50 = eval.stats[:2]  # update results (mAP@0.5:0.95,
    mAP@0.5)
475         except Exception as e:
476             print(f'pycocotools unable to run: {e}')
477 
478     # Return results
479     if not training:
480         s = f"\n{len(list(save_dir.glob('labels/*.txt')))} labels saved to
    {save_dir / 'labels'}" if save_txt else ''
481         print(f"Results saved to {save_dir}{s}")
482     model.float()  # for training
483     maps = np.zeros(nc) + map
484     for i, c in enumerate(ap_class):
485         maps[c] = ap[i]
486     return (mp, mr, map50, map, *(loss.cpu() / len(dataloader)).tolist()),
    maps, t
487 
488 
489 if __name__ == '__main__':
490     parser = argparse.ArgumentParser(prog='test.py')
491     parser.add_argument('--weights', nargs='+', type=str, default='yolov3.
    pt', help='model.pt path(s)')
492     parser.add_argument('--data', type=str, default='data/coco128.yaml',
    help='*.data path')
493     parser.add_argument('--batch-size', type=int, default=32, help='size of
     each image batch')
494     parser.add_argument('--img-size', type=int, default=640, help='
    inference size (pixels)')
495     parser.add_argument('--conf-thres', type=float, default=0.001, help='
    object confidence threshold')
496     parser.add_argument('--iou-thres', type=float, default=0.6, help='IOU
    threshold for NMS')
497     parser.add_argument('--task', default='val', help="'val', 'test', '
    study'")
498     parser.add_argument('--device', default='', help='cuda device, i.e. 0
    or 0,1,2,3 or cpu')
499     parser.add_argument('--single-cls', action='store_true', help='treat as
     single-class dataset')
500     parser.add_argument('--augment', action='store_true', help='augmented
    inference')
```

```
501     parser.add_argument('--verbose', action='store_true', help='report mAP
    by class')
502     parser.add_argument('--save-txt', action='store_true', help='save
    results to *.txt')
503     parser.add_argument('--save-hybrid', action='store_true', help='save
    label+prediction hybrid results to *.txt')
504     parser.add_argument('--save-conf', action='store_true', help='save
    confidences in --save-txt labels')
505     parser.add_argument('--save-json', action='store_true', help='save a
    cocoapi-compatible JSON results file')
506     parser.add_argument('--project', default='runs/test', help='save to
    project/name')
507     parser.add_argument('--name', default='exp', help='save to project/name
    ')
508     parser.add_argument('--exist-ok', action='store_true', help='existing
    project/name ok, do not increment')
509     opt = parser.parse_args()
510     opt.save_json |= opt.data.endswith('coco.yaml')
511     opt.data = check_file(opt.data)  # check file
512     print(opt)
513     check_requirements()
514 
515     if opt.task in ['val', 'test']:  # run normally
516         test(opt.data,
517              opt.weights,
518              opt.batch_size,
519              opt.img_size,
520              opt.conf_thres,
521              opt.iou_thres,
522              opt.save_json,
523              opt.single_cls,
524              opt.augment,
525              opt.verbose,
526              save_txt=opt.save_txt | opt.save_hybrid,
527              save_hybrid=opt.save_hybrid,
528              save_conf=opt.save_conf,
529              )
530 
531     elif opt.task == 'study':  # run over a range of settings and save/plot
532         for weights in ['yolov3.pt', 'yolov3-spp.pt', 'yolov3-tiny.pt']:
533             f = 'study_%s_%s.txt' % (Path(opt.data).stem, Path(weights).
    stem)  # filename to save to
534             x = list(range(320, 800, 64))  # x axis
535             y = []  # y axis
536             for i in x:  # img-size
537                 print('\nRunning %s point %s...' % (f, i))
538                 r, _, t = test(opt.data, weights, opt.batch_size, i, opt.
```

```
    conf_thres, opt.iou_thres, opt.save_json,
539                                 plots=False)
540                 y.append(r + t)  # results and times
541             np.savetxt(f, y, fmt='%10.4g')  # save
542         os.system('zip -r study.zip study_*.txt')
543         plot_study_txt(f, x)  # plot
544
545
546 import argparse
547 import logging
548 import math
549 import os
550 import random
551 import time
552 from pathlib import Path
553 from threading import Thread
554
555 import numpy as np
556 import torch.distributed as dist
557 import torch.nn as nn
558 import torch.nn.functional as F
559 import torch.optim as optim
560 import torch.optim.lr_scheduler as lr_scheduler
561 import torch.utils.data
562 import yaml
563 from torch.cuda import amp
564 from torch.nn.parallel import DistributedDataParallel as DDP
565 from torch.utils.tensorboard import SummaryWriter
566 from tqdm import tqdm
567
568 import test  # import test.py to get mAP after each epoch
569 from models.experimental import attempt_load
570 from models.yolo import Model
571 from utils.autoanchor import check_anchors
572 from utils.datasets import create_dataloader
573 from utils.general import labels_to_class_weights, increment_path,
    labels_to_image_weights, init_seeds, \
574     fitness, strip_optimizer, get_latest_run, check_dataset, check_file,
    check_git_status, check_img_size, \
575     check_requirements, print_mutation, set_logging, one_cycle, colorstr
576 from utils.google_utils import attempt_download
577 from utils.loss import compute_loss
578 from utils.plots import plot_images, plot_labels, plot_results,
    plot_evolution
579 from utils.torch_utils import ModelEMA, select_device, intersect_dicts,
    torch_distributed_zero_first
580
```

```
581 logger = logging.getLogger(__name__)
582
583
584 def train(hyp, opt, device, tb_writer=None, wandb=None):
585     logger.info(colorstr('hyperparameters: ') + ', '.join(f'{k}={v}' for k,
        v in hyp.items()))
586     save_dir, epochs, batch_size, total_batch_size, weights, rank = \
587         Path(opt.save_dir), opt.epochs, opt.batch_size, opt.
        total_batch_size, opt.weights, opt.global_rank
588
589     # Directories
590     wdir = save_dir / 'weights'
591     wdir.mkdir(parents=True, exist_ok=True)  # make dir
592     last = wdir / 'last.pt'
593     best = wdir / 'best.pt'
594     results_file = save_dir / 'results.txt'
595
596     # Save run settings
597     with open(save_dir / 'hyp.yaml', 'w') as f:
598         yaml.dump(hyp, f, sort_keys=False)
599     with open(save_dir / 'opt.yaml', 'w') as f:
600         yaml.dump(vars(opt), f, sort_keys=False)
601
602     # Configure
603     plots = not opt.evolve  # create plots
604     cuda = device.type != 'cpu'
605     init_seeds(2 + rank)
606     with open(opt.data) as f:
607         data_dict = yaml.load(f, Loader=yaml.FullLoader)  # data dict
608     with torch_distributed_zero_first(rank):
609         check_dataset(data_dict)  # check
610     train_path = data_dict['train']
611     test_path = data_dict['val']
612     nc = 1 if opt.single_cls else int(data_dict['nc'])  # number of classes
613     names = ['item'] if opt.single_cls and len(data_dict['names']) != 1
        else data_dict['names']  # class names
614     assert len(names) == nc, '%g names found for nc=%g dataset in %s' % (
        len(names), nc, opt.data)  # check
615
616     # Model
617     pretrained = weights.endswith('.pt')
618     if pretrained:
619         with torch_distributed_zero_first(rank):
620             attempt_download(weights)  # download if not found locally
621         ckpt = torch.load(weights, map_location=device)  # load checkpoint
622         if hyp.get('anchors'):
623             ckpt['model'].yaml['anchors'] = round(hyp['anchors'])  # force
```

```
    autoanchor
624         model = Model(opt.cfg or ckpt['model'].yaml, ch=3, nc=nc).to(device
    ) # create
625         exclude = ['anchor'] if opt.cfg or hyp.get('anchors') else []  #
    exclude keys
626         state_dict = ckpt['model'].float().state_dict()  # to FP32
627         state_dict = intersect_dicts(state_dict, model.state_dict(),
    exclude=exclude)  # intersect
628         model.load_state_dict(state_dict, strict=False)  # load
629         logger.info('Transferred %g/%g items from %s' % (len(state_dict),
    len(model.state_dict()), weights))  # report
630     else:
631         model = Model(opt.cfg, ch=3, nc=nc).to(device)  # create
632
633     # Freeze
634     freeze = []  # parameter names to freeze (full or partial)
635     for k, v in model.named_parameters():
636         v.requires_grad = True  # train all layers
637         if any(x in k for x in freeze):
638             print('freezing %s' % k)
639             v.requires_grad = False
640
641     # Optimizer
642     nbs = 64  # nominal batch size
643     accumulate = max(round(nbs / total_batch_size), 1)  # accumulate loss
    before optimizing
644     hyp['weight_decay'] *= total_batch_size * accumulate / nbs  # scale
    weight_decay
645     logger.info(f"Scaled weight_decay = {hyp['weight_decay']}")
646
647     pg0, pg1, pg2 = [], [], []  # optimizer parameter groups
648     for k, v in model.named_modules():
649         if hasattr(v, 'bias') and isinstance(v.bias, nn.Parameter):
650             pg2.append(v.bias)  # biases
651         if isinstance(v, nn.BatchNorm2d):
652             pg0.append(v.weight)  # no decay
653         elif hasattr(v, 'weight') and isinstance(v.weight, nn.Parameter):
654             pg1.append(v.weight)  # apply decay
655
656     if opt.adam:
657         optimizer = optim.Adam(pg0, lr=hyp['lr0'], betas=(hyp['momentum'],
    0.999))  # adjust beta1 to momentum
658     else:
659         optimizer = optim.SGD(pg0, lr=hyp['lr0'], momentum=hyp['momentum'],
     nesterov=True)
660
661     optimizer.add_param_group({'params': pg1, 'weight_decay': hyp['
```

```
    weight_decay']})  # add pg1 with weight_decay
662     optimizer.add_param_group({'params': pg2})  # add pg2 (biases)
663     logger.info('Optimizer groups: %g .bias, %g conv.weight, %g other' % (
    len(pg2), len(pg1), len(pg0)))
664     del pg0, pg1, pg2
665 
666     # Scheduler https://arxiv.org/pdf/1812.01187.pdf
667     # https://pytorch.org/docs/stable/_modules/torch/optim/lr_scheduler.
    html#OneCycleLR
668     lf = one_cycle(1, hyp['lrf'], epochs)  # cosine 1->hyp['lrf']
669     scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)
670     # plot_lr_scheduler(optimizer, scheduler, epochs)
671 
672     # Logging
673     if rank in [-1, 0] and wandb and wandb.run is None:
674         opt.hyp = hyp  # add hyperparameters
675         wandb_run = wandb.init(config=opt, resume="allow",
676                                project='YOLOv3' if opt.project == 'runs/
    train' else Path(opt.project).stem,
677                                name=save_dir.stem,
678                                id=ckpt.get('wandb_id') if 'ckpt' in locals
    () else None)
679     loggers = {'wandb': wandb}  # loggers dict
680 
681     # Resume
682     start_epoch, best_fitness = 0, 0.0
683     if pretrained:
684         # Optimizer
685         if ckpt['optimizer'] is not None:
686             optimizer.load_state_dict(ckpt['optimizer'])
687             best_fitness = ckpt['best_fitness']
688 
689         # Results
690         if ckpt.get('training_results') is not None:
691             with open(results_file, 'w') as file:
692                 file.write(ckpt['training_results'])  # write results.txt
693 
694         # Epochs
695         start_epoch = ckpt['epoch'] + 1
696         if opt.resume:
697             assert start_epoch > 0, '%s training to %g epochs is finished,
    nothing to resume.' % (weights, epochs)
698         if epochs < start_epoch:
699             logger.info('%s has been trained for %g epochs. Fine-tuning for
     %g additional epochs.' %
700                         (weights, ckpt['epoch'], epochs))
701             epochs += ckpt['epoch']  # finetune additional epochs
```

```
702
703         del ckpt, state_dict
704
705     # Image sizes
706     gs = int(model.stride.max())  # grid size (max stride)
707     nl = model.model[-1].nl  # number of detection layers (used for scaling
        hyp['obj'])
708     imgsz, imgsz_test = [check_img_size(x, gs) for x in opt.img_size]  #
    verify imgsz are gs-multiples
709
710     # DP mode
711     if cuda and rank == -1 and torch.cuda.device_count() > 1:
712         model = torch.nn.DataParallel(model)
713
714     # SyncBatchNorm
715     if opt.sync_bn and cuda and rank != -1:
716         model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model).to(
    device)
717         logger.info('Using SyncBatchNorm()')
718
719     # EMA
720     ema = ModelEMA(model) if rank in [-1, 0] else None
721
722     # DDP mode
723     if cuda and rank != -1:
724         model = DDP(model, device_ids=[opt.local_rank], output_device=opt.
    local_rank)
725
726     # Trainloader
727     dataloader, dataset = create_dataloader(train_path, imgsz, batch_size,
    gs, opt,
728                                             hyp=hyp, augment=True, cache=
    opt.cache_images, rect=opt.rect, rank=rank,
729                                             world_size=opt.world_size,
    workers=opt.workers,
730                                             image_weights=opt.image_weights
    , quad=opt.quad, prefix=colorstr('train: '))
731     mlc = np.concatenate(dataset.labels, 0)[:, 0].max()  # max label class
732     nb = len(dataloader)  # number of batches
733     assert mlc < nc, 'Label class %g exceeds nc=%g in %s. Possible class
    labels are 0-%g' % (mlc, nc, opt.data, nc - 1)
734
735     # Process 0
736     if rank in [-1, 0]:
737         ema.updates = start_epoch * nb // accumulate  # set EMA updates
738         testloader = create_dataloader(test_path, imgsz_test,
    total_batch_size, gs, opt,  # testloader
```

```
739                                         hyp=hyp, cache=opt.cache_images and
    not opt.notest, rect=True, rank=-1,
740                                         world_size=opt.world_size, workers=
    opt.workers,
741                                         pad=0.5, prefix=colorstr('val: '))
    [0]
742
743         if not opt.resume:
744             labels = np.concatenate(dataset.labels, 0)
745             c = torch.tensor(labels[:, 0])  # classes
746             # cf = torch.bincount(c.long(), minlength=nc) + 1.  # frequency
747             # model._initialize_biases(cf.to(device))
748             if plots:
749                 plot_labels(labels, save_dir, loggers)
750                 if tb_writer:
751                     tb_writer.add_histogram('classes', c, 0)
752
753             # Anchors
754             if not opt.noautoanchor:
755                 check_anchors(dataset, model=model, thr=hyp['anchor_t'],
    imgsz=imgsz)
756
757     # Model parameters
758     hyp['box'] *= 3. / nl  # scale to layers
759     hyp['cls'] *= nc / 80. * 3. / nl  # scale to classes and layers
760     hyp['obj'] *= (imgsz / 640) ** 2 * 3. / nl  # scale to image size and
    layers
761     model.nc = nc  # attach number of classes to model
762     model.hyp = hyp  # attach hyperparameters to model
763     model.gr = 1.0  # iou loss ratio (obj_loss = 1.0 or iou)
764     model.class_weights = labels_to_class_weights(dataset.labels, nc).to(
    device) * nc  # attach class weights
765     model.names = names
766
767     # Start training
768     t0 = time.time()
769     nw = max(round(hyp['warmup_epochs'] * nb), 1000)  # number of warmup
    iterations, max(3 epochs, 1k iterations)
770     # nw = min(nw, (epochs - start_epoch) / 2 * nb)  # limit warmup to <
    1/2 of training
771     maps = np.zeros(nc)  # mAP per class
772     results = (0, 0, 0, 0, 0, 0, 0)  # P, R, mAP@.5, mAP@.5-.95, val_loss(
    box, obj, cls)
773     scheduler.last_epoch = start_epoch - 1  # do not move
774     scaler = amp.GradScaler(enabled=cuda)
775     logger.info(f'Image sizes {imgsz} train, {imgsz_test} test\n'
776                 f'Using {dataloader.num_workers} dataloader workers\n'
```

```
777                 f'Logging results to {save_dir}\n'
778                 f'Starting training for {epochs} epochs...')
779     for epoch in range(start_epoch, epochs):  # epoch
780         model.train()
781 
782         # Update image weights (optional)
783         if opt.image_weights:
784             # Generate indices
785             if rank in [-1, 0]:
786                 cw = model.class_weights.cpu().numpy() * (1 - maps) ** 2 / nc  # class weights
787                 iw = labels_to_image_weights(dataset.labels, nc=nc, class_weights=cw)  # image weights
788                 dataset.indices = random.choices(range(dataset.n), weights=iw, k=dataset.n)  # rand weighted idx
789             # Broadcast if DDP
790             if rank != -1:
791                 indices = (torch.tensor(dataset.indices) if rank == 0 else torch.zeros(dataset.n)).int()
792                 dist.broadcast(indices, 0)
793                 if rank != 0:
794                     dataset.indices = indices.cpu().numpy()
795 
796         # Update mosaic border
797         # b = int(random.uniform(0.25 * imgsz, 0.75 * imgsz + gs) // gs * gs)
798         # dataset.mosaic_border = [b - imgsz, -b]  # height, width borders
799 
800         mloss = torch.zeros(4, device=device)  # mean losses
801         if rank != -1:
802             dataloader.sampler.set_epoch(epoch)
803         pbar = enumerate(dataloader)
804         logger.info(('\n' + '%10s' * 8) % ('Epoch', 'gpu_mem', 'box', 'obj', 'cls', 'total', 'targets', 'img_size'))
805         if rank in [-1, 0]:
806             pbar = tqdm(pbar, total=nb)  # progress bar
807         optimizer.zero_grad()
808         for i, (imgs, targets, paths, _) in pbar:  # batch
809             ni = i + nb * epoch  # number integrated batches (since train start)
810             imgs = imgs.to(device, non_blocking=True).float() / 255.0  # uint8 to float32, 0-255 to 0.0-1.0
811 
812             # Warmup
813             if ni <= nw:
```

```
814                 xi = [0, nw]  # x interp
815                 # model.gr = np.interp(ni, xi, [0.0, 1.0])  # iou loss
    ratio (obj_loss = 1.0 or iou)
816                 accumulate = max(1, np.interp(ni, xi, [1, nbs /
    total_batch_size]).round())
817                 for j, x in enumerate(optimizer.param_groups):
818                     # bias lr falls from 0.1 to lr0, all other lrs rise
    from 0.0 to lr0
819                     x['lr'] = np.interp(ni, xi, [hyp['warmup_bias_lr'] if j
     == 2 else 0.0, x['initial_lr'] * lf(epoch)])
820                     if 'momentum' in x:
821                         x['momentum'] = np.interp(ni, xi, [hyp['
    warmup_momentum'], hyp['momentum']])
822
823             # Multi-scale
824             if opt.multi_scale:
825                 sz = random.randrange(imgsz * 0.5, imgsz * 1.5 + gs) // gs
    * gs  # size
826                 sf = sz / max(imgs.shape[2:])  # scale factor
827                 if sf != 1:
828                     ns = [math.ceil(x * sf / gs) * gs for x in imgs.shape
    [2:]]  # new shape (stretched to gs-multiple)
829                     imgs = F.interpolate(imgs, size=ns, mode='bilinear',
    align_corners=False)
830
831             # Forward
832             with amp.autocast(enabled=cuda):
833                 pred = model(imgs)  # forward
834                 loss, loss_items = compute_loss(pred, targets.to(device),
    model)  # loss scaled by batch_size
835                 if rank != -1:
836                     loss *= opt.world_size  # gradient averaged between
    devices in DDP mode
837                 if opt.quad:
838                     loss *= 4.
839
840             # Backward
841             scaler.scale(loss).backward()
842
843             # Optimize
844             if ni % accumulate == 0:
845                 scaler.step(optimizer)  # optimizer.step
846                 scaler.update()
847                 optimizer.zero_grad()
848                 if ema:
849                     ema.update(model)
850
```

```
851             # Print
852             if rank in [-1, 0]:
853                 mloss = (mloss * i + loss_items) / (i + 1)  # update mean
    losses
854                 mem = '%.3gG' % (torch.cuda.memory_reserved() / 1E9 if
    torch.cuda.is_available() else 0)  # (GB)
855                 s = ('%10s' * 2 + '%10.4g' * 6) % (
856                     '%g/%g' % (epoch, epochs - 1), mem, *mloss, targets.
    shape[0], imgs.shape[-1])
857                 pbar.set_description(s)
858
859                 # Plot
860                 if plots and ni < 3:
861                     f = save_dir / f'train_batch{ni}.jpg'  # filename
862                     Thread(target=plot_images, args=(imgs, targets, paths,
    f), daemon=True).start()
863                     # if tb_writer:
864                     #     tb_writer.add_image(f, result, dataformats='HWC',
     global_step=epoch)
865                     #     tb_writer.add_graph(model, imgs)  # add model to
    tensorboard
866                 elif plots and ni == 3 and wandb:
867                     wandb.log({"Mosaics": [wandb.Image(str(x), caption=x.
    name) for x in save_dir.glob('train*.jpg')]})
868
869             # end batch
    ------------------------------------------------------------------------------

870         # end epoch
    ------------------------------------------------------------------------------


871
872         # Scheduler
873         lr = [x['lr'] for x in optimizer.param_groups]  # for tensorboard
874         scheduler.step()
875
876         # DDP process 0 or single-GPU
877         if rank in [-1, 0]:
878             # mAP
879             if ema:
880                 ema.update_attr(model, include=['yaml', 'nc', 'hyp', 'gr',
    'names', 'stride', 'class_weights'])
881             final_epoch = epoch + 1 == epochs
882             if not opt.notest or final_epoch:  # Calculate mAP
883                 results, maps, times = test.test(opt.data,
884                                                  batch_size=
    total_batch_size,
```

```
885                                                 imgsz=imgsz_test,
886                                                 model=ema.ema,
887                                                 single_cls=opt.single_cls,
888                                                 dataloader=testloader,
889                                                 save_dir=save_dir,
890                                                 plots=plots and
    final_epoch,
891                                                 log_imgs=opt.log_imgs if
    wandb else 0)
892
893             # Write
894             with open(results_file, 'a') as f:
895                 f.write(s + '%10.4g' * 7 % results + '\n')  # P, R, mAP@.5,
     mAP@.5-.95, val_loss(box, obj, cls)
896             if len(opt.name) and opt.bucket:
897                 os.system('gsutil cp %s gs://%s/results/results%s.txt' % (
    results_file, opt.bucket, opt.name))
898
899             # Log
900             tags = ['train/box_loss', 'train/obj_loss', 'train/cls_loss',
    # train loss
901                     'metrics/precision', 'metrics/recall', 'metrics/mAP_0
    .5', 'metrics/mAP_0.5:0.95',
902                     'val/box_loss', 'val/obj_loss', 'val/cls_loss',  # val
    loss
903                     'x/lr0', 'x/lr1', 'x/lr2']  # params
904             for x, tag in zip(list(mloss[:-1]) + list(results) + lr, tags):
905                 if tb_writer:
906                     tb_writer.add_scalar(tag, x, epoch)  # tensorboard
907                 if wandb:
908                     wandb.log({tag: x})  # W&B
909
910             # Update best mAP
911             fi = fitness(np.array(results).reshape(1, -1))  # weighted
    combination of [P, R, mAP@.5, mAP@.5-.95]
912             if fi > best_fitness:
913                 best_fitness = fi
914
915             # Save model
916             save = (not opt.nosave) or (final_epoch and not opt.evolve)
917             if save:
918                 with open(results_file, 'r') as f:  # create checkpoint
919                     ckpt = {'epoch': epoch,
920                             'best_fitness': best_fitness,
921                             'training_results': f.read(),
922                             'model': ema.ema,
923                             'optimizer': None if final_epoch else optimizer
```

```
.state_dict(),
                        'wandb_id': wandb_run.id if wandb else None}

            # Save last, best and delete
            torch.save(ckpt, last)
            if best_fitness == fi:
                torch.save(ckpt, best)
            del ckpt
    # end epoch
```

```
# end training

if rank in [-1, 0]:
    # Strip optimizers
    final = best if best.exists() else last  # final model
    for f in [last, best]:
        if f.exists():
            strip_optimizer(f)  # strip optimizers
    if opt.bucket:
        os.system(f'gsutil cp {final} gs://{opt.bucket}/weights')  #
upload

    # Plots
    if plots:
        plot_results(save_dir=save_dir)  # save as results.png
        if wandb:
            files = ['results.png', 'precision_recall_curve.png', '
confusion_matrix.png']
            wandb.log({"Results": [wandb.Image(str(save_dir / f),
caption=f) for f in files
                                   if (save_dir / f).exists()]})
            if opt.log_artifacts:
                wandb.log_artifact(artifact_or_path=str(final), type='
model', name=save_dir.stem)

    # Test best.pt
    logger.info('%g epochs completed in %.3f hours.\n' % (epoch -
start_epoch + 1, (time.time() - t0) / 3600))
    if opt.data.endswith('coco.yaml') and nc == 80:  # if COCO
        for conf, iou, save_json in ([0.25, 0.45, False], [0.001, 0.65,
 True]):  # speed, mAP tests
            results, _, _ = test.test(opt.data,
                                      batch_size=total_batch_size,
                                      imgsz=imgsz_test,
                                      conf_thres=conf,
                                      iou_thres=iou,
```

```
962                                             model=attempt_load(final, device)
    .half(),
963                                             single_cls=opt.single_cls,
964                                             dataloader=testloader,
965                                             save_dir=save_dir,
966                                             save_json=save_json,
967                                             plots=False)
968
969     else:
970         dist.destroy_process_group()
971
972     wandb.run.finish() if wandb and wandb.run else None
973     torch.cuda.empty_cache()
974     return results
975
976
977 if __name__ == '__main__':
978     parser = argparse.ArgumentParser()
979     parser.add_argument('--weights', type=str, default='yolov3.pt', help='
    initial weights path')
980     parser.add_argument('--cfg', type=str, default='', help='model.yaml
    path')
981     parser.add_argument('--data', type=str, default='data/coco128.yaml',
    help='data.yaml path')
982     parser.add_argument('--hyp', type=str, default='data/hyp.scratch.yaml',
     help='hyperparameters path')
983     parser.add_argument('--epochs', type=int, default=300)
984     parser.add_argument('--batch-size', type=int, default=16, help='total
    batch size for all GPUs')
985     parser.add_argument('--img-size', nargs='+', type=int, default=[640,
    640], help='[train, test] image sizes')
986     parser.add_argument('--rect', action='store_true', help='rectangular
    training')
987     parser.add_argument('--resume', nargs='?', const=True, default=False,
    help='resume most recent training')
988     parser.add_argument('--nosave', action='store_true', help='only save
    final checkpoint')
989     parser.add_argument('--notest', action='store_true', help='only test
    final epoch')
990     parser.add_argument('--noautoanchor', action='store_true', help='
    disable autoanchor check')
991     parser.add_argument('--evolve', action='store_true', help='evolve
    hyperparameters')
992     parser.add_argument('--bucket', type=str, default='', help='gsutil
    bucket')
993     parser.add_argument('--cache-images', action='store_true', help='cache
    images for faster training')
```

```
994     parser.add_argument('--image-weights', action='store_true', help='use
        weighted image selection for training')
995     parser.add_argument('--device', default='', help='cuda device, i.e. 0
        or 0,1,2,3 or cpu')
996     parser.add_argument('--multi-scale', action='store_true', help='vary
        img-size +/- 50%%')
997     parser.add_argument('--single-cls', action='store_true', help='train
        multi-class data as single-class')
998     parser.add_argument('--adam', action='store_true', help='use torch.
        optim.Adam() optimizer')
999     parser.add_argument('--sync-bn', action='store_true', help='use
        SyncBatchNorm, only available in DDP mode')
1000    parser.add_argument('--local_rank', type=int, default=-1, help='DDP
        parameter, do not modify')
1001    parser.add_argument('--log-imgs', type=int, default=16, help='number of
         images for W&B logging, max 100')
1002    parser.add_argument('--log-artifacts', action='store_true', help='log
        artifacts, i.e. final trained model')
1003    parser.add_argument('--workers', type=int, default=8, help='maximum
        number of dataloader workers')
1004    parser.add_argument('--project', default='runs/train', help='save to
        project/name')
1005    parser.add_argument('--name', default='exp', help='save to project/name
        ')
1006    parser.add_argument('--exist-ok', action='store_true', help='existing
        project/name ok, do not increment')
1007    parser.add_argument('--quad', action='store_true', help='quad
        dataloader')
1008    opt = parser.parse_args()
1009
1010    # Set DDP variables
1011    opt.world_size = int(os.environ['WORLD_SIZE']) if 'WORLD_SIZE' in os.
        environ else 1
1012    opt.global_rank = int(os.environ['RANK']) if 'RANK' in os.environ else
        -1
1013    set_logging(opt.global_rank)
1014    if opt.global_rank in [-1, 0]:
1015        check_git_status()
1016        check_requirements()
1017
1018    # Resume
1019    if opt.resume:  # resume an interrupted run
1020        ckpt = opt.resume if isinstance(opt.resume, str) else
        get_latest_run()  # specified or most recent path
1021        assert os.path.isfile(ckpt), 'ERROR: --resume checkpoint does not
        exist'
1022        apriori = opt.global_rank, opt.local_rank
```

```
1023            with open(Path(ckpt).parent.parent / 'opt.yaml') as f:
1024                opt = argparse.Namespace(**yaml.load(f, Loader=yaml.FullLoader)
    ) # replace
1025            opt.cfg, opt.weights, opt.resume, opt.global_rank, opt.local_rank =
     '', ckpt, True, *apriori # reinstate
1026            logger.info('Resuming training from %s' % ckpt)
1027        else:
1028            # opt.hyp = opt.hyp or ('hyp.finetune.yaml' if opt.weights else '
    hyp.scratch.yaml')
1029            opt.data, opt.cfg, opt.hyp = check_file(opt.data), check_file(opt.
    cfg), check_file(opt.hyp) # check files
1030            assert len(opt.cfg) or len(opt.weights), 'either --cfg or --weights
     must be specified'
1031            opt.img_size.extend([opt.img_size[-1]] * (2 - len(opt.img_size)))
    # extend to 2 sizes (train, test)
1032            opt.name = 'evolve' if opt.evolve else opt.name
1033            opt.save_dir = increment_path(Path(opt.project) / opt.name,
    exist_ok=opt.exist_ok | opt.evolve) # increment run
1034
1035        # DDP mode
1036        opt.total_batch_size = opt.batch_size
1037        device = select_device(opt.device, batch_size=opt.batch_size)
1038        if opt.local_rank != -1:
1039            assert torch.cuda.device_count() > opt.local_rank
1040            torch.cuda.set_device(opt.local_rank)
1041            device = torch.device('cuda', opt.local_rank)
1042            dist.init_process_group(backend='nccl', init_method='env://') #
    distributed backend
1043            assert opt.batch_size % opt.world_size == 0, '--batch-size must be
    multiple of CUDA device count'
1044            opt.batch_size = opt.total_batch_size // opt.world_size
1045
1046        # Hyperparameters
1047        with open(opt.hyp) as f:
1048            hyp = yaml.load(f, Loader=yaml.FullLoader) # load hyps
1049
1050        # Train
1051        logger.info(opt)
1052        try:
1053            import wandb
1054        except ImportError:
1055            wandb = None
1056            prefix = colorstr('wandb: ')
1057            logger.info(f"{prefix}Install Weights & Biases for YOLOv3 logging
    with 'pip install wandb' (recommended)")
1058        if not opt.evolve:
1059            tb_writer = None # init loggers
```

```
1060        if opt.global_rank in [-1, 0]:
1061            logger.info(f'Start Tensorboard with "tensorboard --logdir {opt
    .project}", view at http://localhost:6006/')
1062            tb_writer = SummaryWriter(opt.save_dir)  # Tensorboard
1063        train(hyp, opt, device, tb_writer, wandb)
1064
1065    # Evolve hyperparameters (optional)
1066    else:
1067        # Hyperparameter evolution metadata (mutation scale 0-1,
    lower_limit, upper_limit)
1068        meta = {'lr0': (1, 1e-5, 1e-1),  # initial learning rate (SGD=1E-2,
     Adam=1E-3)
1069                'lrf': (1, 0.01, 1.0),  # final OneCycleLR learning rate (
    lr0 * lrf)
1070                'momentum': (0.3, 0.6, 0.98),  # SGD momentum/Adam beta1
1071                'weight_decay': (1, 0.0, 0.001),  # optimizer weight decay
1072                'warmup_epochs': (1, 0.0, 5.0),  # warmup epochs (fractions
     ok)
1073                'warmup_momentum': (1, 0.0, 0.95),  # warmup initial
    momentum
1074                'warmup_bias_lr': (1, 0.0, 0.2),  # warmup initial bias lr
1075                'box': (1, 0.02, 0.2),  # box loss gain
1076                'cls': (1, 0.2, 4.0),  # cls loss gain
1077                'cls_pw': (1, 0.5, 2.0),  # cls BCELoss positive_weight
1078                'obj': (1, 0.2, 4.0),  # obj loss gain (scale with pixels)
1079                'obj_pw': (1, 0.5, 2.0),  # obj BCELoss positive_weight
1080                'iou_t': (0, 0.1, 0.7),  # IoU training threshold
1081                'anchor_t': (1, 2.0, 8.0),  # anchor-multiple threshold
1082                'anchors': (2, 2.0, 10.0),  # anchors per output grid (0 to
     ignore)
1083                'fl_gamma': (0, 0.0, 2.0),  # focal loss gamma (
    efficientDet default gamma=1.5)
1084                'hsv_h': (1, 0.0, 0.1),  # image HSV-Hue augmentation (
    fraction)
1085                'hsv_s': (1, 0.0, 0.9),  # image HSV-Saturation
    augmentation (fraction)
1086                'hsv_v': (1, 0.0, 0.9),  # image HSV-Value augmentation (
    fraction)
1087                'degrees': (1, 0.0, 45.0),  # image rotation (+/- deg)
1088                'translate': (1, 0.0, 0.9),  # image translation (+/-
    fraction)
1089                'scale': (1, 0.0, 0.9),  # image scale (+/- gain)
1090                'shear': (1, 0.0, 10.0),  # image shear (+/- deg)
1091                'perspective': (0, 0.0, 0.001),  # image perspective (+/-
    fraction), range 0-0.001
1092                'flipud': (1, 0.0, 1.0),  # image flip up-down (probability
    )
```

```
1093                'fliplr': (0, 0.0, 1.0),  # image flip left-right (
    probability)
1094                'mosaic': (1, 0.0, 1.0),  # image mixup (probability)
1095                'mixup': (1, 0.0, 1.0)}  # image mixup (probability)
1096
1097        assert opt.local_rank == -1, 'DDP mode not implemented for --evolve
    '
1098        opt.notest, opt.nosave = True, True  # only test/save final epoch
1099        # ei = [isinstance(x, (int, float)) for x in hyp.values()]  #
    evolvable indices
1100        yaml_file = Path(opt.save_dir) / 'hyp_evolved.yaml'  # save best
    result here
1101        if opt.bucket:
1102            os.system('gsutil cp gs://%s/evolve.txt .' % opt.bucket)  #
    download evolve.txt if exists
1103
1104        for _ in range(300):  # generations to evolve
1105            if Path('evolve.txt').exists():  # if evolve.txt exists: select
     best hyps and mutate
1106                # Select parent(s)
1107                parent = 'single'  # parent selection method: 'single' or '
    weighted'
1108                x = np.loadtxt('evolve.txt', ndmin=2)
1109                n = min(5, len(x))  # number of previous results to
    consider
1110                x = x[np.argsort(-fitness(x))][:n]  # top n mutations
1111                w = fitness(x) - fitness(x).min()  # weights
1112                if parent == 'single' or len(x) == 1:
1113                    # x = x[random.randint(0, n - 1)]  # random selection
1114                    x = x[random.choices(range(n), weights=w)[0]]  #
    weighted selection
1115                elif parent == 'weighted':
1116                    x = (x * w.reshape(n, 1)).sum(0) / w.sum()  # weighted
    combination
1117
1118                # Mutate
1119                mp, s = 0.8, 0.2  # mutation probability, sigma
1120                npr = np.random
1121                npr.seed(int(time.time()))
1122                g = np.array([x[0] for x in meta.values()])  # gains 0-1
1123                ng = len(meta)
1124                v = np.ones(ng)
1125                while all(v == 1):  # mutate until a change occurs (prevent
     duplicates)
1126                    v = (g * (npr.random(ng) < mp) * npr.randn(ng) * npr.
    random() * s + 1).clip(0.3, 3.0)
1127                for i, k in enumerate(hyp.keys()):  # plt.hist(v.ravel(),
```

```
    300)
                    hyp[k] = float(x[i + 7] * v[i])  # mutate

            # Constrain to limits
            for k, v in meta.items():
                hyp[k] = max(hyp[k], v[1])  # lower limit
                hyp[k] = min(hyp[k], v[2])  # upper limit
                hyp[k] = round(hyp[k], 5)  # significant digits

            # Train mutation
            results = train(hyp.copy(), opt, device, wandb=wandb)

            # Write mutation results
            print_mutation(hyp.copy(), results, yaml_file, opt.bucket)

        # Plot results
        plot_evolution(yaml_file)
        print(f'Hyperparameter evolution complete. Best results saved as: {
    yaml_file}\n'
              f'Command to train a new model with these hyperparameters: $
    python train.py --hyp {yaml_file}')


# This file contains modules common to various models

import math

import numpy as np
import requests
import torch
import torch.nn as nn
from PIL import Image, ImageDraw

from utils.datasets import letterbox
from utils.general import non_max_suppression, make_divisible, scale_coords
    , xyxy2xywh
from utils.plots import color_list


def autopad(k, p=None):  # kernel, padding
    # Pad to 'same'
    if p is None:
        p = k // 2 if isinstance(k, int) else [x // 2 for x in k]  # auto-
    pad
    return p


```

```
1170 def DWConv(c1, c2, k=1, s=1, act=True):
1171     # Depthwise convolution
1172     return Conv(c1, c2, k, s, g=math.gcd(c1, c2), act=act)
1173
1174
1175 class Conv(nn.Module):
1176     # Standard convolution
1177     def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=True):  # ch_in,
     ch_out, kernel, stride, padding, groups
1178         super(Conv, self).__init__()
1179         self.conv = nn.Conv2d(c1, c2, k, s, autopad(k, p), groups=g, bias=
     False)
1180         self.bn = nn.BatchNorm2d(c2)
1181         self.act = nn.LeakyReLU(0.1) if act is True else (act if isinstance
     (act, nn.Module) else nn.Identity())
1182
1183     def forward(self, x):
1184         return self.act(self.bn(self.conv(x)))
1185
1186     def fuseforward(self, x):
1187         return self.act(self.conv(x))
1188
1189
1190 class Bottleneck(nn.Module):
1191     # Standard bottleneck
1192     def __init__(self, c1, c2, shortcut=True, g=1, e=0.5):  # ch_in, ch_out
     , shortcut, groups, expansion
1193         super(Bottleneck, self).__init__()
1194         c_ = int(c2 * e)  # hidden channels
1195         self.cv1 = Conv(c1, c_, 1, 1)
1196         self.cv2 = Conv(c_, c2, 3, 1, g=g)
1197         self.add = shortcut and c1 == c2
1198
1199     def forward(self, x):
1200         return x + self.cv2(self.cv1(x)) if self.add else self.cv2(self.cv1
     (x))
1201
1202
1203 class BottleneckCSP(nn.Module):
1204     # CSP Bottleneck https://github.com/WongKinYiu/
     CrossStagePartialNetworks
1205     def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):  # ch_in,
     ch_out, number, shortcut, groups, expansion
1206         super(BottleneckCSP, self).__init__()
1207         c_ = int(c2 * e)  # hidden channels
1208         self.cv1 = Conv(c1, c_, 1, 1)
1209         self.cv2 = nn.Conv2d(c1, c_, 1, 1, bias=False)
```

```
1210         self.cv3 = nn.Conv2d(c_, c_, 1, 1, bias=False)
1211         self.cv4 = Conv(2 * c_, c2, 1, 1)
1212         self.bn = nn.BatchNorm2d(2 * c_)  # applied to cat(cv2, cv3)
1213         self.act = nn.LeakyReLU(0.1, inplace=True)
1214         self.m = nn.Sequential(*[Bottleneck(c_, c_, shortcut, g, e=1.0) for
     _ in range(n)])
1215 
1216     def forward(self, x):
1217         y1 = self.cv3(self.m(self.cv1(x)))
1218         y2 = self.cv2(x)
1219         return self.cv4(self.act(self.bn(torch.cat((y1, y2), dim=1))))
1220 
1221 
1222 class C3(nn.Module):
1223     # CSP Bottleneck with 3 convolutions
1224     def __init__(self, c1, c2, n=1, shortcut=True, g=1, e=0.5):  # ch_in,
    ch_out, number, shortcut, groups, expansion
1225         super(C3, self).__init__()
1226         c_ = int(c2 * e)  # hidden channels
1227         self.cv1 = Conv(c1, c_, 1, 1)
1228         self.cv2 = Conv(c1, c_, 1, 1)
1229         self.cv3 = Conv(2 * c_, c2, 1)  # act=FReLU(c2)
1230         self.m = nn.Sequential(*[Bottleneck(c_, c_, shortcut, g, e=1.0) for
     _ in range(n)])
1231         # self.m = nn.Sequential(*[CrossConv(c_, c_, 3, 1, g, 1.0, shortcut
    ) for _ in range(n)])
1232 
1233     def forward(self, x):
1234         return self.cv3(torch.cat((self.m(self.cv1(x)), self.cv2(x)), dim
    =1))
1235 
1236 
1237 class SPP(nn.Module):
1238     # Spatial pyramid pooling layer used in YOLOv3-SPP
1239     def __init__(self, c1, c2, k=(5, 9, 13)):
1240         super(SPP, self).__init__()
1241         c_ = c1 // 2  # hidden channels
1242         self.cv1 = Conv(c1, c_, 1, 1)
1243         self.cv2 = Conv(c_ * (len(k) + 1), c2, 1, 1)
1244         self.m = nn.ModuleList([nn.MaxPool2d(kernel_size=x, stride=1,
    padding=x // 2) for x in k])
1245 
1246     def forward(self, x):
1247         x = self.cv1(x)
1248         return self.cv2(torch.cat([x] + [m(x) for m in self.m], 1))
1249 
1250 
```

```
1251 class Focus(nn.Module):
1252     # Focus wh information into c-space
1253     def __init__(self, c1, c2, k=1, s=1, p=None, g=1, act=True):  # ch_in,
     ch_out, kernel, stride, padding, groups
1254         super(Focus, self).__init__()
1255         self.conv = Conv(c1 * 4, c2, k, s, p, g, act)
1256         # self.contract = Contract(gain=2)
1257 
1258     def forward(self, x):  # x(b,c,w,h) -> y(b,4c,w/2,h/2)
1259         return self.conv(torch.cat([x[..., ::2, ::2], x[..., 1::2, ::2], x
     [..., ::2, 1::2], x[..., 1::2, 1::2]], 1))
1260         # return self.conv(self.contract(x))
1261 
1262 
1263 class Contract(nn.Module):
1264     # Contract width-height into channels, i.e. x(1,64,80,80) to x
     (1,256,40,40)
1265     def __init__(self, gain=2):
1266         super().__init__()
1267         self.gain = gain
1268 
1269     def forward(self, x):
1270         N, C, H, W = x.size()  # assert (H / s == 0) and (W / s == 0), '
     Indivisible gain'
1271         s = self.gain
1272         x = x.view(N, C, H // s, s, W // s, s)  # x(1,64,40,2,40,2)
1273         x = x.permute(0, 3, 5, 1, 2, 4).contiguous()  # x(1,2,2,64,40,40)
1274         return x.view(N, C * s * s, H // s, W // s)  # x(1,256,40,40)
1275 
1276 
1277 class Expand(nn.Module):
1278     # Expand channels into width-height, i.e. x(1,64,80,80) to x
     (1,16,160,160)
1279     def __init__(self, gain=2):
1280         super().__init__()
1281         self.gain = gain
1282 
1283     def forward(self, x):
1284         N, C, H, W = x.size()  # assert C / s ** 2 == 0, 'Indivisible gain'
1285         s = self.gain
1286         x = x.view(N, s, s, C // s ** 2, H, W)  # x(1,2,2,16,80,80)
1287         x = x.permute(0, 3, 4, 1, 5, 2).contiguous()  # x(1,16,80,2,80,2)
1288         return x.view(N, C // s ** 2, H * s, W * s)  # x(1,16,160,160)
1289 
1290 
1291 class Concat(nn.Module):
1292     # Concatenate a list of tensors along dimension
```

```
1293     def __init__(self, dimension=1):
1294         super(Concat, self).__init__()
1295         self.d = dimension
1296 
1297     def forward(self, x):
1298         return torch.cat(x, self.d)
1299 
1300 
1301 class NMS(nn.Module):
1302     # Non-Maximum Suppression (NMS) module
1303     conf = 0.25  # confidence threshold
1304     iou = 0.45  # IoU threshold
1305     classes = None  # (optional list) filter by class
1306 
1307     def __init__(self):
1308         super(NMS, self).__init__()
1309 
1310     def forward(self, x):
1311         return non_max_suppression(x[0], conf_thres=self.conf, iou_thres=
    self.iou, classes=self.classes)
1312 
1313 
1314 class autoShape(nn.Module):
1315     # input-robust model wrapper for passing cv2/np/PIL/torch inputs.
    Includes preprocessing, inference and NMS
1316     img_size = 640  # inference size (pixels)
1317     conf = 0.25  # NMS confidence threshold
1318     iou = 0.45  # NMS IoU threshold
1319     classes = None  # (optional list) filter by class
1320 
1321     def __init__(self, model):
1322         super(autoShape, self).__init__()
1323         self.model = model.eval()
1324 
1325     def autoshape(self):
1326         print('autoShape already enabled, skipping... ')  # model already
    converted to model.autoshape()
1327         return self
1328 
1329     def forward(self, imgs, size=640, augment=False, profile=False):
1330         # Inference from various sources. For height=720, width=1280, RGB
    images example inputs are:
1331         #   filename:   imgs = 'data/samples/zidane.jpg'
1332         #   URI:             = 'https://github.com/ultralytics/yolov5/
    releases/download/v1.0/zidane.jpg'
1333         #   OpenCV:          = cv2.imread('image.jpg')[:,:,::-1]  # HWC BGR
     to RGB x(720,1280,3)
```

```
1334        #   PIL:                = Image.open('image.jpg')  # HWC x(720,1280,3)
1335        #   numpy:              = np.zeros((720,1280,3))  # HWC
1336        #   torch:              = torch.zeros(16,3,720,1280)  # BCHW
1337        #   multiple:           = [Image.open('image1.jpg'), Image.open('
    image2.jpg'), ...]  # list of images
1338
1339        p = next(self.model.parameters())  # for device and type
1340        if isinstance(imgs, torch.Tensor):  # torch
1341            return self.model(imgs.to(p.device).type_as(p), augment,
    profile)  # inference
1342
1343        # Pre-process
1344        n, imgs = (len(imgs), imgs) if isinstance(imgs, list) else (1, [
    imgs])  # number of images, list of images
1345        shape0, shape1 = [], []  # image and inference shapes
1346        for i, im in enumerate(imgs):
1347            if isinstance(im, str):  # filename or uri
1348                im = Image.open(requests.get(im, stream=True).raw if im.
    startswith('http') else im)  # open
1349            im = np.array(im)  # to numpy
1350            if im.shape[0] < 5:  # image in CHW
1351                im = im.transpose((1, 2, 0))  # reverse dataloader .
    transpose(2, 0, 1)
1352            im = im[:, :, :3] if im.ndim == 3 else np.tile(im[:, :, None],
    3)  # enforce 3ch input
1353            s = im.shape[:2]  # HWC
1354            shape0.append(s)  # image shape
1355            g = (size / max(s))  # gain
1356            shape1.append([y * g for y in s])
1357            imgs[i] = im  # update
1358        shape1 = [make_divisible(x, int(self.stride.max())) for x in np.
    stack(shape1, 0).max(0)]  # inference shape
1359        x = [letterbox(im, new_shape=shape1, auto=False)[0] for im in imgs]
      # pad
1360        x = np.stack(x, 0) if n > 1 else x[0][None]  # stack
1361        x = np.ascontiguousarray(x.transpose((0, 3, 1, 2)))  # BHWC to BCHW
1362        x = torch.from_numpy(x).to(p.device).type_as(p) / 255.  # uint8 to
    fp16/32
1363
1364        # Inference
1365        with torch.no_grad():
1366            y = self.model(x, augment, profile)[0]  # forward
1367        y = non_max_suppression(y, conf_thres=self.conf, iou_thres=self.iou
    , classes=self.classes)  # NMS
1368
1369        # Post-process
1370        for i in range(n):
```

```
1371            scale_coords(shape1, y[i][:, :4], shape0[i])
1372
1373        return Detections(imgs, y, self.names)
1374
1375
1376 class Detections:
1377     # detections class for YOLOv5 inference results
1378     def __init__(self, imgs, pred, names=None):
1379         super(Detections, self).__init__()
1380         d = pred[0].device  # device
1381         gn = [torch.tensor([*[im.shape[i] for i in [1, 0, 1, 0]], 1., 1.],
     device=d) for im in imgs]  # normalizations
1382         self.imgs = imgs  # list of images as numpy arrays
1383         self.pred = pred  # list of tensors pred[0] = (xyxy, conf, cls)
1384         self.names = names  # class names
1385         self.xyxy = pred  # xyxy pixels
1386         self.xywh = [xyxy2xywh(x) for x in pred]  # xywh pixels
1387         self.xyxyn = [x / g for x, g in zip(self.xyxy, gn)]  # xyxy
     normalized
1388         self.xywhn = [x / g for x, g in zip(self.xywh, gn)]  # xywh
     normalized
1389         self.n = len(self.pred)
1390
1391     def display(self, pprint=False, show=False, save=False, render=False):
1392         colors = color_list()
1393         for i, (img, pred) in enumerate(zip(self.imgs, self.pred)):
1394             str = f'Image {i + 1}/{len(self.pred)}: {img.shape[0]}x{img.
     shape[1]} '
1395             if pred is not None:
1396                 for c in pred[:, -1].unique():
1397                     n = (pred[:, -1] == c).sum()  # detections per class
1398                     str += f'{n} {self.names[int(c)]}s, '  # add to string
1399                 if show or save or render:
1400                     img = Image.fromarray(img.astype(np.uint8)) if
     isinstance(img, np.ndarray) else img  # from np
1401                     for *box, conf, cls in pred:  # xyxy, confidence, class
1402                         # str += '%s %.2f, ' % (names[int(cls)], conf)  #
     label
1403                         ImageDraw.Draw(img).rectangle(box, width=4, outline
     =colors[int(cls) % 10])  # plot
1404             if pprint:
1405                 print(str)
1406             if show:
1407                 img.show(f'Image {i}')  # show
1408             if save:
1409                 f = f'results{i}.jpg'
1410                 str += f"saved to '{f}'"
```

```
1411                 img.save(f)  # save
1412             if render:
1413                 self.imgs[i] = np.asarray(img)
1414 
1415     def print(self):
1416         self.display(pprint=True)  # print results
1417 
1418     def show(self):
1419         self.display(show=True)  # show results
1420 
1421     def save(self):
1422         self.display(save=True)  # save results
1423 
1424     def render(self):
1425         self.display(render=True)  # render results
1426         return self.imgs
1427 
1428     def __len__(self):
1429         return self.n
1430 
1431     def tolist(self):
1432         # return a list of Detections objects, i.e. 'for result in results.
    tolist():'
1433         x = [Detections([self.imgs[i]], [self.pred[i]], self.names) for i
    in range(self.n)]
1434         for d in x:
1435             for k in ['imgs', 'pred', 'xyxy', 'xyxyn', 'xywh', 'xywhn']:
1436                 setattr(d, k, getattr(d, k)[0])  # pop out of list
1437         return x
1438 
1439 
1440 class Classify(nn.Module):
1441     # Classification head, i.e. x(b,c1,20,20) to x(b,c2)
1442     def __init__(self, c1, c2, k=1, s=1, p=None, g=1):  # ch_in, ch_out,
    kernel, stride, padding, groups
1443         super(Classify, self).__init__()
1444         self.aap = nn.AdaptiveAvgPool2d(1)  # to x(b,c1,1,1)
1445         self.conv = nn.Conv2d(c1, c2, k, s, autopad(k, p), groups=g)  # to
    x(b,c2,1,1)
1446         self.flat = nn.Flatten()
1447 
1448     def forward(self, x):
1449         z = torch.cat([self.aap(y) for y in (x if isinstance(x, list) else
    [x])], 1)  # cat if list
1450         return self.flat(self.conv(z))  # flatten to x(b,c2)
1451 
1452 
```

```
# This file contains experimental modules

import numpy as np
import torch
import torch.nn as nn

from models.common import Conv, DWConv
from utils.google_utils import attempt_download


class CrossConv(nn.Module):
    # Cross Convolution Downsample
    def __init__(self, c1, c2, k=3, s=1, g=1, e=1.0, shortcut=False):
        # ch_in, ch_out, kernel, stride, groups, expansion, shortcut
        super(CrossConv, self).__init__()
        c_ = int(c2 * e)  # hidden channels
        self.cv1 = Conv(c1, c_, (1, k), (1, s))
        self.cv2 = Conv(c_, c2, (k, 1), (s, 1), g=g)
        self.add = shortcut and c1 == c2

    def forward(self, x):
        return x + self.cv2(self.cv1(x)) if self.add else self.cv2(self.cv1(x))


class Sum(nn.Module):
    # Weighted sum of 2 or more layers https://arxiv.org/abs/1911.09070
    def __init__(self, n, weight=False):  # n: number of inputs
        super(Sum, self).__init__()
        self.weight = weight  # apply weights boolean
        self.iter = range(n - 1)  # iter object
        if weight:
            self.w = nn.Parameter(-torch.arange(1., n) / 2, requires_grad=True)  # layer weights

    def forward(self, x):
        y = x[0]  # no weight
        if self.weight:
            w = torch.sigmoid(self.w) * 2
            for i in self.iter:
                y = y + x[i + 1] * w[i]
        else:
            for i in self.iter:
                y = y + x[i + 1]
        return y


```

```
1498 class GhostConv(nn.Module):
1499     # Ghost Convolution https://github.com/huawei-noah/ghostnet
1500     def __init__(self, c1, c2, k=1, s=1, g=1, act=True):  # ch_in, ch_out,
     kernel, stride, groups
1501         super(GhostConv, self).__init__()
1502         c_ = c2 // 2  # hidden channels
1503         self.cv1 = Conv(c1, c_, k, s, None, g, act)
1504         self.cv2 = Conv(c_, c_, 5, 1, None, c_, act)
1505 
1506     def forward(self, x):
1507         y = self.cv1(x)
1508         return torch.cat([y, self.cv2(y)], 1)
1509 
1510 
1511 class GhostBottleneck(nn.Module):
1512     # Ghost Bottleneck https://github.com/huawei-noah/ghostnet
1513     def __init__(self, c1, c2, k, s):
1514         super(GhostBottleneck, self).__init__()
1515         c_ = c2 // 2
1516         self.conv = nn.Sequential(GhostConv(c1, c_, 1, 1),  # pw
1517                                   DWConv(c_, c_, k, s, act=False) if s == 2
     else nn.Identity(),  # dw
1518                                   GhostConv(c_, c2, 1, 1, act=False))  # pw
     -linear
1519         self.shortcut = nn.Sequential(DWConv(c1, c1, k, s, act=False),
1520                                       Conv(c1, c2, 1, 1, act=False)) if s
     == 2 else nn.Identity()
1521 
1522     def forward(self, x):
1523         return self.conv(x) + self.shortcut(x)
1524 
1525 
1526 class MixConv2d(nn.Module):
1527     # Mixed Depthwise Conv https://arxiv.org/abs/1907.09595
1528     def __init__(self, c1, c2, k=(1, 3), s=1, equal_ch=True):
1529         super(MixConv2d, self).__init__()
1530         groups = len(k)
1531         if equal_ch:  # equal c_ per group
1532             i = torch.linspace(0, groups - 1E-6, c2).floor()  # c2 indices
1533             c_ = [(i == g).sum() for g in range(groups)]  # intermediate
     channels
1534         else:  # equal weight.numel() per group
1535             b = [c2] + [0] * groups
1536             a = np.eye(groups + 1, groups, k=-1)
1537             a -= np.roll(a, 1, axis=1)
1538             a *= np.array(k) ** 2
1539             a[0] = 1
```

```
            c_ = np.linalg.lstsq(a, b, rcond=None)[0].round()  # solve for equal weight indices, ax = b

            self.m = nn.ModuleList([nn.Conv2d(c1, int(c_[g]), k[g], s, k[g] // 2, bias=False) for g in range(groups)])
            self.bn = nn.BatchNorm2d(c2)
            self.act = nn.LeakyReLU(0.1, inplace=True)

        def forward(self, x):
            return x + self.act(self.bn(torch.cat([m(x) for m in self.m], 1)))


class Ensemble(nn.ModuleList):
    # Ensemble of models
    def __init__(self):
        super(Ensemble, self).__init__()

    def forward(self, x, augment=False):
        y = []
        for module in self:
            y.append(module(x, augment)[0])
        # y = torch.stack(y).max(0)[0]  # max ensemble
        # y = torch.stack(y).mean(0)  # mean ensemble
        y = torch.cat(y, 1)  # nms ensemble
        return y, None  # inference, train output


def attempt_load(weights, map_location=None):
    # Loads an ensemble of models weights=[a,b,c] or a single model weights=[a] or weights=a
    model = Ensemble()
    for w in weights if isinstance(weights, list) else [weights]:
        attempt_download(w)
        model.append(torch.load(w, map_location=map_location)['model'].float().fuse().eval())  # load FP32 model

    # Compatibility updates
    for m in model.modules():
        if type(m) in [nn.Hardswish, nn.LeakyReLU, nn.ReLU, nn.ReLU6]:
            m.inplace = True  # pytorch 1.7.0 compatibility
        elif type(m) is Conv:
            m._non_persistent_buffers_set = set()  # pytorch 1.6.0 compatibility

    if len(model) == 1:
        return model[-1]  # return model
    else:
```

```
1582         print('Ensemble created with %s\n' % weights)
1583         for k in ['names', 'stride']:
1584             setattr(model, k, getattr(model[-1], k))
1585         return model  # return ensemble
1586 
1587 
1588 """Exports a YOLOv5 *.pt model to ONNX and TorchScript formats
1589 
1590 Usage:
1591     $ export PYTHONPATH="$PWD" && python models/export.py --weights ./
    weights/yolov3.pt --img 640 --batch 1
1592 """
1593 
1594 import argparse
1595 import sys
1596 import time
1597 
1598 sys.path.append('./')  # to run '$ python *.py' files in subdirectories
1599 
1600 import torch
1601 import torch.nn as nn
1602 
1603 import models
1604 from models.experimental import attempt_load
1605 from utils.activations import Hardswish, SiLU
1606 from utils.general import set_logging, check_img_size
1607 
1608 if __name__ == '__main__':
1609     parser = argparse.ArgumentParser()
1610     parser.add_argument('--weights', type=str, default='./yolov3.pt', help
    ='weights path')  # from yolov3/models/
1611     parser.add_argument('--img-size', nargs='+', type=int, default=[640,
    640], help='image size')  # height, width
1612     parser.add_argument('--batch-size', type=int, default=1, help='batch
    size')
1613     opt = parser.parse_args()
1614     opt.img_size *= 2 if len(opt.img_size) == 1 else 1  # expand
1615     print(opt)
1616     set_logging()
1617     t = time.time()
1618 
1619     # Load PyTorch model
1620     model = attempt_load(opt.weights, map_location=torch.device('cpu'))  #
    load FP32 model
1621     labels = model.names
1622 
1623     # Checks
```

```
1624     gs = int(max(model.stride))  # grid size (max stride)
1625     opt.img_size = [check_img_size(x, gs) for x in opt.img_size]  # verify
    img_size are gs-multiples
1626
1627     # Input
1628     img = torch.zeros(opt.batch_size, 3, *opt.img_size)  # image size
    (1,3,320,192) iDetection
1629
1630     # Update model
1631     for k, m in model.named_modules():
1632         m._non_persistent_buffers_set = set()  # pytorch 1.6.0
    compatibility
1633         if isinstance(m, models.common.Conv):  # assign export-friendly
    activations
1634             if isinstance(m.act, nn.Hardswish):
1635                 m.act = Hardswish()
1636             elif isinstance(m.act, nn.SiLU):
1637                 m.act = SiLU()
1638         # elif isinstance(m, models.yolo.Detect):
1639         #     m.forward = m.forward_export  # assign forward (optional)
1640     model.model[-1].export = True  # set Detect() layer export=True
1641     y = model(img)  # dry run
1642
1643     # TorchScript export
1644     try:
1645         print('\nStarting TorchScript export with torch %s...' % torch.
    __version__)
1646         f = opt.weights.replace('.pt', '.torchscript.pt')  # filename
1647         ts = torch.jit.trace(model, img)
1648         ts.save(f)
1649         print('TorchScript export success, saved as %s' % f)
1650     except Exception as e:
1651         print('TorchScript export failure: %s' % e)
1652
1653     # ONNX export
1654     try:
1655         import onnx
1656
1657         print('\nStarting ONNX export with onnx %s...' % onnx.__version__)
1658         f = opt.weights.replace('.pt', '.onnx')  # filename
1659         torch.onnx.export(model, img, f, verbose=False, opset_version=12,
    input_names=['images'],
1660                           output_names=['classes', 'boxes'] if y is None
    else ['output'])
1661
1662         # Checks
1663         onnx_model = onnx.load(f)  # load onnx model
```

```
1664            onnx.checker.check_model(onnx_model)  # check onnx model
1665            # print(onnx.helper.printable_graph(onnx_model.graph))  # print a
    human readable model
1666            print('ONNX export success, saved as %s' % f)
1667        except Exception as e:
1668            print('ONNX export failure: %s' % e)
1669
1670        # CoreML export
1671        try:
1672            import coremltools as ct
1673
1674            print('\nStarting CoreML export with coremltools %s...' % ct.
    __version__)
1675            # convert model from torchscript and apply pixel scaling as per
    detect.py
1676            model = ct.convert(ts, inputs=[ct.ImageType(name='image', shape=img
    .shape, scale=1 / 255.0, bias=[0, 0, 0])])
1677            f = opt.weights.replace('.pt', '.mlmodel')  # filename
1678            model.save(f)
1679            print('CoreML export success, saved as %s' % f)
1680        except Exception as e:
1681            print('CoreML export failure: %s' % e)
1682
1683        # Finish
1684        print('\nExport complete (%.2fs). Visualize with https://github.com/
    lutzroeder/netron.' % (time.time() - t))
1685
1686
1687 import argparse
1688 import logging
1689 import sys
1690 from copy import deepcopy
1691 from pathlib import Path
1692
1693 sys.path.append('./')  # to run '$ python *.py' files in subdirectories
1694 logger = logging.getLogger(__name__)
1695
1696 from models.common import *
1697 from models.experimental import MixConv2d, CrossConv
1698 from utils.autoanchor import check_anchor_order
1699 from utils.general import make_divisible, check_file, set_logging
1700 from utils.torch_utils import time_synchronized, fuse_conv_and_bn,
    model_info, scale_img, initialize_weights, \
1701     select_device, copy_attr
1702
1703 try:
1704     import thop  # for FLOPS computation
```

```
except ImportError:
    thop = None


class Detect(nn.Module):
    stride = None  # strides computed during build
    export = False  # onnx export

    def __init__(self, nc=80, anchors=(), ch=()):  # detection layer
        super(Detect, self).__init__()
        self.nc = nc  # number of classes
        self.no = nc + 5  # number of outputs per anchor
        self.nl = len(anchors)  # number of detection layers
        self.na = len(anchors[0]) // 2  # number of anchors
        self.grid = [torch.zeros(1)] * self.nl  # init grid
        a = torch.tensor(anchors).float().view(self.nl, -1, 2)
        self.register_buffer('anchors', a)  # shape(nl,na,2)
        self.register_buffer('anchor_grid', a.clone().view(self.nl, 1, -1, 1, 1, 2))  # shape(nl,1,na,1,1,2)
        self.m = nn.ModuleList(nn.Conv2d(x, self.no * self.na, 1) for x in ch)  # output conv

    def forward(self, x):
        # x = x.copy()  # for profiling
        z = []  # inference output
        self.training |= self.export
        for i in range(self.nl):
            x[i] = self.m[i](x[i])  # conv
            bs, _, ny, nx = x[i].shape  # x(bs,255,20,20) to x(bs,3,20,20,85)
            x[i] = x[i].view(bs, self.na, self.no, ny, nx).permute(0, 1, 3, 4, 2).contiguous()

            if not self.training:  # inference
                if self.grid[i].shape[2:4] != x[i].shape[2:4]:
                    self.grid[i] = self._make_grid(nx, ny).to(x[i].device)

                y = x[i].sigmoid()
                y[..., 0:2] = (y[..., 0:2] * 2. - 0.5 + self.grid[i].to(x[i].device)) * self.stride[i]  # xy
                y[..., 2:4] = (y[..., 2:4] * 2) ** 2 * self.anchor_grid[i]  # wh
                z.append(y.view(bs, -1, self.no))

        return x if self.training else (torch.cat(z, 1), x)

    @staticmethod
```

```
1746     def _make_grid(nx=20, ny=20):
1747         yv, xv = torch.meshgrid([torch.arange(ny), torch.arange(nx)])
1748         return torch.stack((xv, yv), 2).view((1, 1, ny, nx, 2)).float()
1749 
1750 
1751 class Model(nn.Module):
1752     def __init__(self, cfg='yolov3.yaml', ch=3, nc=None):  # model, input
    channels, number of classes
1753         super(Model, self).__init__()
1754         if isinstance(cfg, dict):
1755             self.yaml = cfg  # model dict
1756         else:  # is *.yaml
1757             import yaml  # for torch hub
1758             self.yaml_file = Path(cfg).name
1759             with open(cfg) as f:
1760                 self.yaml = yaml.load(f, Loader=yaml.FullLoader)  # model
    dict
1761 
1762         # Define model
1763         ch = self.yaml['ch'] = self.yaml.get('ch', ch)  # input channels
1764         if nc and nc != self.yaml['nc']:
1765             logger.info('Overriding model.yaml nc=%g with nc=%g' % (self.
    yaml['nc'], nc))
1766             self.yaml['nc'] = nc  # override yaml value
1767         self.model, self.save = parse_model(deepcopy(self.yaml), ch=[ch])
    # model, savelist
1768         self.names = [str(i) for i in range(self.yaml['nc'])]  # default
    names
1769         # print([x.shape for x in self.forward(torch.zeros(1, ch, 64, 64))
    ])
1770 
1771         # Build strides, anchors
1772         m = self.model[-1]  # Detect()
1773         if isinstance(m, Detect):
1774             s = 256  # 2x min stride
1775             m.stride = torch.tensor([s / x.shape[-2] for x in self.forward(
    torch.zeros(1, ch, s, s))])  # forward
1776             m.anchors /= m.stride.view(-1, 1, 1)
1777             check_anchor_order(m)
1778             self.stride = m.stride
1779             self._initialize_biases()  # only run once
1780             # print('Strides: %s' % m.stride.tolist())
1781 
1782         # Init weights, biases
1783         initialize_weights(self)
1784         self.info()
1785         logger.info('')
```

```
1786
1787    def forward(self, x, augment=False, profile=False):
1788        if augment:
1789            img_size = x.shape[-2:]  # height, width
1790            s = [1, 0.83, 0.67]  # scales
1791            f = [None, 3, None]  # flips (2-ud, 3-lr)
1792            y = []  # outputs
1793            for si, fi in zip(s, f):
1794                xi = scale_img(x.flip(fi) if fi else x, si, gs=int(self.
    stride.max()))
1795                yi = self.forward_once(xi)[0]  # forward
1796                # cv2.imwrite('img%g.jpg' % s, 255 * xi[0].numpy().
    transpose((1, 2, 0))[:, :, ::-1])  # save
1797                yi[..., :4] /= si  # de-scale
1798                if fi == 2:
1799                    yi[..., 1] = img_size[0] - yi[..., 1]  # de-flip ud
1800                elif fi == 3:
1801                    yi[..., 0] = img_size[1] - yi[..., 0]  # de-flip lr
1802                y.append(yi)
1803            return torch.cat(y, 1), None  # augmented inference, train
1804        else:
1805            return self.forward_once(x, profile)  # single-scale inference,
     train
1806
1807    def forward_once(self, x, profile=False):
1808        y, dt = [], []  # outputs
1809        for m in self.model:
1810            if m.f != -1:  # if not from previous layer
1811                x = y[m.f] if isinstance(m.f, int) else [x if j == -1 else
    y[j] for j in m.f]  # from earlier layers
1812
1813            if profile:
1814                o = thop.profile(m, inputs=(x,), verbose=False)[0] / 1E9 *
    2 if thop else 0  # FLOPS
1815                t = time_synchronized()
1816                for _ in range(10):
1817                    _ = m(x)
1818                dt.append((time_synchronized() - t) * 100)
1819                print('%10.1f%10.0f%10.1fms %-40s' % (o, m.np, dt[-1], m.
    type))
1820
1821            x = m(x)  # run
1822            y.append(x if m.i in self.save else None)  # save output
1823
1824        if profile:
1825            print('%.1fms total' % sum(dt))
1826        return x
```

```
1827
1828     def _initialize_biases(self, cf=None):  # initialize biases into Detect
    (), cf is class frequency
1829         # https://arxiv.org/abs/1708.02002 section 3.3
1830         # cf = torch.bincount(torch.tensor(np.concatenate(dataset.labels,
    0)[:, 0]).long(), minlength=nc) + 1.
1831         m = self.model[-1]  # Detect() module
1832         for mi, s in zip(m.m, m.stride):  # from
1833             b = mi.bias.view(m.na, -1)  # conv.bias(255) to (3,85)
1834             b.data[:, 4] += math.log(8 / (640 / s) ** 2)  # obj (8 objects
    per 640 image)
1835             b.data[:, 5:] += math.log(0.6 / (m.nc - 0.99)) if cf is None
    else torch.log(cf / cf.sum())  # cls
1836             mi.bias = torch.nn.Parameter(b.view(-1), requires_grad=True)
1837
1838     def _print_biases(self):
1839         m = self.model[-1]  # Detect() module
1840         for mi in m.m:  # from
1841             b = mi.bias.detach().view(m.na, -1).T  # conv.bias(255) to
    (3,85)
1842             print(('%6g Conv2d.bias:' + '%10.3g' * 6) % (mi.weight.shape
    [1], *b[:5].mean(1).tolist(), b[5:].mean()))
1843
1844     # def _print_weights(self):
1845     #     for m in self.model.modules():
1846     #         if type(m) is Bottleneck:
1847     #             print('%10.3g' % (m.w.detach().sigmoid() * 2))  #
    shortcut weights
1848
1849     def fuse(self):  # fuse model Conv2d() + BatchNorm2d() layers
1850         print('Fusing layers... ')
1851         for m in self.model.modules():
1852             if type(m) is Conv and hasattr(m, 'bn'):
1853                 m.conv = fuse_conv_and_bn(m.conv, m.bn)  # update conv
1854                 delattr(m, 'bn')  # remove batchnorm
1855                 m.forward = m.fuseforward  # update forward
1856         self.info()
1857         return self
1858
1859     def nms(self, mode=True):  # add or remove NMS module
1860         present = type(self.model[-1]) is NMS  # last layer is NMS
1861         if mode and not present:
1862             print('Adding NMS... ')
1863             m = NMS()  # module
1864             m.f = -1  # from
1865             m.i = self.model[-1].i + 1  # index
1866             self.model.add_module(name='%s' % m.i, module=m)  # add
```

```
1867            self.eval()
1868        elif not mode and present:
1869            print('Removing NMS... ')
1870            self.model = self.model[:-1]  # remove
1871        return self
1872
1873    def autoshape(self):  # add autoShape module
1874        print('Adding autoShape... ')
1875        m = autoShape(self)  # wrap model
1876        copy_attr(m, self, include=('yaml', 'nc', 'hyp', 'names', 'stride')
    , exclude=())  # copy attributes
1877        return m
1878
1879    def info(self, verbose=False, img_size=640):  # print model information
1880        model_info(self, verbose, img_size)
1881
1882
1883 def parse_model(d, ch):  # model_dict, input_channels(3)
1884    logger.info('\n%3s%18s%3s%10s  %-40s%-30s' % ('', 'from', 'n', 'params
    ', 'module', 'arguments'))
1885    anchors, nc, gd, gw = d['anchors'], d['nc'], d['depth_multiple'], d['
    width_multiple']
1886    na = (len(anchors[0]) // 2) if isinstance(anchors, list) else anchors
    # number of anchors
1887    no = na * (nc + 5)  # number of outputs = anchors * (classes + 5)
1888
1889    layers, save, c2 = [], [], ch[-1]  # layers, savelist, ch out
1890    for i, (f, n, m, args) in enumerate(d['backbone'] + d['head']):  # from
    , number, module, args
1891        m = eval(m) if isinstance(m, str) else m  # eval strings
1892        for j, a in enumerate(args):
1893            try:
1894                args[j] = eval(a) if isinstance(a, str) else a  # eval
    strings
1895            except:
1896                pass
1897
1898        n = max(round(n * gd), 1) if n > 1 else n  # depth gain
1899        if m in [Conv, Bottleneck, SPP, DWConv, MixConv2d, Focus, CrossConv
    , BottleneckCSP, C3]:
1900            c1, c2 = ch[f], args[0]
1901
1902            # Normal
1903            # if i > 0 and args[0] != no:  # channel expansion factor
1904            #     ex = 1.75  # exponential (default 2.0)
1905            #     e = math.log(c2 / ch[1]) / math.log(2)
1906            #     c2 = int(ch[1] * ex ** e)
```

```
            # if m != Focus:

            c2 = make_divisible(c2 * gw, 8) if c2 != no else c2

            # Experimental
            # if i > 0 and args[0] != no:  # channel expansion factor
            #     ex = 1 + gw  # exponential (default 2.0)
            #     ch1 = 32  # ch[1]
            #     e = math.log(c2 / ch1) / math.log(2)  # level 1-n
            #     c2 = int(ch1 * ex ** e)
            # if m != Focus:
            #     c2 = make_divisible(c2, 8) if c2 != no else c2

            args = [c1, c2, *args[1:]]
            if m in [BottleneckCSP, C3]:
                args.insert(2, n)
                n = 1
        elif m is nn.BatchNorm2d:
            args = [ch[f]]
        elif m is Concat:
            c2 = sum([ch[x if x < 0 else x + 1] for x in f])
        elif m is Detect:
            args.append([ch[x + 1] for x in f])
            if isinstance(args[1], int):  # number of anchors
                args[1] = [list(range(args[1] * 2))] * len(f)
        elif m is Contract:
            c2 = ch[f if f < 0 else f + 1] * args[0] ** 2
        elif m is Expand:
            c2 = ch[f if f < 0 else f + 1] // args[0] ** 2
        else:
            c2 = ch[f if f < 0 else f + 1]

        m_ = nn.Sequential(*[m(*args) for _ in range(n)]) if n > 1 else m(*
    args)  # module
        t = str(m)[8:-2].replace('__main__.', '')  # module type
        np = sum([x.numel() for x in m_.parameters()])  # number params
        m_.i, m_.f, m_.type, m_.np = i, f, t, np  # attach index, 'from'
    index, type, number params
        logger.info('%3s%18s%3s%10.0f  %-40s%-30s' % (i, f, n, np, t, args)
    )  # print
        save.extend(x % i for x in ([f] if isinstance(f, int) else f) if x
    != -1)  # append to savelist
        layers.append(m_)
        ch.append(c2)
    return nn.Sequential(*layers), sorted(save)
```

```
1950 if __name__ == '__main__':
1951     parser = argparse.ArgumentParser()
1952     parser.add_argument('--cfg', type=str, default='yolov3.yaml', help='
     model.yaml')
1953     parser.add_argument('--device', default='', help='cuda device, i.e. 0
     or 0,1,2,3 or cpu')
1954     opt = parser.parse_args()
1955     opt.cfg = check_file(opt.cfg)  # check file
1956     set_logging()
1957     device = select_device(opt.device)
1958 
1959     # Create model
1960     model = Model(opt.cfg).to(device)
1961     model.train()
1962 
1963     # Profile
1964     # img = torch.rand(8 if torch.cuda.is_available() else 1, 3, 640, 640).
     to(device)
1965     # y = model(img, profile=True)
1966 
1967     # Tensorboard
1968     # from torch.utils.tensorboard import SummaryWriter
1969     # tb_writer = SummaryWriter()
1970     # print("Run 'tensorboard --logdir=models/runs' to view tensorboard at
     http://localhost:6006/")
1971     # tb_writer.add_graph(model.model, img)  # add model to tensorboard
1972     # tb_writer.add_image('test', img[0], dataformats='CWH')  # add model
     to tensorboard
1973 
1974 
1975 # parameters
1976 nc: 80  # number of classes
1977 depth_multiple: 1.0  # model depth multiple
1978 width_multiple: 1.0  # layer channel multiple
1979 
1980 # anchors
1981 anchors:
1982   - [10,13, 16,30, 33,23]  # P3/8
1983   - [30,61, 62,45, 59,119]  # P4/16
1984   - [116,90, 156,198, 373,326]  # P5/32
1985 
1986 # darknet53 backbone
1987 backbone:
1988   # [from, number, module, args]
1989   [[-1, 1, Conv, [32, 3, 1]],  # 0
1990    [-1, 1, Conv, [64, 3, 2]],  # 1-P1/2
1991    [-1, 1, Bottleneck, [64]],
```

```
    [-1, 1, Conv, [128, 3, 2]],  # 3-P2/4
    [-1, 2, Bottleneck, [128]],
    [-1, 1, Conv, [256, 3, 2]],  # 5-P3/8
    [-1, 8, Bottleneck, [256]],
    [-1, 1, Conv, [512, 3, 2]],  # 7-P4/16
    [-1, 8, Bottleneck, [512]],
    [-1, 1, Conv, [1024, 3, 2]],  # 9-P5/32
    [-1, 4, Bottleneck, [1024]],  # 10
  ]

# YOLOv3-SPP head
head:
  [[-1, 1, Bottleneck, [1024, False]],
   [-1, 1, SPP, [512, [5, 9, 13]]],
   [-1, 1, Conv, [1024, 3, 1]],
   [-1, 1, Conv, [512, 1, 1]],
   [-1, 1, Conv, [1024, 3, 1]],  # 15 (P5/32-large)

   [-2, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 8], 1, Concat, [1]],  # cat backbone P4
   [-1, 1, Bottleneck, [512, False]],
   [-1, 1, Bottleneck, [512, False]],
   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, Conv, [512, 3, 1]],  # 22 (P4/16-medium)

   [-2, 1, Conv, [128, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P3
   [-1, 1, Bottleneck, [256, False]],
   [-1, 2, Bottleneck, [256, False]],  # 27 (P3/8-small)

   [[27, 22, 15], 1, Detect, [nc, anchors]],   # Detect(P3, P4, P5)
  ]


# parameters
nc: 80  # number of classes
depth_multiple: 1.0  # model depth multiple
width_multiple: 1.0  # layer channel multiple

# anchors
anchors:
  - [10,14, 23,27, 37,58]  # P4/16
  - [81,82, 135,169, 344,319]  # P5/32

# YOLOv3-tiny backbone
```

```
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [16, 3, 1]], # 0
   [-1, 1, nn.MaxPool2d, [2, 2, 0]], # 1-P1/2
   [-1, 1, Conv, [32, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]], # 3-P2/4
   [-1, 1, Conv, [64, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]], # 5-P3/8
   [-1, 1, Conv, [128, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]], # 7-P4/16
   [-1, 1, Conv, [256, 3, 1]],
   [-1, 1, nn.MaxPool2d, [2, 2, 0]], # 9-P5/32
   [-1, 1, Conv, [512, 3, 1]],
   [-1, 1, nn.ZeroPad2d, [[0, 1, 0, 1]]], # 11
   [-1, 1, nn.MaxPool2d, [2, 1, 0]], # 12
  ]

# YOLOv3-tiny head
head:
  [[-1, 1, Conv, [1024, 3, 1]],
   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, Conv, [512, 3, 1]], # 15 (P5/32-large)

   [-2, 1, Conv, [128, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 8], 1, Concat, [1]], # cat backbone P4
   [-1, 1, Conv, [256, 3, 1]], # 19 (P4/16-medium)

   [[19, 15], 1, Detect, [nc, anchors]], # Detect(P4, P5)
  ]


# parameters
nc: 80 # number of classes
depth_multiple: 1.0 # model depth multiple
width_multiple: 1.0 # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23] # P3/8
  - [30,61, 62,45, 59,119] # P4/16
  - [116,90, 156,198, 373,326] # P5/32

# darknet53 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [32, 3, 1]], # 0
```

```
    [-1, 1, Conv, [64, 3, 2]],  # 1-P1/2
    [-1, 1, Bottleneck, [64]],
    [-1, 1, Conv, [128, 3, 2]],  # 3-P2/4
    [-1, 2, Bottleneck, [128]],
    [-1, 1, Conv, [256, 3, 2]],  # 5-P3/8
    [-1, 8, Bottleneck, [256]],
    [-1, 1, Conv, [512, 3, 2]],  # 7-P4/16
    [-1, 8, Bottleneck, [512]],
    [-1, 1, Conv, [1024, 3, 2]],  # 9-P5/32
    [-1, 4, Bottleneck, [1024]],  # 10
  ]

# YOLOv3 head
head:
  [[-1, 1, Bottleneck, [1024, False]],
    [-1, 1, Conv, [512, [1, 1]]],
    [-1, 1, Conv, [1024, 3, 1]],
    [-1, 1, Conv, [512, 1, 1]],
    [-1, 1, Conv, [1024, 3, 1]],  # 15 (P5/32-large)

    [-2, 1, Conv, [256, 1, 1]],
    [-1, 1, nn.Upsample, [None, 2, 'nearest']],
    [[-1, 8], 1, Concat, [1]],  # cat backbone P4
    [-1, 1, Bottleneck, [512, False]],
    [-1, 1, Bottleneck, [512, False]],
    [-1, 1, Conv, [256, 1, 1]],
    [-1, 1, Conv, [512, 3, 1]],  # 22 (P4/16-medium)

    [-2, 1, Conv, [128, 1, 1]],
    [-1, 1, nn.Upsample, [None, 2, 'nearest']],
    [[-1, 6], 1, Concat, [1]],  # cat backbone P3
    [-1, 1, Bottleneck, [256, False]],
    [-1, 2, Bottleneck, [256, False]],  # 27 (P3/8-small)

    [[27, 22, 15], 1, Detect, [nc, anchors]],   # Detect(P3, P4, P5)
  ]
```